Programming Technology

for Bachelor of Engineering in Computer Engineering (IV Semester) Pokhara University

<u>Compiled By:</u>
Assistant Professor Madan Kadariya
HoD, Department of Information Technology Engineering
Nepal College of Information Technology (NCIT)
Balkumari, Lalitpur, Nepal

Table of Contents

List of Tables

List of Figures

1 Chapter 1: Introduction

1.1 Procedure Oriented Programming/Structured Programming

- A computer program is a set of instructions for a computer to perform a specific task.
- The traditional programming languages like C, FORTRAN, Pascal, and Basic are Procedural Programming Languages.
- A procedural program is written as a list of instructions, telling the computer, step-by-step, what to do: Get two numbers, add them and display the sum.
- The problem is divided into a number of functions. The primary focus of programmer is on creating functions.
- While developing functions, very little attention is given to data that are being used by various functions.
- Procedural programming is fine for small projects. It does not model real world problems well.
- This is because functions are action oriented and does not really correspond to the elements of the problems.

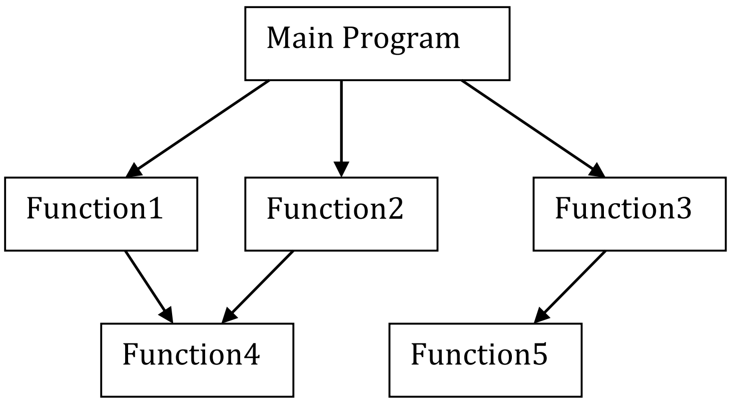


Figure 1: Typical structure of Procedural Oriented Program

1.1.1 Features of Structured Programming Language

- Emphasis is on algorithm (step by step description of problem solving) to solve a problem. Each step of algorithm is converted into corresponding instruction or code in programming language.
- Large program is divided into a number of functions, with each function having a clearly defined purpose and a clearly defined interface to other functions in the program. The integration of these functions constitutes a whole program.
- Uses top-down approach in program design**.** In top-down programming, we conceive the program as a whole as a process to be decomposed. We decompose the problem into sub-problems with the following three characteristics:
  1. The sub-problem stands alone as a problem in its own right
  2. The sub-problem appears to be solvable, though we may not have an immediate solution, and
  3. If all of the sub-problems in the decomposition are solvable, then we know a composition that will put the solutions of the sub-problems together as a solution of the original.

- In multi-function program, data items that are important and that are used by more than one functions, are placed as global so that they may be accessed by all the functions. Global data are more vulnerable to accidental changes.
- Data move openly around the system from function to function. Information is passed between functions using parameters or arguments.

1.2 Object-Oriented Programming Language

- In Object-Oriented Programming, a program is decomposed into a number of entities called objects.
- *Data* and *the functions that operate on that data* are combined into that object. OOP mainly focus on data rather than procedure.
- It considers data as critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions.

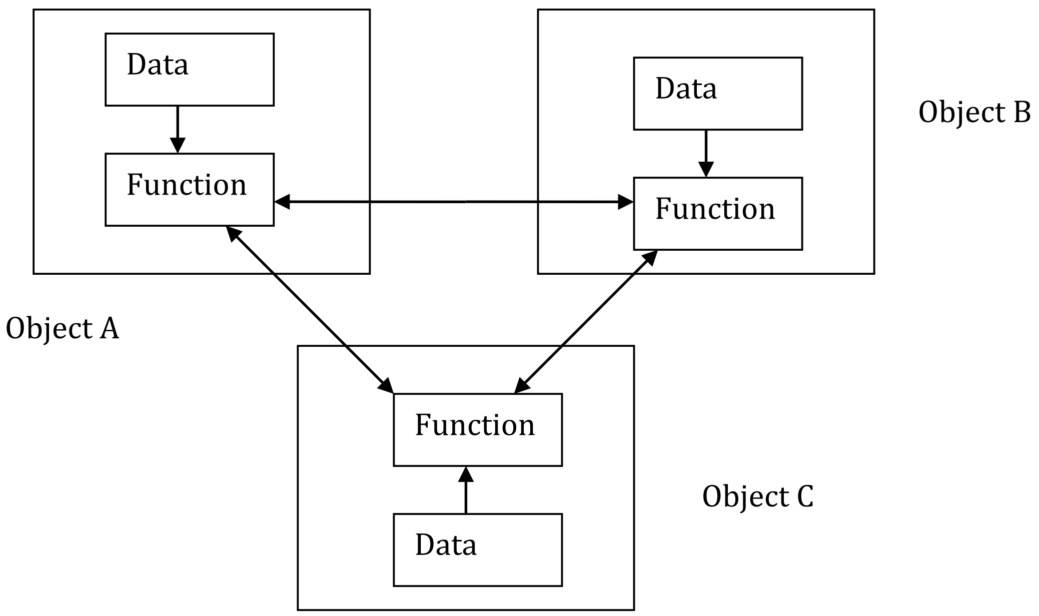


Figure 2: Organization of data and functions in OOP

.

1.2.1 Features of Object Oriented Programming Language

- Emphasis is on the data rather than procedure. It focuses on security of data from unauthorized modification in the program.
- A program is divided into a number of objects. Each object has its own data and functions. These functions are called member functions and data are known as member data.
- Data is hidden and cannot be accessed by external functions.
- Follows bottom-up approach in program design. The methodology for constructing an object-oriented program is to discover the related objects first. Then, appropriate class is formed for similar objects. These different classes and their objects constitute a program.
- Object may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.

- Once a class has been written, created and debugged, it can be used in a number of programs without modification i.e. a same code can be reused. It is similar to the way a library functions in a procedural language.

1.3 Basic Characteristics of Object-Oriented Languages

1.3.1 Objects

They are basic runtime entities of an object-oriented system. An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain. An object may be

Tangible Things	as a car, printer, ...
Roles	as employee, boss, ...
Incidents	as flight, overflow, ...
Interactions	as contract, sale, ...
Specifications	as colour, shape, …
Elements of Computer- User environment	as Windows, Menu etc

In object-oriented programming, problem is analysed in terms of objects. Programming object should be chosen such that they match closely with the real world objects. Objects take up space in memory. During program execution, objects interact by sending message to one another. Each object contains data and code to manipulate them. It can be defined as
Object = Data + Methods or functions

1.3.2 Classes

A class may be defined as a collection of similar objects. In other words, it is a general name for all similar objects. For example, mango, apple, banana all may be described under the common name **fruits.** Thus, **fruits** is class name for objects like mango, apple, and banana. In fact, a class is user defined data type and behaves like the built-in data types of programming languages. A class serves as a blue print or a plan or a template. It specifies what data and what functions will be included in objects of that class. Once a class has been defined, we can create any number of objects belonging to that class. If **fruits** has been defined as a class, then statements

```
fruits mango;
fruits apple;
```

will create objects mango and apple belonging to the class fruits. This is similar to creating variables of built-in data types.

1.3.3 Data abstraction and encapsulation

Wrapping of data and functions into a single unit is known as data encapsulation. Data and functions of object are encapsulated using class. With data encapsulation, data is not accessible to the outside world, only the functions which are wrapped in the class can access them. These functions act as an interface between object's data and the program. This insulation of data is called data hiding.
The act of representing essential features without including background details or explanations is called abstraction. Classes use the concept of abstraction. They are defined as the list of abstract attributes such as site, amount and functions to operate on these attributes. They encapsulate all the properties of the object to be created. As classes use the concept of abstraction, they are also called as abstract data types.

A result of abstraction, when two different people interact with the same object, they often deal with a different subset of attributes. When I drive my car, for example, I need to know the speed of the car and the direction it is going. Because the car is using an automatic transmission, I do not need to know the revolutions per minute (RPMs) of the engine, so I filter this information out. On the other hand, this information would be critical to a racecar driver, who would not filter it out.

1.3.4 Inheritance

Inheritance is the process by which objects of one class acquires the properties of objects of another class. Inheritance allows us to create classes which are derived from other classes, so that they automatically include their "parent's" members, plus their own. Thus, a class can be divided into a number of sub classes. The derived classes are known as sub classes (or child classes) and original classes are called base classes or parent classes. For example, a class of animals can be divided into mammals, amphibians, insects, birds and so on. The class of vehicles can be divided into cars, trucks, buses and motorcycles. Each sub class shares common characteristics with the class from which it is derived. Cars, trucks, buses and motorcycles all have wheel and a motor; these are the defining characteristics of vehicles. In addition to these shared characteristics, each sub class also has its own characteristics: buses have seats for many people while trucks have space for heavy loads.

The concept of inheritance provides the idea of reusability, means additional features can be added to an existing class without modifying it. In above example, the class ‘buses’ can be inherited from base class ‘vehicles’. Then, all features of vehicles class are also of class ‘buses’. Thus, we do not need to mention common properties to class ‘buses’. The only special features of buses are included in class ‘buses’. The common properties are shared from class ‘vehicles’. Thus,

features of class ‘buses’= special features of class ‘buses’+ common features of class ‘vehicles’

1.3.5 Reusability

Object-oriented programming uses concept of reusability. Reusability implies the reuse of existing code in another program without modification to it. The concept of inheritance provides an idea of reusability in OOP. Once a class has been written, created and debugged, it can be used in another program. A programmer can use an existing class and without modifying it, add additional features and capabilities to it. This is done by deriving a new class from existing one. The new class will inherit the capabilities of the old one but is free to add new features of its own.

1.3.6 Creating new data types

Object-oriented programming gives the programmer a convenient way to construct new data type. Creating a class in object oriented programming can be considered as creating new data types. As we can define different variables of built-in data types like int, float, char, we can also create various objects of classes in similar manner. For example, if **man** is class name defined in the program, then ***man ram;*** will create an object ram of type man. Here man is a user defined data type. A class name specifies what data and what functions will be included in objects of that class.

1.3.7 Polymorphism and overloading:

The property of object-oriented programming **polymorphism** is ability to take more than one form in different instances. For example, same function name can be used for different purposes. Similarly, same operator can be used for different operations. O**verloading (both function overloading and operator overloading)** is a kind of polymorphism. If an operator exhibits different behaviors in different instances, then it is known as operator overloading. The type of behavior depends upon the type of the data used in the operation. Similarly, if a same function name is used to perform different types of tasks, then it is known as function overloading. The task to be performed by the function is determined by number of arguments and type of arguments.

1.4 Applications and Benefits of using OOP

Applications of using OOP:
Main application areas of OOP are

- User interface design such as windows, menus.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation system etc

1.4.1 Benefits of OOP

The main advantages are

- It is easy to model a real system as programming objects in OOP represents real objects. The objects are processed by their member data and functions. It is easy to analyze the user requirements.
- With the help of inheritance, we can reuse the existing class to derive a new class such that redundant code is eliminated and the use of existing class is extended. This saves time and cost of program.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- It is possible to have multiple instances of an object to co-exist without any interference i.e. each object has its own separate member data and function.

1.5 Event Oriented Programming

Event oriented Programming means that the program everything as a response to an event, instead of a top down program where the ordering of the code determines the order of execution.

Program code based on what happens when the user does something. Events can be mouse moves, mouse clicks or double-clicks, keystrokes, changes made in textboxes, timing based on a timer, etc. Note that this is not an exhaustive list of events, just a small sampling. It is quite a bit different from top-down coding. An event-oriented language implies that an application (the computer program) waits for an event to occur before taking any action.

1.5.1 Difference Between Object Oriented and Event Oriented Language

Object oriented programming focuses on performing actions and manipulation of data that is encapsulated in objects within a sequential series of steps while event driven is more dynamic and relies on event triggering and event handling to determine the sequencing of the program. Event driven programs can have threads that perform actions based upon triggers/event in program. Example: fire sprinkler system; let's say there is a sensor for detecting a fire and sprinkler system used to extinguish the fire. If we used only object oriented, we would create an endless loop that constantly checks the status of the sensor if it detects a fire we turn on the sprinkler, the program would work but it wouldn't be much use for any other task. In an event driven program, we could set an event that triggers a thread when the fire sensor activates and we can create an event handler that would process the thread in order to tell the sprinkler system to turn on. In this fashion, our event program could also have other functionality, as it is not stuck in an endless loop.

1.6 Aspect oriented programming

What Is an Aspect?

When thinking of an object and its relationship to other objects we often think in terms of inheritance. We define some abstract class; let us use a Dog class as an example. As we identify similar classes but with unique behaviors of their own, we often use inheritance to extend the functionality. For instance, if we identified a **Poodle** we could say a **Poodle** Is A Dog, so **Poodle** inherits Dog. So far so good, but what happens when we define another unique behavior later on that we label as an **Obedient Dog**? Surely not all Dogs are **obedient**, so the Dog class cannot contain the obedience behavior. Furthermore, if we were to create an Obedient Dog class that inherited from Dog, then where would a Poodle fit in that hierarchy? A **Poodle** is A Dog, but a Poodle may or may not be obedient; does Poodle, then, inherit from Dog, or does Poodle inherit from Obedient Dog? Instead, we can look at obedience as an aspect that we apply to any type of Dog that is obedient, as opposed to inappropriately forcing that behavior in the Dog hierarchy.

In software terms, aspect-oriented programming allows us the ability to apply aspects that alter behavior to classes or objects independent of any inheritance hierarchy. We can then apply these aspects either during runtime or compile time. It is easier to demonstrate AOP by example then to describe it. To start, though, it is important to define four key AOP terms I will use repeatedly:
Joinpoint—Well defined points in code that can be identified.
Pointcut—A way of specifying a Joinpoint by some means of configuration or code.
Advice—A way of expressing a cross cutting action that needs to occur.
Mixin—An instance of a class to be mixed in with the target instance of a class to introduce new behavior.

To better understand these terms, think of a ***joinpoint*** as a defined point in program flow. A good example of a ***joinpoint*** is the following: when code invokes a method, that point at which that invocation occurs is considered the ***joinpoint***. The ***pointcut*** allows us to specify or define the ***joinpoints*** that we wish to intercept in our program flow. A ***Pointcut*** also contains an advice that is to occur when the ***joinpoint*** is reached. So if we define a ***Pointcut*** on a particular method being invoked, when the invocation occurs or the ***joinpoint*** is invoked, it is intercepted by the AOP framework and the ***pointcut's*** advice is executed. An advice can be several things, but we should most commonly think of it as another method to invoke. So when we invoke a method with a ***pointcut***, our advice to execute would be another method to invoke. This advice or method to invoke could be on the object whose method was intercepted or on another object that we mixed in.

Aspect-Oriented Programming (AOP) complements OO programming by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements. Just as objects in the real world can change their states during their lifecycles, an application can adopt new characteristics as it develops.

1.7 Subject Oriented Programming

Subject-oriented programming is an enhancement of object-oriented programming that allows decentralized class definition. An application developer who needs new operations associated with classes can implement them him/herself, not by editing existing code for the classes, but as a separate collection of class definitions called a subject. Multiple subjects can be composed to yield a complete suite of applications; class definitions within the subjects will be combined so as to satisfy the needs of all the applications in the suite. Neither source code access nor recompilation is required to perform this composition, allowing extension and composition of object-code-only applications.

Without eliminating the advantages of encapsulation, this approach eliminates the need for class ownership. An application developer can write all the code needed for the application, irrespective of which classes are involved, without
Leaving development requirements on others. The cost of this flexibility is a small run-time overhead on operation calls.

Subject-oriented programming gets its name from the fact that each subject defines a subjective view of objects: the particular operations and internal data that that subject requires the objects to have. Subject-oriented programming supports decentralization in time as well as in space. The developers of an application can program extensions to it as separate subjects to be composed with the base application, perhaps in multiple configurations. This leads to requirement-based development: the code that implements a new requirement is built as a coherent subject rather than being interleaved amongst other application code in a manner that makes it difficult to identify and maintain.

1.8 Integrated Development Environment

An integrated development environment (IDE) or interactive development environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor; build automation tools and a debugger. Several modern IDEs integrate with Intelli-sense coding features.

Some examples of IDEs are: Microsoft Visual Studio, Eclipse, NetBeans, Delphi, JBuilder, Borland C++ Builders, FrontPage, Dreamweaver etc.

1.9 Visual Programming

In modern times, few inventions have affected everyday life as much as the computer. The earliest computers were gigantic machine that filled entire buildings and were difficult to use. Those were the days when using computers was the privilege of the chosen few who mastered its peculiarities.

As the time went by, computers have evolved and been put to good use in a variety of areas. Today computers dominate every sphere of life be it entertainment, education, banking, insurance, research, medicine, design or manufacturing.

One of the primary reason for the immense popularity of computers is the speed with which they can accomplish specified tasks. However, computer applications are not only easy to use.
Any application has two parts:

- User Interface: This is the screen displayed by the application. We interact with an application via the interface. The application asks for and we provide the necessary information using the interface.
- Program: how the application performs the specified task. This is what goes on in the background. We do not know how the application performs a specified task.

To present an analogy from the real world, consider how we use a television. We use the ON/OFF switch to put it on, volume control to increase/decrease the volume, channel selector to switch channels etc. These controls represent the interface while what happens internally when we use any of the controls constitutes the program.

All interaction between the user and the application are via the user interface. Thus , for any application to be successful it needs to have a good user interface.
A good user interface will be

- Easy to learn
- Easy to use
- Attractive

In Character User Interface (CUI) based applications, text was the medium of information exchange. The application displayed text that promoted the user for the required information. It is specified the valid options. The user than responded to these prompts by specifying the required information. Such application is not very easy to use, essentially because the user cannot react instinctively to this interface.

The disadvantage of CUI application was that with each new application the user had to spend time and effort getting used to the way it worked.

Advent of Graphical User Interface (GUI)
With the advent of graphical operating systems like windows, the world of computing went through a dramatic change. It opened up a whole new world of graphics.

Interface that use graphics came to be known as Graphical User Interface (GUI). These became very popular because the user could identify with the graphics displayed on screen. In every day life too, we associate more with imagery than words. The print icon of any windows based application invokes the mental image of printer.

The primary requirement of an interface is that it is easy to use. We must be able to react intuitively to an interface presented to us. This is possible if the interface presented is such that it emulates real life.

1.9.1 Advantages of Visual Programming

Visual programming enables visual development of graphical user interface. Such user interfaces are easy to use and easy to learn.

1.9.1.1 Ready to Use Component

- One of the principal advantages of visual programming environment is that the programmer need not write code to display the required component.

- The Visual programming environment displays a list of available components. The programmer picks up the required components from this list.
- The components can be moved, resized and even deleted, if so required.
- There is no restriction in the number of controls that can be placed this way.

1.9.1.2 Built-in Code

The interface components provided by the visual programming environment have some code built into them.

For example, a button 'knows' when it has been clicked upon. In the case of conventional programming tools, the programmer has to write code to determine the component that has been clicked and then execute the appropriate code.

1.9.2 Disadvantages of Visual Programming

While Visual programming makes it very simple to create complex user interfaces, it suffers from some disadvantages.

- As the name implies, the entire process of developing an application using a visual development environment is visual. Thus, the development environment in itself is highly graphical in nature and therefore requires more memory.
- Visual development environments require computers of higher configuration in comparison to the conventional programming tools.
  - Larger capacity hard disk
  - More RAM
  - Faster Processor
- Primarily, Visual development environment can be used only with GUI operating system such as windows.

1.9.3 Components of Graphical Visual Interface

Following are the popular components of Visual programming

1. **Window:** A window sometimes also called a form is the most important of all the visual interface components. A window plays the role of the canvas in a painting. Without the canvas there is no painting. Similarly, without the window there is no user interface. The components that make up the user interface have to be placed in the window and cannot exist independent of the window.
2. **Buttons:** A button is used to initiate an action. The text on the button is indictive of the action that it will initiate. Clicking on a button initiate the action associated with the button.
3. **Text boxes**: Text boxes are used to accept information from the user. The user interface will display one text for each piece of information
4. **List Boxes or Pop-up Lists**: List boxes are used to present the user with the possible options. The user can select one or more of the listed options.
5. **Label**: The label is used to place text in a window. Typically label is used to identify controls that do not have caption properties of their own.
6. **Radio button or Option Button**: The radio buttons, also referred as option buttons, are used when the user can select only one of multiple options.

1.10 Document View Architecture

In the early days of MFC (Microsoft Foundation Class), applications were built very much like the sample programs. An application had two principal components: an **application object** representing the application itself, and a **window object** representing the application's window. The application object's primary duty was to create the **window object**, and the window object, in turn, processed messages. Other than the provision of general-purpose classes such as CString and CTime to represent non-Windows objects, MFC was little more than a thin wrapper around the Windows API. It grafted an object-oriented interface onto windows, dialog boxes, device contexts, and other objects already present in Windows in one form or another.

MFC (Microsoft Foundation Class) 2.0 changed the way that Windows-based applications are written by introducing the document/view architecture. This architecture is carried through to MFC 4.0. In a document/view application, a **document object** represents an application's data and one or more **view objects** represent views of that data. The document and **view objects** work together to process the user's input and draw textual and graphical representations of the resulting data. MFC's CDocument class serves as the base class for all document objects, while the CView class and its derivatives serve as base classes for view objects. The top-level window, which is derived from either CFrameWnd or CMDIFrameWnd, is no longer the focal point for message processing, but serves primarily as a container for views, toolbars, status bars, and other objects.

The document/view architecture simplifies the development process. Code to perform routine tasks such as prompting the user to save unsaved data before a document is closed is provided for us by the framework. So is code to transform our application's documents into OLE containers, simplify printing, use splitter windows to divide a window into two or more panes, and more.

MFC supports two types of document/view applications. The first is single-document interface (SDI) applications, which support just one open document at a time. The second is multiple-document interface (MDI) applications, which permit the user to have two or more documents open concurrently. MDI apps support multiple views of each document, but SDI apps are generally limited to one view per document.

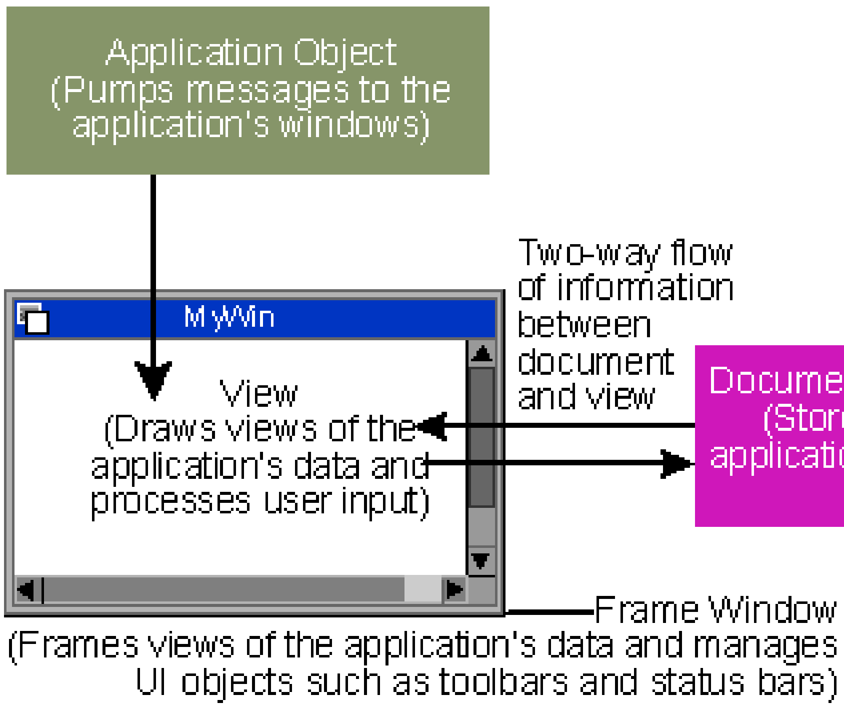

Application Object
(Pumps messages to the application's windows)
MyWin
View
(Draws views of the application's data and processes user input)
Two-way flow of information between document and view
Document Object
(Stores the application's data)
Frame Window
(Frames views of the application's data and manages UI objects such as toolbars and status bars)

Figure 3: Document View Architecture

The architecture's main elements are the Document and the View classes. The Document class can be thought of as a generic container for any kind of data that an application uses, such as baseball scores, images, files names, text documents, etc. This class derives from CDocument. Its main purpose is to load the data when requested, to keep the data updated when a user changes it, and to save out the data to disk, database, or any other source.

The View class functions to show the user the data of the application in a formatted or stylized way: This class derives from CView, and it is used to present the data to the user, take input in the form of mouse and keyboard events from the user, alert the document of the changes to the data that the user has requested, and update itself when the data changes.

- MFC document object reads and writes data to persistent storage.
- The document may also provide an interface to the data wherever it resides (such as in a database).
- A separate view object manages data display, from rendering the data in a window to user selection and editing of data.
- The view obtains display data from the document and communicates back to the document any data changes.

1.10.1 Key Classes of Document/View Architecture

The **CDocument** (or COleDocument) class supports objects used to store or control our program's data and provides the basic functionality for programmer-defined document classes. A document represents the unit of data that the user typically opens with the Open command on the File menu and saves with the Save command on the File menu.

The **CView** (or one of its many derived classes) provides the basic functionality for programmer-defined view classes. A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen and interprets user input as operations upon the document. The view also renders the image for both printing and print preview.

CFrameView (or one of its variations) supports objects that provides the frame around one or more views of a document.

CDocTemplate (or CSingleDocTemplate or CMultiDocTemplate) supports an object that coordinates one or more existing documents of a given type and manages creating the correct document, view, and frame window objects for that type.

The following figure shows the relationship between a document and its view.

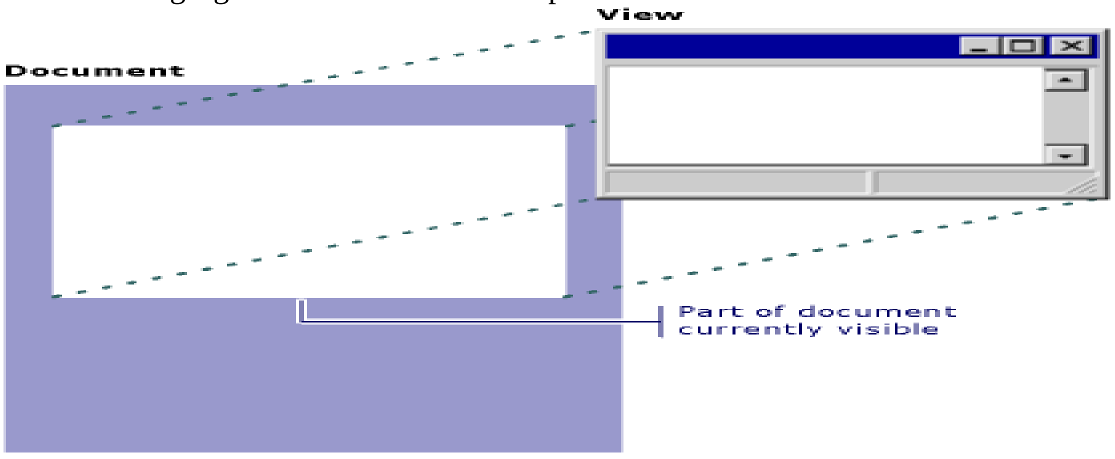


Figure 4: Document View Architecture

- The document/view implementation in the class library separates the data itself from its display and from user operations on the data.
- All changes to the data are managed through the document class. The view calls this interface to access and update the data.

A document template creates documents, their associated views, and the frame windows that frame the views. The document template is responsible for creating and managing all documents of one document type.

1.10.2 Advantages of the Document/View Architecture

The key advantage to using the MFC document/view architecture is that the architecture supports multiple views of the same document particularly well.
Suppose our application lets users view numerical data either in spreadsheet form or in chart form. A user might want to see simultaneously both the raw data, in spreadsheet form, and a chart that results from the data. We display these separate views in separate frame windows or in splitter panes within a single window. Now suppose the user can edit the data in the spreadsheet and see the changes instantly reflected in the chart.

In MFC, the spreadsheet view and the chart view would be based on different classes derived from **CView**. Both views would be associated with a single document object. The document stores the data (or perhaps obtains it from a database). Both views access the document and display the data they retrieve from it.

When a user updates one of the views, that view object calls **CDocument::UpdateAllViews**. That function notifies all of the document's views, and each view updates itself using the latest data from the document. The single call to **UpdateAllViews** synchronizes the different views.

This scenario would be difficult to code without the separation of data from view, particularly if the views stored the data themselves. With document/view, it's easy. The framework does most of the coordination work for us.

1.10.3 Alternatives to the Document/View Architecture

MFC applications normally use the document/view architecture to manage information, file formats, and the visual representation of data to users. For the majority of desktop applications, the document/view architecture is appropriate and efficient application architecture. This architecture separates data from viewing and, in most cases, simplifies our application and reduces redundant code.

However, the document/view architecture is not appropriate for some situations. Consider these examples:

- If we are porting an application written in C for Windows, we might want to complete our port before adding document/view support to our application.
- For writing a lightweight utility, we might find that we can do without the document/view architecture.
- If an original code already mixes data management with data viewing, moving the code to the document/view model is not worth the effort because we must separate the two.

2 Chapter 2: Programming Architecture

2.1 Model View Controller (MVC)

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and Interactions between classes or objects, without specifying the final application classes or objects that are involved. One of these design patterns is Model-View-Controller (MVC). The programming language Smalltalk first defined the MVC concept it in the 1970's. Since that time, the MVC design idiom has become commonplace, especially in object-oriented systems.

2.1.1 The Architecture

In a MVC application as having three main layers: **presentation (UI), application logic, and resource management**. In MVC, the presentation layer is split into controller and view. The most important separation is between presentation and application logic. The View/Controller split is less so. MVC encompasses more of the architecture of an application than is typical for a design pattern. Hence the term architectural pattern may be useful, or perhaps an aggregate design pattern.

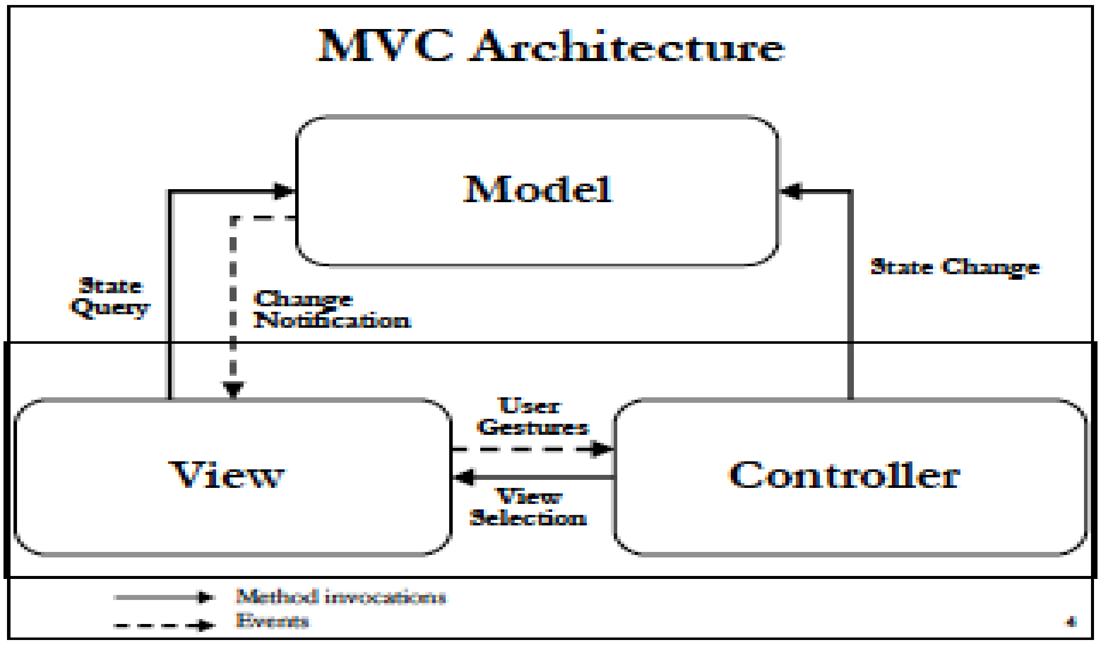


Figure 5: Model View Controller

1. **Model:** The domain-specific representation of the information on which the application operates. The model is another name for the application logic layer (sometimes also called the domain layer). Application (or domain) logic adds meaning to raw data (e.g., calculating if today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items). Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the resource management layer because it is understood to be underneath or encapsulated by the Model.
2. **View:** Renders the model into a form suitable for interaction, typically a user interface element. MVC is often seen in web applications, where the view is the

HTML page and the code which gathers dynamic data for the page.

3. **Controller**: Processes and responds to events, typically user actions, and may invoke changes on the model and view.

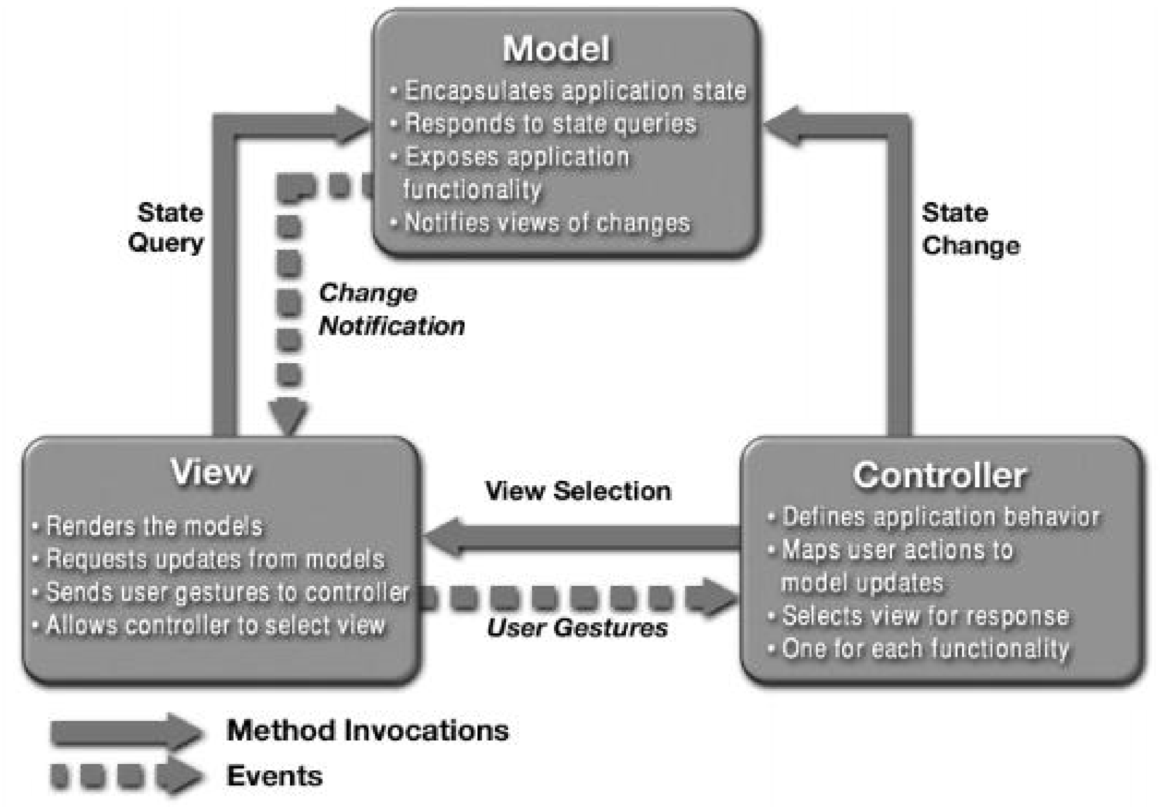

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 6: Summary of relationship between Model, View and Controller

Though MVC comes in different flavours, the control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button)
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
5. The user interface waits for further user interactions, which begins the cycle anew.

2.2 Client Server Model

Early computer system can best be described as monolithic system in which all processing occurred on the same machine. User interacted with the monolithic system through dumb terminal. It is desirable to move away from such monolithic system and divide the work between several different computers- this change signaled the birth of client/server systems.

All major software system relies on a persistent storage mechanism for maintaining state between program invocations. In such system, it is possible to clearly separate processing into two distinct areas

1. Processing that supports the storage and retrieval of persistent data as well as concurrent access to persistent storage.
2. All other processing required within software system

Splitting the work among multiple machines was not the motivation for dividing client/server technology. **Business data** is the most important asset that most companies posses. Maintaining database on server separate from the client allows for **greater security, reliability and performance** with regard to data storage while simultaneously reducing the high cost associated with monolithic system.

The client–server model is a distributed application structure in computing that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a **computer network** on separate hardware, but both client and server may reside in the same system. A server is a host that is running one or more server programs which share their resources with clients. A **client does not share any of its resources**, but **requests a server's content** or service function. Clients therefore initiate communication sessions with servers that await incoming requests.

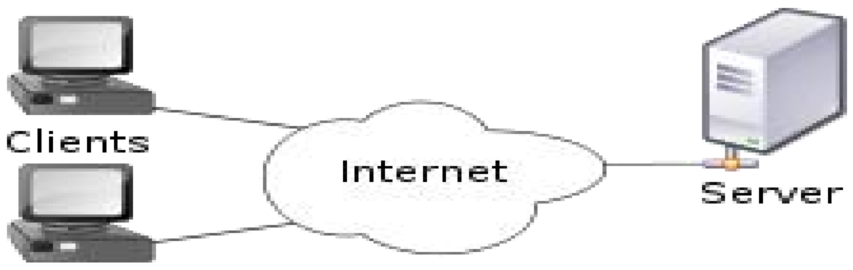


Figure 7: Client Server Model

Types of Server

There are two types of servers you can have:

1. **Iterative Server:** This is the simplest form of server where a server process serves one client and after completing first request then it takes request from another client. Meanwhile another client keeps waiting.

2. **Concurrent Servers:** This type of server runs multiple concurrent processes to serve many request at a time. Because one process may take longer and another client cannot wait for so long.

2.2.1 2-tier Client Server Architecture

In this architecture, client directly interacts with the server. 2-tier architecture is used to describe client/server systems where the client requests resources and the server responds directly to the request, using its own resources. This means that the server does not call on another application in order to provide part of the service.

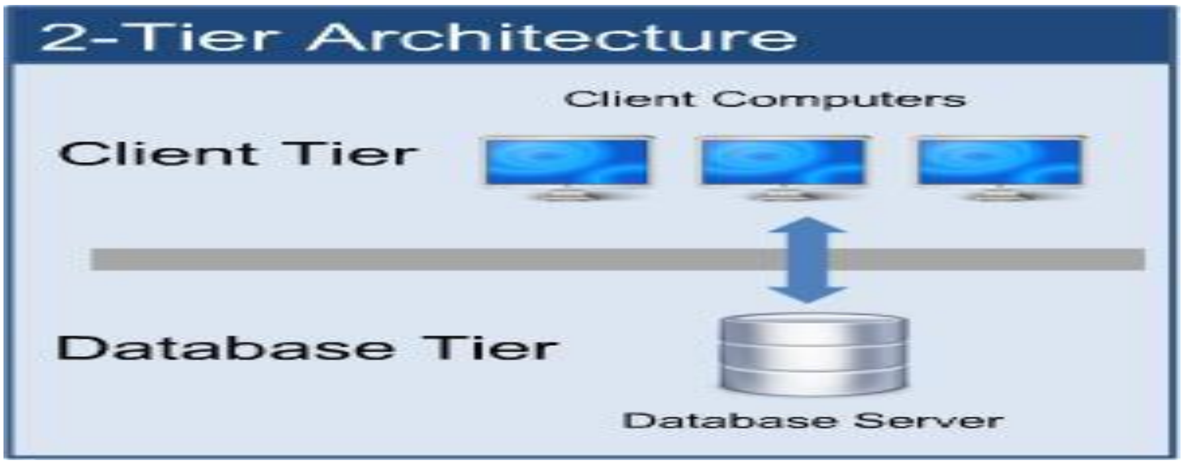


Figure 8: 2-tier client server architecture

2-tier Pros and Cons

Advantages	Disadvantage
Development Issues: • Simple structure • Easy to setup and maintain	*Development Issues:* • Complex application rules difficult to Implement in database server – requires more code for the client • Complex application rules difficult to implement in client and have poor performance • Changes to business logic not automatically enforced by a server – changes require new client side software to be distributed and installed
Performance: • Adequate performance for low to medium volume environments • Business logic and database are physically close, which provides higher performance.	*Performance:* • Inadequate performance for medium to high volume environments, since database server is required to perform business logic. This slows down database operations on database server.

Table 1: Pros and Cons of 2 tier CS architecture

Fat-Client/Thin-Server: Client does most of the processing. User Interface and all data processing/Business logic are done on client. The Data Service Layer is the actual Database Server, which handles the storage or services the data.

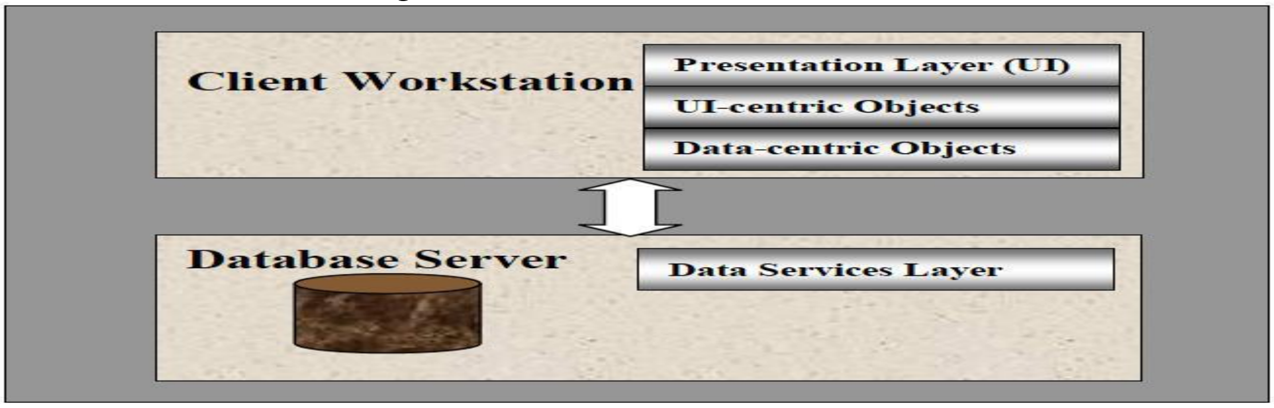


Figure 9: Fat Client Thin Server

Thin-Client/Fat-Server: Client has less processing, usually UI & UI-Centric *Business Objects*. The *Data-Centric Business Objects,* which handles the data access code, resides on the Data Service Layer or Database Server.

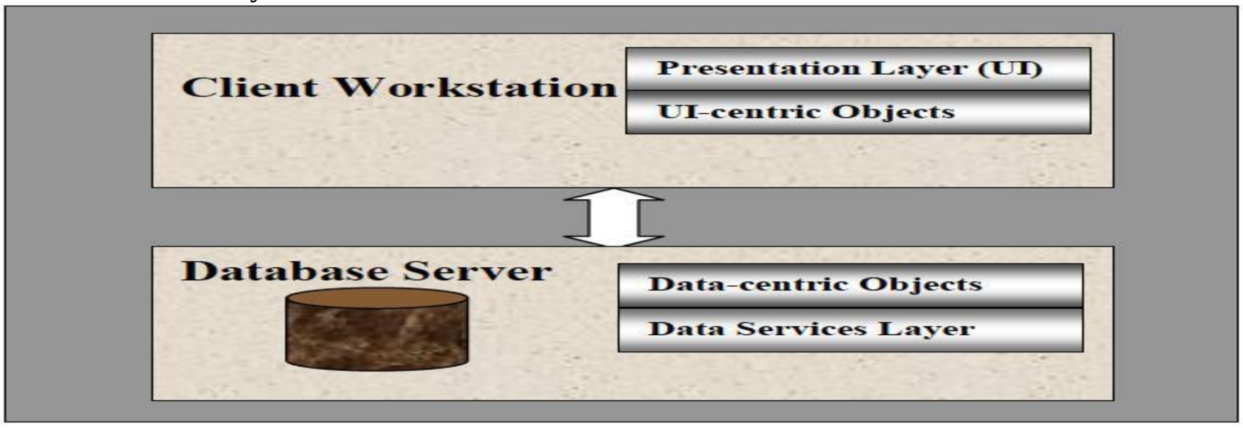


Figure 10: Thin client Fat Server

2.2.2 3-tier Architecture

In this architecture, one more software sits in between client and server. This middle software is called **middleware**. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after doing required authentication it passes that request to the server. Then server does required processing and sends response back to the middleware and finally middleware passes this response back to the client.

Middle tier is also called as business logic tier or service tired. In the simple client-server solution the client was handling the business logic and that makes the client "**thick**". A thick client means that it requires heavy traffic with the server, thus making it difficult to use over slower network connections like Internet and Wireless (LTE, 3G, or Wi-Fi).

By introducing the middle layer, the client is only handling presentation logic. This means that only little communication is needed between the client and the middle tier making the client "**thin**" or "thinner". An example of a thin client is an Internet browser that allows you to see and provide information fast and almost with no delay.

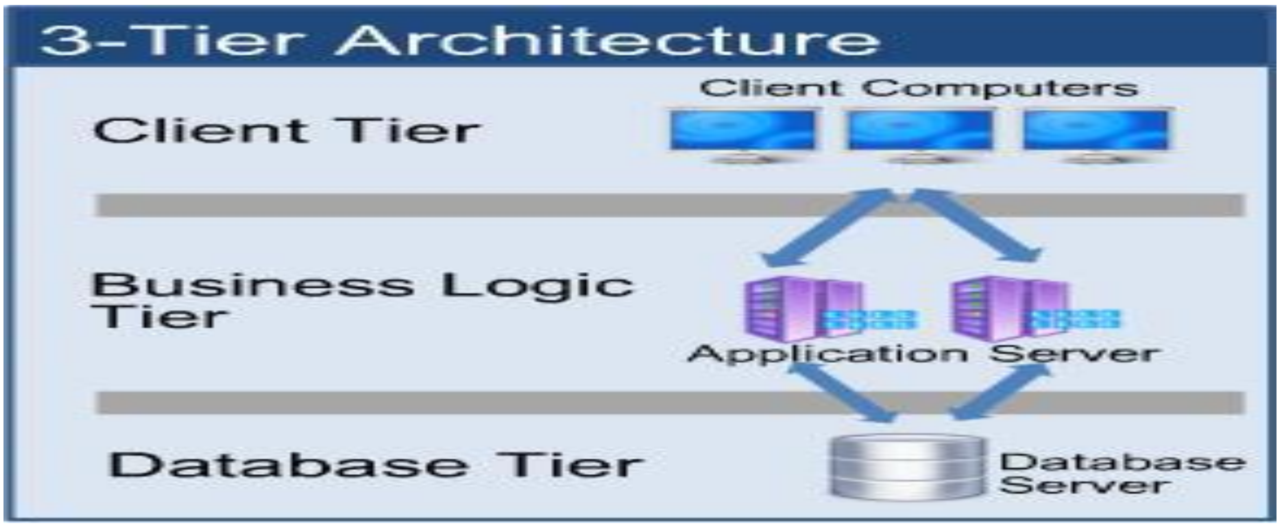


Figure 11: 3-tier client server architecture

3-tier pros and cons

Advantage	**Disadvantage**
Development Issues: • Complex application rules easy to implement in application server • Business logic off-loaded from database server and client, which improves performance • Changes to business logic automatically enforced by server – changes require only new application server software to be installed • Application server logic is portable to other database server platforms by virtue of the application software	*Development Issues:* • More complex structure • More difficult to setup and maintain.
Performance: • Superior performance for medium to high volume environments.	*Performance*: • The physical separation of application servers containing business logic functions and database servers containing databases may moderately affect performance.

Table 2: Pros and Cons of 3 tier CS architecture

2.2.3 N-tier Architecture

As more users access the system a three-tier solution is more scalable than the other solutions because we can add as many middle tiers (running on each own server) as needed to ensure good performance (N-tier or multiple-tier). Security is also the best in the three-tier architecture because the middle layer protects the database tier. There is one major drawback to the N-tier architecture and that is that the additional tiers increase the complexity and cost of the installation.

In software engineering, multi-tier architecture (often referred to as n-tier architecture) is client-server architecture in which, the **presentation**, the **application processing** and the **data management** are logically separate processes. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of "multi-tier architecture" refers to three-tier architecture.

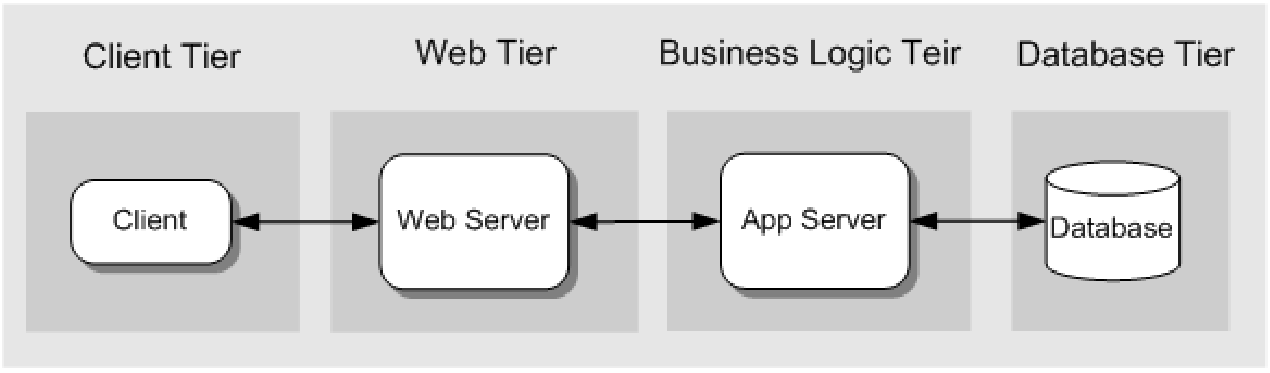


Figure 12: An example of N-tier Architecture

An example of N-tier architecture is web-based application. A web-based application might consist of the following tiers.

1. **Tier 1:** a client tier presentation implemented by a web browser.
2. **Tier 2:** a middle-tier distribution mechanism implemented by a web server.
3. **Tier 3:** a middle-tier service implemented by a set of server side scripts.
4. **Tier 4:** a data-tier storage mechanism implemented by a relational database.

2.3 Comparison between different Client Server Models

	1-tier	**2-tier**	**3-tier (N-tier)**
Benefits	Very simple Inexpensive No Server needed	Good Security More Scalable Faster execution	Exceptional security Faster execution Thin client Very scalable
Issues	Poor security Multi user issues	More costly More complex Think client	Very costly Very complex
Users	Usually 1(or few)	2-100	50-2000(+)

Figure 13: Summary of CS architecture

2.4 Client Server versus MVC

1. **Communication:** A fundamental rule in a three-tier architecture is the **client tier never communicates directly with the data tier**; in a three-tier model all communication must pass through the middle tier. Conceptually the three-tier architecture is linear.

   However, the MVC architecture is triangular: **the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.**

2. **History:** From a historical perspective the three-tier architecture concept emerged in the 1990s from observations of distributed systems (e.g., web applications) where the client, middleware and data tiers ran on physically separate platforms.

   Whereas MVC comes from the previous decade (by work at Xerox PARC in the late 1970s and early 1980s) and is based on observations of applications that ran on a single graphical workstation; MVC was applied to distributed applications later in its history.

Today, MVC and similar model-view-presenter (MVP) are Separation of Concerns design patterns that apply exclusively to the presentation layer of a larger system. In simple scenarios MVC may represent the primary design of a system, reaching directly into the database; however, in most scenarios the Controller and Model in MVC have a loose dependency on either a Service or Data layer/tier. This is all about Client-Server architecture in the 3-tier equivalent, communication between layers is bi-directional and always passes through the Middle tier in the MVC equivalent the communication is in unidirectional; we could say that each "layer" is updated by the one at the left and, in turn, updates the one at the right –where "left" and "right" are merely illustrative

3 Chapter 3: Elements of Dot net (.net) Languages

C# is part of .Net framework and is used for writing .Net applications.

3.1 The .Net Framework

The .Net framework is a revolutionary platform that helps us to write the following types of applications:

1. Windows applications
2. Web applications
3. Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL etc. All these languages can access the framework as well as communicate with each other.
The .Net framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .Net framework:

1. Common Language Runtime (CLR)
    - The .Net Framework Class Library
    - Common Language Specification (CLS)
    - Common Type System (CTS)
    - Metadata and Assemblies
    - Windows Forms
    - ASP.Net and ASP.Net AJAX
    - ADO.Net

2. Windows Workflow Foundation (WF)
    - Windows Presentation Foundation
    - Windows Communication Foundation (WCF)
    - LINQ

3.2 Integrated Development Environment (IDE) For C#

Microsoft provides the following development tools for C# programming:

3. Visual Studio 2010 (VS)
4. Visual C# 2010 Express (VCE)
5. Visual Web Developer

3.3 C# Program Structure

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure.
C# Hello World Example
A C# program basically consists of the following parts:

- Namespace declaration
- A class
- Class methods
- Class attributes

- A Main method
- Statements & Expressions
- Comments

```
using System;
namespace HelloWorldApplication
{
      class HelloWorld
      {
            static void Main(string[] args)
            {
                  /* my first program in C# */
                  Console.WriteLine("Hello World");
                  Console.ReadLine();
            }
      }
}
```

1. The first line of the program **using System;** - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.
2. The next line has the **namespace** declaration. A namespace is a collection of classes. TheHelloWorldApplication namespace contains the class HelloWorld.
3. The next line has a class declaration, the class **HelloWorld**, contains the data and method definitions that the program uses. Classes generally would contain more than one method. Methods define the behavior of the class.
4. The next line defines the **Main** method, which is the entry point for all C# programs. The Main method states what the class will do when executed.
5. The next line /*...*/ will be ignored by the compiler and it has been put to add additional **comments** in the program.
6. The Main method specifies its behavior with the statement Console.WriteLine("Hello World");
7. WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
8. The last line Console.ReadLine(); makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.
9. Its worth to note the following points:
   - C# is case sensitive.
   - All statements and expression must end with a semicolon (;).
   - The program execution starts at the Main method.
   - Unlike Java, file name could be different from the class name.

Compilation

Type **csc helloworld.cs** and press enter to compile the code.
If there are no errors in our code the command prompt will take us to the next line and would generate helloworld.exe executable file.

3.4 C# Basic Syntax

C# is an object oriented programming language. In Object Oriented Programming methodology a program consists of various objects that interact with each other by means of

actions. The actions that an object may take are called **methods**. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.

```
using System;
namespace RectangleApplication
{
      class Rectangle
      {
            // member variables
            double length;
            double width;
            public void Acceptdetails()
            {
                  length = 4.5;
                  width = 3.5;
            }
            public double GetArea()
            {
                  return length * width;
            }
            public void Display()
            {
                  Console.WriteLine("Length: {0}", length);
                  Console.WriteLine("Width: {0}", width);
                  Console.WriteLine("Area: {0}", GetArea());
            }
      }
      class ExecuteRectangle
      {
            static void Main(string[] args)
            {
                  Rectangle r = new Rectangle();
                  r.Acceptdetails();
                  r.Display();
                  Console.ReadLine();
            }
      }
}
```

The **class** Keyword
The class keyword is used for declaring a class.

Member Variables
Variables are attributes or data members of a class, used for storing data. In the preceding program, the Rectangle class has two member variables named length and width.

Member Functions
Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: AcceptDetails, GetArea and Display.

Instantiating a Class

In the preceding program, the class ExecuteRectangle is used as a class which contains the Main() method and instantiates the Rectangle class.

Identifiers
An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

1. A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
2. It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However an underscore (_) can be used.
3. It should not be a C# keyword.

3.5 C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers, however, if we want to use these keywords as identifiers, we may prefix the keyword with the @ character. In C# some identifiers have special meaning in context of code, such as get and set, these are called contextual keywords.

The following table lists the reserved keywords in C#:

abstract	*As*	*Base*	*bool*	*break*	*byte*	*case*
catch	*Char*	*checked*	*class*	*const*	*continue*	*decimal*
default	*delegate*	*do*	*double*	*else*	*enum*	*event*
explicit	*extern*	*false*	*finally*	*fixed*	*float*	*for*
foreach	*Goto*	*if*	*implicit*	*in*	*in*	*int*
interface	*internal*	*is*	*lock*	*long*	*namespae*	*new*
null	*object*	*operator*	*out*	*out*	*override*	*params*
private	*protected*	*public*	*readonly*	*ref*	*return*	*sbyte*
sealed	*short*	*sizeof*	*stackalloc*	*static*	*string*	*struct*
switch	*This*	*throw*	*true*	*try*	*typeof*	*uint*
ulong	*unchecked*	*unsafe*	*ushort*	*using*	*virtual*	*void*
volatile			*While*			

Table 3: C# Keywords

3.6 C# Data types

In C#, variables are categorized into the following types:

1. Value types
2. Reference types
3. Pointer types

3.6.1 Value Types

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, float**, which stores numbers, alphabets and floating point numbers respectively. When we declare an int type, the system allocates memory to store the value.

The following table lists the available value types in C#.

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

Table 4: C# Value data types

To get the exact size of a type or a variable on a particular platform, we can use the **sizeof** method. The expression **sizeof(type)** yields the storage size of the object or type in bytes. Following is an example to get the size of int type on any machine:

```
namespace DataTypeApplication
{
      class Program
      {
            static void Main(string[] args)
            {
                  Console.WriteLine("Size of int: {0}", sizeof(int));
                  Console.ReadLine();
            }
      }
}
```

3.6.2 Reference Types

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables. In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Examples of built in reference types are: **object, dynamic and string.**

3.6.2.1 Object Type

The Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for **System.Object** class. So object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called boxing and on the other hand, when an object type is converted to a value type it is called unboxing.

```
object obj;
obj = 100; // this is boxing
```

3.6.2.2 Dynamic Type

We can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at runtime.
Syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

Dynamic types are similar to object types except that, type checking for object type variables takes place at compile time, whereas those for the dynamic type variables take place at run time.

3.6.2.3 String Type

The String Type allows us to assign any string values to a variable. The string type is an alias for the **System.String** class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.
For example,

```
String str = "First String";
```

A @quoted string literal looks like:

```
@"First String";
```

The user defined reference types are: class, interface, or delegate.

3.6.3 Pointer Types

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as in C or C++.
Syntax for declaring a pointer type is:

```
type* identifier;
For example,
char* cptr;
int* iptr;
```

3.7 C# Type Conversition

Type conversion is basically type casting, or converting one type of data to another type. In C#, type casting has two forms:

1. **Implicit type conversion** - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types, and conversions from derived classes to base classes.
2. **Explicit type conversion** - users using the pre-defined functions do these conversions explicitly. Explicit conversions require a cast operator.

The following example shows an explicit type conversion:

```
namespace TypeConversionApplication
{
      class ExplicitConversion
      {
            static void Main(string[] args)
            {
                double d = 5673.74;
                int i;
                // cast double to int.
                i = (int)d;
                Console.WriteLine(i);
                Console.ReadLine();
            }
        }
}
```

Output:
5673

3.7.1.1 C# Type Conversion Methods

C# provides the following built-in type conversion methods:

1. **ToSingle**: Converts a type to a small floating-point number.
2. **ToString**: Converts a type to a string.
3. **ToType**: Converts a type to a specified type.
4. **ToUInt16**: Converts a type to an unsigned int type.
5. **ToUInt32**: Converts a type to an unsigned long type.
6. **ToUInt64**: Converts a type to an unsigned big integer.
7. **ToBoolean**: Converts a type to a Boolean value, where possible.
8. **ToByte** : Converts a type to a byte.
9. **ToChar**: Converts a type to a single Unicode character, where possible.
10. **ToDateTime**: Converts a type (integer or string type) to date-time structures.
11. **ToDecimal**: Converts a floating point or integer type to a decimal type.
12. **ToDouble**: Converts a type to a double type.
13. **ToInt16**: Converts a type to a 16-bit integer.
14. **ToInt32**: Converts a type to a 32-bit integer.
15. **ToInt64**: Converts a type to a 64-bit integer.
16. **ToSbyte**: Converts a type to a signed byte type.

Table 5: C# Type Conversion Methods

The following example converts various value types to string type:

```
namespace TypeConversionApplication
{
class StringConversion
{
static void Main(string[] args)
{
int i = 75;
float f = 53.005f;
double d = 2345.7652;
bool b = true;
Console.WriteLine(i.ToString());
Console.WriteLine(f.ToString());
Console.WriteLine(d.ToString());
Console.WriteLine(b.ToString());
Console.ReadLine();
}
}
}
```

Output:

```
75
53.005
2345.7652
True
```

3.8 C# Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of Values that can be stored within that memory; and the set of operations that can be applied to the variable.
We have already discussed various data types. The basic value types provided in C# can be categorized as:
C# also allows defining other value types of variable like **enum** and reference types of variables like class.

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

3.8.1.1 Variable Declaration in C#

Syntax for variable declaration in C# is:

```
<data_type> <variable_list>;
```

Here, data_type must be a valid C# data type including char, int, float, double, or any user defined data type etc., and variable_list may consist of one or more identifier names separated by commas.

Some valid variable declarations along with their definition are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

3.8.1.2 Variable Initialization in C#

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

variable_name = value;

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:

<data_type> <variable_name> = value;

Some examples are:

```
int d = 3, f = 5; /* initializing d and f. */
byte z = 22; /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x'; /* the variable x has the value 'x'. */
```

Following example which makes use of various types of variables:

```
namespace VariableDeclaration
{
      class Program
      {
            static void Main(string[] args)
            {
                  short a;
                  int b ;
                  double c;
                  /* actual initialization */
                  a = 10;
                  b = 20;
                  c = a + b;
                  Console.WriteLine("a = {0}, b = {1}, c = {2}", a,
                  b, c);
                  Console.ReadLine();
            }
      }
}
```

Output:

```
a = 10, b = 20, c = 30
```

3.8.1.3 Accepting Values from User

The Console class in the System namespace provides a function ReadLine() for accepting input from the user and store it into a variable. int num;

```
num = Convert.ToInt32(Console.ReadLIne());
```

The function *Convert.ToInt32()* converts the data entered by the user to int data type, because Console.ReadLine() accepts the data in string format.

Lvalues and Rvalues in C#:

There are two kinds of expressions in C#:

1. lvalue : An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. rvalue : An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side. Following is a valid statement:
int g = 20;
But following is not a valid statement and would generate compile-time error:
10 = 20;

3.9 C# Constants and Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.
The constants are treated just like regular variables except that their values cannot be modified after their definition.

3.9.1.1 Integer Literals
An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for an integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.
Here are some examples of integer literals:
```
212 /* Legal */
215u /* Legal */
0xFeeL /* Legal */
078 /* Illegal: 8 is not an octal digit */
032UU /* Illegal: cannot repeat a suffix */
Following are other examples of various types of Integer
literals:
85 /* decimal */
0213 /* octal */
0x4b /* hexadecimal */
30 /* int */
30u /* unsigned int */
30l /* long */
30ul /* unsigned long */
```

3.9.1.2 Floating-point Literals
A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. We can represent floating point literals either in decimal form or exponential form.
Here are some examples of floating-point literals:
```
3.14159 /* Legal */
314159E-5L /* Legal */
510E /* Illegal: incomplete exponent */
210f /* Illegal: no decimal or exponent */
.e55 /* Illegal: missing integer or fraction */
```

While representing using decimal form, we must include the decimal point, the exponent, or both and while representing using exponential form we must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

3.9.1.3 Character Constants

Character literals are enclosed in single quotes e.g., 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C# when they are proceeded by a back slash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here you have a list of some of such escape sequence codes:

Escape sequence	**Meaning**
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

3.9.1.4 String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

We can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

3.10 Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is:

const <data_type> <constant_name> = value;

The following program demonstrates defining and using a constant in your program:

```
using System;
namespace DeclaringConstants
{
      class Program
```

```
        {
                static void Main(string[] args)
                {
                        const double pi = 3.14159; // constant declaration
                        double r;
                        Console.WriteLine("Enter Radius: ");
                        r = Convert.ToDouble(Console.ReadLine());
                        double areaCircle = pi * r * r;
                        Console.WriteLine("Radius:  {0},  Area:  {1}",  r,
                        areaCircle);
                        Console.ReadLine();
                }
        }
}
```

Output:

```
Enter Radius
3
Radius: 3, Area: 28.27431
```

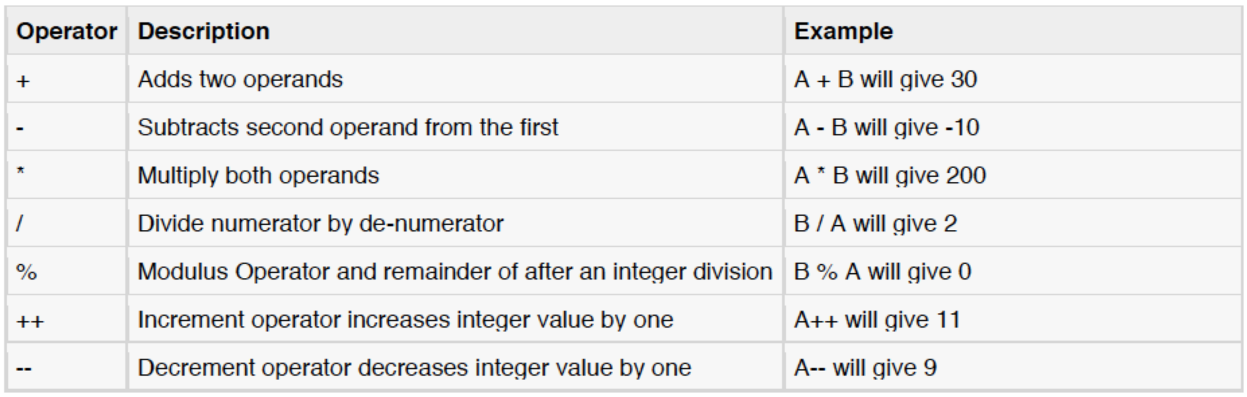
3.11 C# Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Misc Operators

3.11.1 Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable A holds 10 and variable B holds 20 then:



Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Table 6: C# Arithmetic Operators

3.11.2 Relational Operators

Following table shows all the relational operators supported by C#. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Table 7: C# Relational Operators

3.11.3 Logical Operators

Following table shows all the logical operators supported by C#. Assume variable A holds Boolean value true and variable B holds Boolean value false then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Table 8: C# Logical Operators

3.11.4 Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P	Q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Table 9: C# Bitwise Operators

Assume if A = 60; and B = 13; Now in binary format they will be as follows:
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

```
using System;
namespace OperatorsAppl
{
      class Program
      {
            static void Main(string[] args)
            {
                  int a = 60; /* 60 = 0011 1100 */
                  int b = 13; /* 13 = 0000 1101 */
                  int c = 0;
                  c = a & b; /* 12 = 0000 1100 */
                  Console.WriteLine("Line 1 - Value of c is {0}", c
                  );
                  c = a | b; /* 61 = 0011 1101 */
                  Console.WriteLine("Line 2 - Value of c is {0}", c);
                  c = a ^ b; /* 49 = 0011 0001 */
                  Console.WriteLine("Line 3 - Value of c is {0}", c);
                  c = ~a; /*-61 = 1100 0011 */
                  Console.WriteLine("Line 4 - Value of c is {0}", c);
                  c = a << 2; /* 240 = 1111 0000 */
                  Console.WriteLine("Line 5 - Value of c is {0}", c);
                  c = a >> 2; /* 15 = 0000 1111 */
                  Console.WriteLine("Line 6 - Value of c is {0}", c);
                  Console.ReadLine();
            }
      }
}
```

Output:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

3.11.5 Assignment Operators

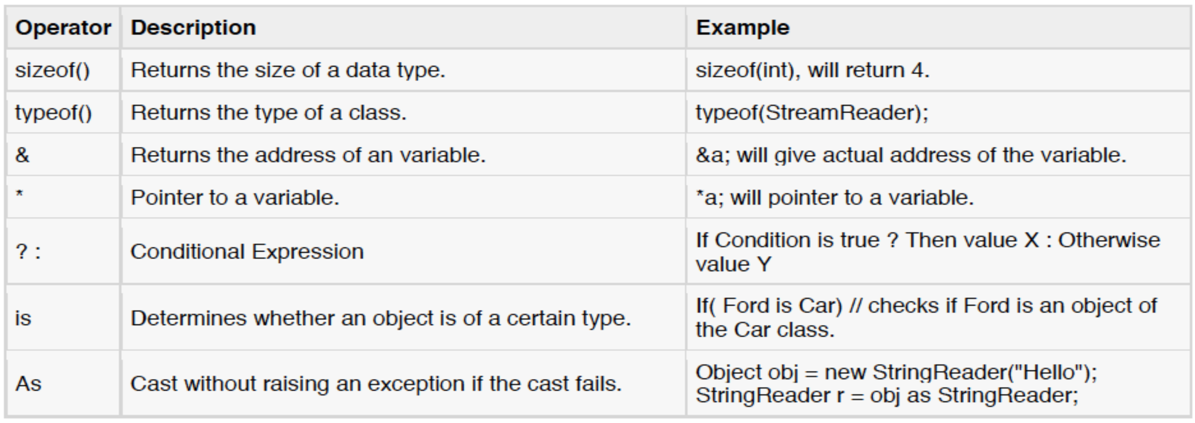
Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes	C %= A is

	modulus using two operands and assign the result to left operand	equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Table 10: C# Assignment Operators

3.11.6 Misc Operators

There are few other important operators including sizeof, typeof and ? : supported by C#.



Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
As	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Table 11: C# Misc Operators

3.12 C# Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. C# provides following types of decision making statements.

3.12.1 If statement

An if statement consists of a boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement in C# is:

```
if(boolean_expression)
{
/* statement(s) will execute if the boolean expression is true */
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If boolean expression evaluates to false then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

3.12.2 If…else statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

Syntax:

The syntax of an if...else statement in C# is:

```
if(boolean_expression)
{
/* statement(s) will execute if the boolean expression is true */
}
else
{
/* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to true then the if block of code will be executed otherwise else block of code will be executed.

3.12.3 The if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

1. An if can have zero or one else's and it must come after any else if's.
2. An if can have zero to many else if's and they must come before the else.
3. Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else if...else statement in C# is:

```
if(boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
/* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
/* Executes when the boolean expression 3 is true */
}
else
{
/* executes when the none of the above condition is true */
}
```

3.12.4 nested if statements

It is always legal in C# to nest if-else statements, which means we can use one if or else if statement inside another if or else if statement(s).

Syntax:

The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
if(boolean_expression 2)
{
/* Executes when the boolean expression 2 is true */
}
}
```

3.12.5 Switch statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax:

The syntax for a switch statement in C# is as follows:

```
switch(expression){
     case constant-expression :
          statement(s);
          break; /* optional */
     case constant-expression :
          statement(s);
          break; /* optional */
     /* We can have any number of case statements */
     default : /* Optional */
          statement(s);
}
```

The following rules apply to a switch statement:

1. The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
2. We can have any number of case statements within a switch. The value to be compared to and a colon follow each case.
3. The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
4. When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

5. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
6. Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
7. A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

3.12.6 The ? : Operator:

It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions.The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

3.13 C# Loops

There may be a situation when we need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:

3.13.1 While loop

A while loop statement in C# repeatedly executes a target statement as long as a given condition is true.

Syntax:

The syntax of a while loop in C# is:

```
while(condition)
{
      statement(s);
}
```

Here statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

3.13.2 for loop

A for loop is a repetition control structure that allows us to efficiently write a loop that needs to execute a specific number of times.

Syntax:

The syntax of a for loop in C# is:

```
for ( init; condition; increment )
{
      statement(s);
}
```

Here is the flow of control in a for loop:

1. The init step is executed first, and only once. This step allows us to declare and initialize any loop control variables. We are not required to put a statement here, as long as a semicolon appears.

2. Next the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows us to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

3.13.3 do…while loop

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop. A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop in C# is:

```
do
{
      statement(s);
}while( condition );
```

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

3.13.4 nested loops

C# allows to use one loop inside another loop.

Syntax:

The syntax for a nested for loop statement in C# is as follows:

```
for ( init; condition; increment )
{
      for ( init; condition; increment )
      {
            statement(s);
      }
      statement(s);
}
```

Example: Program to find the prime numbers less than 100.

```
using System;
namespace Loops
{
      class Program
      {
            static void Main(string[] args)
            {
                  /* local variable definition */
                  int i, j;
                  for (i = 2; i < 100; i++)
                  {
                        for (j = 2; j <= (i / j); j++)
                              if ((i % j) == 0) break; // if factor
                  found, not prime
                        if (j > (i / j))
                        Console.WriteLine("{0} is prime", i);
                  }
                  Console.ReadLine();
            }
      }
}

// Demonstrate the for loop.
using System;
class ForDemo {
      static void Main() {
            int count;
            for(count 0; count < 5; count count+1)
                  Console.WriteLine("This is count: " + count);
            Console.WriteLine("Done!");
}
}
// Compute the sum and product of the numbers from 1 to 10.
using System;
class ProdSum {
      static void Main() {
      int prod;
      int sum;
      int i;
      sum=0;
      prod=1;
      for(i 1; i < 10; i++) {
            sum sum + i;
            prod=prod * i;
      }
      Console.WriteLine("Sum is " + sum);
      Console.WriteLine("Product is " + prod);
      }
}
// Demonstrate an @ identifier. Identifier as a keyword
using System;
class IdTest {
      static void Main() {
      int @if; // use if as an identifier
      for(@if=0; @if < 10; @if++)
            Console.WriteLine("@if is " + @if);
      }
}
//Demonstrate Math.Sin(), Math.Cos(), and Math.Tan().
using System;
      class Trigonometry {
      static void Main() {
            Double theta; // angle in radians
            for(theta 0.1; theta < 1.0; theta theta + 0.1) {
                  Console.WriteLine("Sine of " + theta + " is " +
                        Math.Sin(theta));
                  Console.WriteLine("Cosine of " + theta + " is " +
                        Math.Cos(theta));
                  Console.WriteLine("Tangent of " + theta + " is " +
                        Math.Tan(theta));
            Console.WriteLine();
            }
      }
}
//Use format commands.
using System;
class DisplayOptions {
      static void Main() {
            int i;
            Console.WriteLine("Value\tSquared\tCubed");
            for(i 1; i < 10; i++)
                  Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
      }
}
```

Output:

```
Value Squared Cubed
1      1         1
2      4         8
3      9         27
4      16        64
5      25        125
6      36        216
7      49        343
8      64        512
9      81        729
```

3.14 Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. C# provides the following control statements.

3.14.1 break statement

The break statement in C# has following two usage:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement.

If we are using nested loops (ie. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a break statement in C# is as follows:

```
break;
```

3.14.2 continue statement

The continue statement in C# works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between. For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control passes to the conditional tests.

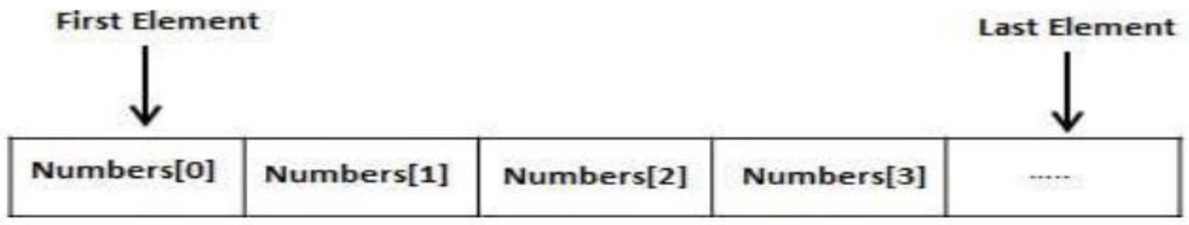
Syntax:

The syntax for a continue statement in C# is as follows:

```
continue;
```

3.15 C# Array

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



3.15.1 Declaring Arrays

To declare an array in C#, you can use the following syntax:

```
datatype[] arrayName;
```

where,

1. datatype is used to specify the type of elements to be stored in the array.
2. [ ] specifies the rank of the array. The rank specifies the size of the array.
3. arrayName specifies the name of the array.

For example,

```
double[] balance;
```

3.15.2 Initializing and assigning values to an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, we can assign values to the array. Array is a reference type, so we need to use the new keyword to create an instance of the array.

```
double[] balance = new double[10];
double[] balance = new double[10];
balance[0] = 4500.0;
```

1. We can assign values to the array at the time of declaration, like:
   ```
   double[] balance = { 2340.0, 4523.69, 3421.0};
   ```
2. We can also create and initialize an array, like:
   ```
   int [] marks = new int[5] { 99, 98, 92, 97, 95};
   ```
3. In the preceding case, we may also omit the size of the array, like:
   ```
   int [] marks = new int[] { 99, 98, 92, 97, 95};
   ```
4. We can also copy an array variable into another target array variable. In that case, both the target and source would point to the same memory location:
   ```
   int [] marks = new int[] { 99, 98, 92, 97, 95};
   int[] score = marks;
   ```

3.15.3 Accessing Array Elements

Indexing the array name accesses an element. Placing the index of the element within square brackets after the name of the array does this. For example:

```
double salary = balance[9];
```

3.15.4 Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. A two dimensional array can be thought of as a table which will have x number of rows and y number of columns.

3.15.4.1 Initializing Two-Dimensional Arrays

Specifying bracketed values for each row may initialize multidimensional arrays. Following is an array with 3 rows and each row has 4 columns.

```
int [,] a = new int [3,4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

3.15.4.2 Accessing Two-Dimensional Array Elements

Using the subscripts ie accesses an element in 2-dimensional array. row index and column index of the array. For example:

```
int val = a[2,3];
```

The above statement will take 4th element from the 3rd row of the array.

3.16 C# Classes

When we define a class, we define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

3.16.1 Class Definition

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. Following is the general form of a class definition:

```
<access specifier> class class_name
{
      // member variables
      <access specifier> <data type> variable1;
      <access specifier> <data type> variable2;
      ...
      <access specifier> <data type> variableN;
      // member methods
      <access specifier> <return type> method1(parameter_list)
      {
            // method body
      }
      <access specifier> <return type> method2(parameter_list)
      {
            // method body
      }
      ...
      <access specifier> <return type> methodN(parameter_list)
      {
            // method body
      }
}
```

Please note that,

1. Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is **internal**. Default access for the members is **private**. Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.
2. To access the class members, we will use the dot (.) operator.
3. The dot operator links the name of an object with the name of a member.

3.16.2 Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

The following example program can demonstrate this concept.

```
using System;
namespace BoxApplication
{
      class Box
      {
            private double length; // Length of a box
            private double breadth; // Breadth of a box
            private double height; // Height of a box
            public void setLength( double len )
            {
                  length = len;
            }
            public void setBreadth( double bre )
            {
                  breadth = bre;
            }
            public void setHeight( double hei )
            {
                  height = hei;
            }
            public double getVolume()
            {
                  return length * breadth * height;
            }
      }
      class Boxtester
      {
            static void Main(string[] args)
            {
                  Box Box1 = new Box(); // Declare Box1 of type Box
                  Box Box2 = new Box();// Declare Box2 of type Box
                  double volume;
                  // box 1 specification
                  Box1.setLength(6.0);
                  Box1.setBreadth(7.0);
                  Box1.setHeight(5.0);
                  // box 2 specification
                  Box2.setLength(12.0);
                  Box2.setBreadth(13.0);
                  Box2.setHeight(10.0);
                  // volume of box 1
                  volume = Box1.getVolume();
                  Console.WriteLine("Volume of Box1 : {0}" ,volume);
                  // volume of box 2
                  volume = Box2.getVolume();
                  Console.WriteLine("Volume of Box2 : {0}", volume);
                  Console.ReadLine();
            }
      }
}
```

Output:

```
Volume of Box1: 210
Volume of Box2: 1560
```

3.17 C# Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces. Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

3.17.1 Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace

1. **ArrayList** : It represents ordered collection of an object that can be indexed individually.
   It is basically an alternative to an array. However unlike array we can add and remove items from a list at a specified position using an index and the array resizes itself automatically. It also allows dynamic memory allocation, add, search and sort items in the list.
2. **Hashtable**: It uses a key to access the elements in the collection.
   A hash table is used when we need to access elements by using key, and we can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.
3. **Sortedlist**: It uses a key as well as an index to access the items in a list.
   A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If we access items using an index, it is an ArrayList, and if we access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.
4. **Stack**: It represents a last-in, first out collection of object.
   It is used when we need a last-in, first-out access of items. When we add an item in the list, it is called pushing the item and when you remove it, it is calledpopping the item.
5. **Queue**: It represents a first-in, first out collection of object.
   It is used when we need a first-in, first-out access of items. When we add an item in the list, it is called enqueue and when we remove an item, it is called deque.
6. **Bitarray**: It represents an array of the binary representation using the values 1 and 0.
   It is used when we need to store the bits but do not know the number of bits in advance. We can access items from the BitArray collection by using an integer index, which starts from zero.

3.17.1.1 Arraylist

```
using System;
using System.Collections;
namespace CollectionApplication
{
      class Program
      {
            static void Main(string[] args)
            {
                  ArrayList al = new ArrayList();
                  Console.WriteLine("Adding some numbers:");
                  al.Add(45);
                  al.Add(78);
                  al.Add(33);
                  al.Add(56);
                  al.Add(12);
                  al.Add(23);
                  al.Add(9);
                  Console.WriteLine("Capacity: {0} ", al.Capacity);
                  Console.WriteLine("Count: {0}", al.Count);
                  Console.Write("Content: ");
                  foreach (int i in al)
                  {
                        Console.Write(i + " ");
                  }
                  Console.WriteLine();
                  Console.Write("Sorted Content: ");
                  al.Sort();
                  foreach (int i in al)
                  {
                       Console.Write(i + " ");
                  }
                  Console.WriteLine();
                  Console.ReadKey();
             } } }
```

Output:

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Sorted Content: 9 12 23 33 45 56 78
```

3.17.1.2 Hash Table

```
using System;
using System.Collections;

namespace CollectionsApplication
{
class Program
{
static void Main(string[] args)
{
      Hashtable ht = new Hashtable();
      ht.Add("001", "Zara Ali");
      ht.Add("002", "Abida Rehman");
      ht.Add("003", "Joe Holzner");
      ht.Add("004", "Mausam Benazir Nur");
      ht.Add("005", "M. Amlan");
      ht.Add("006", "M. Arif");
      ht.Add("007", "Ritesh Saikia");
      if (ht.ContainsValue("Nuha Ali"))
      {
           Console.WriteLine("This student name is already in the
      list");
      }
      else
      {
           ht.Add("008", "Nuha Ali");
      }
      // Get a collection of the keys.
      ICollection key = ht.Keys;
      foreach (string k in key)
      {
           Console.WriteLine(k + ": " + ht[k]);
      }
      Console.ReadKey();
}
}
}
```

Output
```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
004: Mausam Benazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

3.17.1.3 Stack
```
using System;
using System.Collections;
namespace CollectionsApplication
{
      class Program
      {
            static void Main(string[] args)
            {
                  Stack st = new Stack();
                  st.Push('A');
                  st.Push('M');
                  st.Push('G');
                  st.Push('W');
                  Console.WriteLine("Current stack: ");
                  foreach (char c in st)
                  {
                        Console.Write(c + " ");
                  }
                  Console.WriteLine();
                  st.Push('V');
                  st.Push('H');
                  Console.WriteLine("The next poppable value in
                  stack: {0}",st.Peek());
                  Console.WriteLine("Current stack: ");
                  foreach (char c in st)
                  {
                  Console.Write(c + " ");
                  }
                  Console.WriteLine();
                  Console.WriteLine("Removing values ");
                  st.Pop();
                  st.Pop();
                  st.Pop();
                  Console.WriteLine("Current stack: ");
                  foreach (char c in st)
                  {
                  Console.Write(c + " ");
                  }
            }
      }
}
```

Output:
```
Current stack:
```

```
W G M A
The next poppable value in stack: H
Current stack:
H V W G M A
Removing values
Current stack:
G M A
```

3.17.1.4 Queue

```
using System;
using System.Collections;
namespace CollectionsApplication
{
      class Program
      {
            static void Main(string[] args)
            {
                  Queue q = new Queue();
                  q.Enqueue('A');
                  q.Enqueue('M');
                  q.Enqueue('G');
                  q.Enqueue('W');
                  Console.WriteLine("Current queue: ");
                  foreach (char c in q)
                  Console.Write(c + " ");
                  Console.WriteLine();
                  q.Enqueue('V');
                  q.Enqueue('H');
                  Console.WriteLine("Current queue: ");
                  foreach (char c in q)
                  Console.Write(c + " ");
                  Console.WriteLine();
                  Console.WriteLine("Removing some values ");
                  char ch = (char)q.Dequeue();
                  Console.WriteLine("The removed value: {0}", ch);
                  ch = (char)q.Dequeue();
                  Console.WriteLine("The removed value: {0}", ch);
                  Console.WriteLine ("Current queue: ");
                        foreach (char c in q)
                              Console.Write (c + " ");
                  Console.ReadKey();
            }
      }
}
```

Output

```
Current queue:
A M G W
Current queue:
A M G W V H
Removing values
The removed value: A
The removed value: M
Current queue:
G W V H
```

3.17.2 C# Properties

Properties are named members of classes, structures, and interfaces. They use **accessors** through which the values of the private fields can be read, written or manipulated.

For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

Accessors

The **accessor** of a property contains the executable statements that help in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both.

Example.

```
using System;
class Student
{
      private string code = "N.A";
      private string name = "not known";
      private int age = 0;
      // Declare a Code property of type string:
      public string Code
      {
            get
            {
                  return code;
            }
            set
            {
                  code = value;
            }
      }
      // Declare a Name property of type string:
      public string Name
      {
            get
            {
                  return name;
            }
            set
            {
                  name = value;
            }
      }
      // Declare a Age property of type int:
      public int Age
      {
            get
            {
                  return age;
            }
            set
            {
                  age = value;
            }
      }
      public override string ToString()
      {
           return "Code = " + Code +", Name = " + Name + ", Age = "
      + Age;
      }
      public static void Main()
      {
           // Create a new Student object:
           Student s = new Student();
           // Setting code, name and the age of the student
           s.Code = "001";
           s.Name = "Zara";
           s.Age = 9;
           Console.WriteLine("Student Info: {0}", s);
           //let us increase age
           s.Age += 1;
           Console.WriteLine("Student Info: {0}", s);
           Console.ReadKey();
      }
}
```

Output

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

3.18 C# Delegates

C # delegates are similar to pointers to functions, in C or C++. A delegate is a reference type variable that holds the **reference to a method**. The reference can be changed at runtime. Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

3.18.1 Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which have the same signature as that of the delegate. For example, consider a delegate:

public delegate int MyDelegate (string s);

The preceding delegate can be used to reference any method that has a single string parameter and returns an int type variable.

```
public delegate double ProcessResults( double x, double y );

public class ProcessResults : System.MulticastDelegate
{
      public double Invoke( double x, double y );
      // Other stuff omitted for clarity
}
```

3.18.2 Syntax for delegate declaration is:

delegate <return type> <delegate-name> <parameter list>

3.18.3 Instantiating Delegates

Once a delegate type has been declared, a delegate object must be created with the new keyword and be associated with a particular method. When creating a delegate, the argument

passed to the new expression is written like a method call, but without the arguments to the method. Under the .NET platform, the delegate type is the preferred means of defining and responding to callbacks within applications. Essentially, the .NET delegate type is a type-safe object that "points to" a method or a list of methods that can be invoked at a later time. Unlike a traditional C++ function pointer, however, .NET delegates are classes that have built-in support for multicasting and asynchronous
method invocation. For example:

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

Following example demonstrates declaration, instantiation and use of a delegate that can be used to reference methods that take an integer parameter and returns an integer value.

```
using System;
delegate int NumberChanger(int n);
namespace DelegateAppl
{
      class TestDelegate
      {
            static int num = 10;
            public static int AddNum(int p)
            {
                  num += p;
                  return num;
            }
            public static int MultNum(int q)
            {
                  num *= q;
                  return num;
            }
            public static int getNum()
            {
                  return num;
            }
            static void Main(string[] args)
            {
                  //create delegate instances
                  NumberChanger nc1 = new NumberChanger(AddNum);
                  NumberChanger nc2 = new NumberChanger(MultNum);
                  //calling the methods using the delegate objects
                  nc1(25);
                  Console.WriteLine("Value of Num: {0}", getNum());
                  nc2(5);
                  Console.WriteLine("Value of Num: {0}", getNum());
                  Console.ReadKey();
            }
      }
}
```

Output:

```
Value of Num: 35
Value of Num: 175
```

3.18.4 Use of Delegate

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference methods that take a string as input and return nothing. We use this delegate to call two methods, the first prints the string to the console, and the second one prints it to a file:

```
using System;
using System.IO;
namespace DelegateAppl
{
      class PrintString
      {
            static FileStream fs;
            static StreamWriter sw;

            // delegate declaration
            public delegate void printString(string s);

            // this method prints to the console
            public static void WriteToScreen(string str)
            {
                  Console.WriteLine("The String is: {0}", str);
            }

            //this method prints to a file
            public static void WriteToFile(string s)
            {
                  fs = new FileStream("c:\\message.txt",
                  FileMode.Append, FileAccess.Write);
                  sw = new StreamWriter(fs);
                  sw.WriteLine(s);
                  sw.Flush();
                  sw.Close();
                  fs.Close();
            }
      // this method takes the delegate as parameter and uses it to
            // call the methods as required
            public static void sendString(printString ps)
            {
                  ps("Hello World");
            }
            static void Main(string[] args)
            {
                  printString ps1 = new printString(WriteToScreen);
                  printString ps2 = new printString(WriteToFile);
                  sendString(ps1);
                  sendString(ps2);
                  Console.ReadKey();
            }
      }
}
```

Output: `The String is: Hello World`

3.19 C# Events

Events are basically a user action like key press, clicks, mouse movements etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

3.19.1 Using Delegates with Events

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to **publish** the event. This is called the **publisher class**. Some other class that accepts this event is called the **subscriber class**. Events use the **publisher-subscriber** model.

A publisher is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A subscriber is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

3.19.2 Declaring Events

To declare an event inside a class, first a delegate type for the event must be declared. For example,

```
public delegate void BoilerLogHandler(string status);
```

Next, the event itself is declared, using the event keyword: Defining event based on the above delegate

```
public event BoilerLogHandler BoilerEventLog;
```

The preceding code defines a delegate named **BoilerLogHandler** and an event name **BoilerEventLog**, which invokes the delegate when it is raised.

```
using System;
namespace SimpleEvent
{
      using System;
      public class EventTest
      {
            private int value;
            public delegate void NumManipulationHandler();
            public event NumManipulationHandler ChangeNum;
            protected virtual void OnNumChanged()
            {
                  if (ChangeNum != null)
                  {
                        ChangeNum();
                  }
                  else
                  {
                        Console.WriteLine("Event fired!");
                  }
            }
            public EventTest(int n )
            {
                  SetValue(n);
            }
            public void SetValue(int n)
            {
                  if (value != n)
                  {
                        value = n;
```

```
                    OnNumChanged();
                }
            }
        }
        public class MainClass
        {
                public static void Main()
                {
                    EventTest e = new EventTest(5);
                    e.SetValue(7);
                    e.SetValue(11);
                    Console.ReadKey();
                }
        }
}
```

```
Output:
Event Fired!
Event Fired!
Event Fired!
```

3.20 C# Indexer

An indexer allows an object to be indexed like an array. When we define an indexer for a class, this class behaves like a virtual array. We can then access the instance of this class using the array access operator ([ ]).

Syntax

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
    // The get accessor.
get
{
    // return the value specified by index
}
```

3.20.1 Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. Like properties, We use **get** and **set** assessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the this keyword, which refers to the object instance. The following example demonstrates the concept:

```
using System;
namespace IndexerApplication
{
class IndexedNames
    {
        private string[] namelist = new string[size];
        static public int size = 10;
```

```
          public IndexedNames()
          {
                for (int i = 0; i < size; i++)
                namelist[i] = "N. A.";
          }
          public string this[int index]
          {
                get
                {
                string tmp;
                if( index >= 0 && index <= size-1 )
                {
                tmp = namelist[index];
                }
                else
                {
                tmp = "";
                }
                return ( tmp );
          }
          set
          {
                if( index >= 0 && index <= size-1 )
                {
                namelist[index] = value;
                }
          }
    }
          static void Main(string[] args)
          {
                IndexedNames names = new IndexedNames();
                names[0] = "Zara";
                names[1] = "Riz";
                names[2] = "Nuha";
                names[3] = "Asif";
                names[4] = "Davinder";
                names[5] = "Sunil";
                names[6] = "Rubic";
                for ( int i = 0; i < IndexedNames.size; i++ )
                {
                      Console.WriteLine(names[i]);
                }
                Console.ReadKey();
          }
    }
}
```

Output:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
```

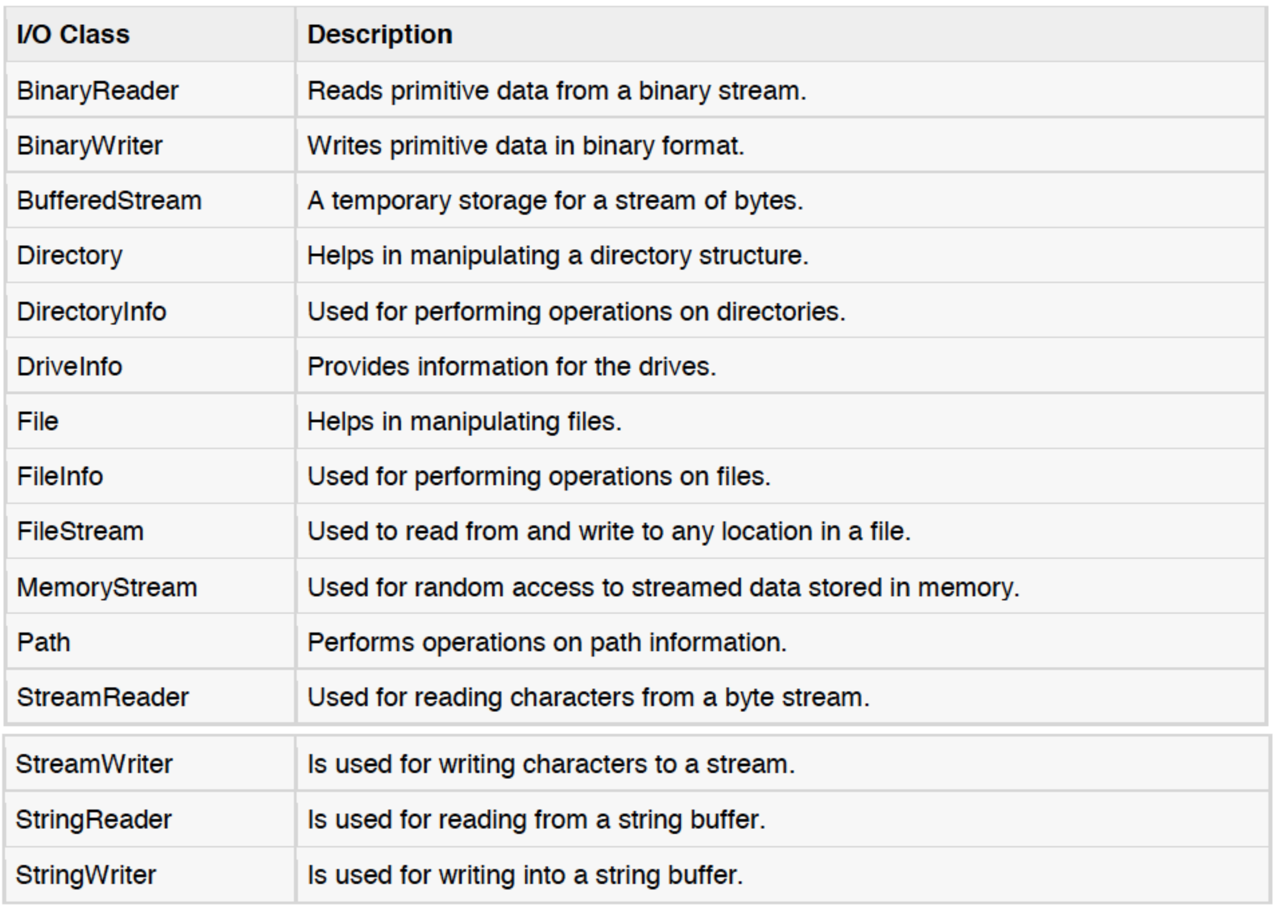
```
N. A.
N. A.
```

3.21 C# File I/O

A file is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**. The stream is basically the **sequence of bytes passing through the communication path**. There are two main streams: the **input stream** and the **output stream**. The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation). The **System.IO** namespace is the region of the base class libraries devoted to file-based (and memory-based) input and output (I/O) services. Like any namespace, System.IO defines a set of classes, interfaces, enumerations, structures, and delegates. Many of the types within the System.IO namespace focus on the programmatic manipulation of physical directories and files. However, additional types provide support to read data from and write data to string buffers, as well as raw memory locations.

3.21.1 C# I/O Classes

The System.IO namespace has various class that are used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.



I/O Class	Description
BinaryReader	Reads primitive data from a binary stream.
BinaryWriter	Writes primitive data in binary format.
BufferedStream	A temporary storage for a stream of bytes.
Directory	Helps in manipulating a directory structure.
DirectoryInfo	Used for performing operations on directories.
DriveInfo	Provides information for the drives.
File	Helps in manipulating files.
FileInfo	Used for performing operations on files.
FileStream	Used to read from and write to any location in a file.
MemoryStream	Used for random access to streamed data stored in memory.
Path	Performs operations on path information.
StreamReader	Used for reading characters from a byte stream.
StreamWriter	Is used for writing characters to a stream.
StringReader	Is used for reading from a string buffer.
StringWriter	Is used for writing into a string buffer.

Table 12: C# IO Classes

3.21.1.1 The FileStream Class

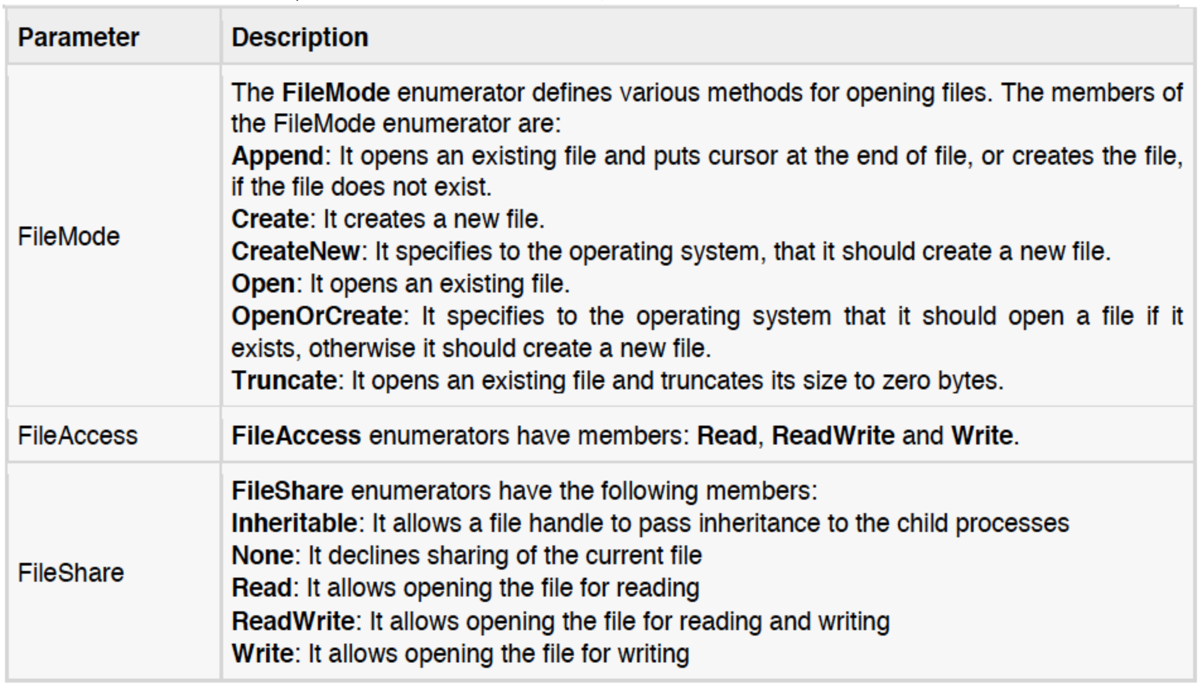
The FileStream class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream. We need to create a

FileStream object to create a new file or open an existing file. The syntax for creating a FileStream object is as follows:

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare
Enumerator>);
```

For example, for creating a FileStream object F for reading a file named sample.txt:

```
FileStream F = new FileStream("sample.txt", FileMode.Open,
FileAccess.Read, FileShare.Read);
```



Parameter	Description
FileMode	The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are: **Append**: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. **Create**: It creates a new file. **CreateNew**: It specifies to the operating system, that it should create a new file. **Open**: It opens an existing file. **OpenOrCreate**: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. **Truncate**: It opens an existing file and truncates its size to zero bytes.
FileAccess	**FileAccess** enumerators have members: **Read**, **ReadWrite** and **Write**.
FileShare	**FileShare** enumerators have the following members: **Inheritable**: It allows a file handle to pass inheritance to the child processes **None**: It declines sharing of the current file **Read**: It allows opening the file for reading **ReadWrite**: It allows opening the file for reading and writing **Write**: It allows opening the file for writing

Example:

```
using System;
using System.IO;
namespace FileIOApplication
{
      class Program
      {
            static void Main(string[] args)
            {
                  FileStream F = new FileStream("test.dat",
                  FileMode.OpenOrCreate, FileAccess.ReadWrite);
                  for (int i = 1; i <= 20; i++)
                  {
                        F.WriteByte((byte)i);
                  }
                  F.Position = 0;
                  for (int i = 0; i <= 20; i++)
                  {
                        Console.Write(F.ReadByte() + " ");
                  }
                  F.Close();
                  Console.ReadKey();
            }
      }
}
```

```
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

3.22 C# Namespace System

Namespaces are heavily used within C# programs in two ways. Firstly, the .NET Framework classes use namespaces to organize its many classes. Secondly, declaring our own namespaces can help control the scope of class and method names in larger programming projects.

3.22.1 Accessing Namespaces

Most C# applications begin with a section of using directives. This section lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time that a method that is contained within is used.
For example, by including the line:
using System;

3.22.2 Namespace Aliases

The using Directive can also be used to create an alias for a namespace. For example, if we are using a previously written namespace that contains nested namespaces, we might want to declare an alias to provide a shorthand way of referencing one in particular, as in the following example:
using Co = Company.Proj.Nested; // define an alias to represent a namespace

The namespace keyword is used to declare a scope. The ability to create scopes within our project helps organize code and let us create globally unique types. In the following example, a class titled *SampleClass* is defined in two namespaces, one nested inside the other. The. Operator is used to differentiate which method gets called.

```
namespace SampleNamespace {
class SampleClass{
      public void SampleMethod()
      {
System.Console.WriteLine("SampleMethod inside SampleNamespace");
      }
}
// Create a nested namespace, and define another class.
namespace NestedNamespace{
      class SampleClass
      {
            public void SampleMethod()
            {
System.Console.WriteLine("SampleMethod inside NestedNamespace");
            }
      }
}
class Program {
      static void Main(string[] args)
      {  // Displays "SampleMethod inside SampleNamespace."
      SampleClass outer = new SampleClass();
      outer.SampleMethod();
      //Display "SampleMethod inside SampleNamespace."
      SampleNamespace.SampleClass outer2 = new
      SampleNamespace.SampleClass();
      outer2.SampleMethod();
```

```
         // Displays "SampleMethod inside NestedNamespace."
        SampleNamespace.NestedNamespace.SampleClass inner = new
        NestedNamespace.SampleClass();
        inner.SampleMethod();
         }
      }
}
```

4 Chapter 4: Dot Net Framework

The .NET Framework is a revolutionary platform created by Microsoft for developing applications. The .NET Framework enables the creation of **Windows applications, Web applications, Web services,** and pretty much anything else we can think of. Also, with Web applications it's worth noting that these are, by definition, multiplatform applications, since any system with a Web browser can access them. With the recent addition of Silverlight, this category also includes applications that run inside browsers on the client, as well as applications that merely render Web content in the form of HTML.

The .NET Framework has been designed so that it can be used from any language, including C# as well as C++, Visual Basic, JScript, and even older languages such as COBOL. For this to work, .NET-specific versions of these languages have also appeared, and more are being released all the time. Not only do all of these have access to the .NET Framework, but they can also communicate with each other. It is perfectly possible for C# developers to make use of code written by Visual Basic programmers, and vice versa. All of this provides an extremely high level of versatility and is part of what makes using the .NET Framework such an attractive prospect.

4.1 What's in the .NET Framework?

The .NET Framework consists primarily of a gigantic library of code that we use from our client languages (such as C#) using object-oriented programming (OOP) techniques. This library is categorize into different modules-we use portions of it depending on the results we want to achieve.
For example, one module contains the building blocks for **Windows applications, another for network programming, and another for Web development**. Some modules are divided into more specific submodules, such as a module for building Web services within the module for Web development.

The intention is for different operating systems to support some or all of these modules, depending on their characteristics. A PDA, for example, would include support for all the core .NET functionality but is unlikely to require some of the more esoteric modules. Part of the .NET Framework library defines some basic types. A type is a representation of data, and specifying some of the most fundamental of these (such as ''a 32-bit signed integer'') facilitates interoperability between languages using the .NET Framework. This is called the **Common Type System (CTS).**
As well as supplying this library, the .Net Framework also includes the .NET **Common Language Runtime (CLR)**, which is responsible for maintaining the execution of all applications, developed using the .NET library.

.NET can be understood as a runtime environment and a comprehensive base class library. The runtime layer is properly referred to as the **Common Language Runtime**, or CLR. The primary role of the CLR is to **locate, load, and manage .NET types** on our behalf. The CLR also takes care of a number of low-level details such as **memory management, application hosting, handling threads, and performing various security checks**.

Another building block of the .NET platform is the **Common Type System**, or CTS. The CTS Specification fully describes all possible data types and programming constructs supported by the runtime, specifies how these entities can interact with each other, and details how they are represented in the .NET metadata format.

Understand that a given .NET-aware language might not support each and every feature defined by the CTS. The **Common Language Specification**, or CLS, is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on. Thus, if we build .NET types that only expose CLS-compliant features, we can rest assured that all .NET-aware languages can consume them. Conversely, if we make use of a data type or programming construct that is outside of the bounds of the CLS, we cannot guarantee that every .NET programming language can interact with our .NET code library. The following figure shows that the .Net Framework.

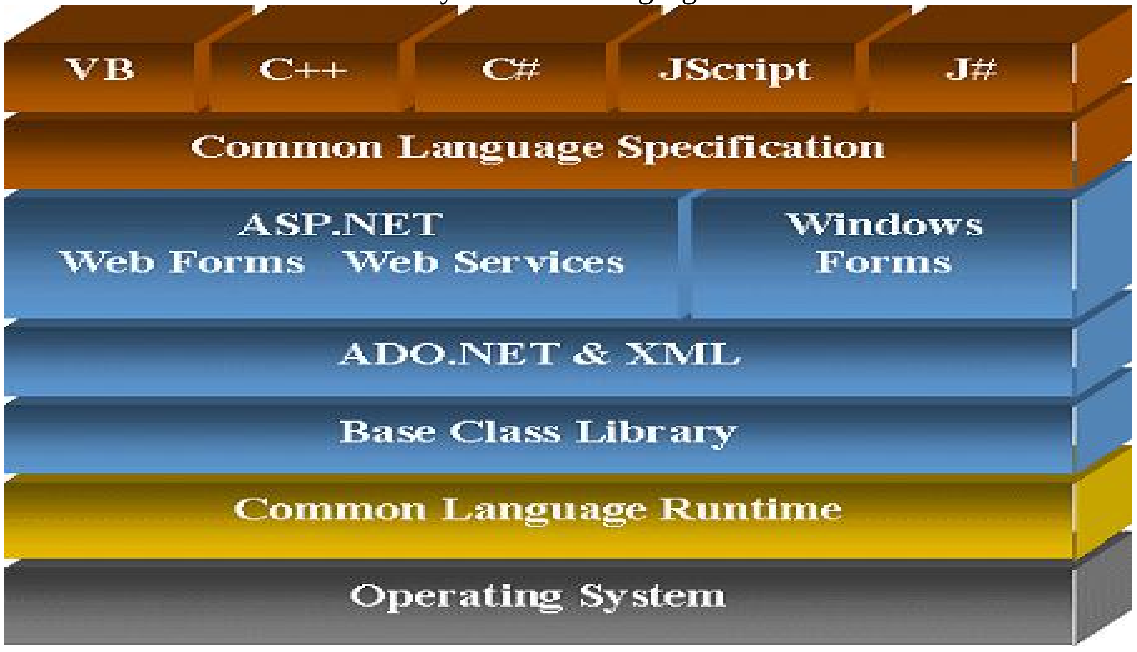


Figure 14: .Net Framework

4.2 The Role of the Base Class Libraries

In addition to the CLR and CTS/CLS specifications, the .NET platform provides a base class library that is available to all .NET programming languages. Not only does this base class library encapsulate various primitives such as **threads, file input/output (I/O), graphical rendering systems, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.**

For example, the base class libraries define types that *facilitate database access, manipulation of XML documents, programmatic security, and the construction of web-enabled as well as traditional desktop and console-based front ends*. From a high level, we can visualize the relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure.

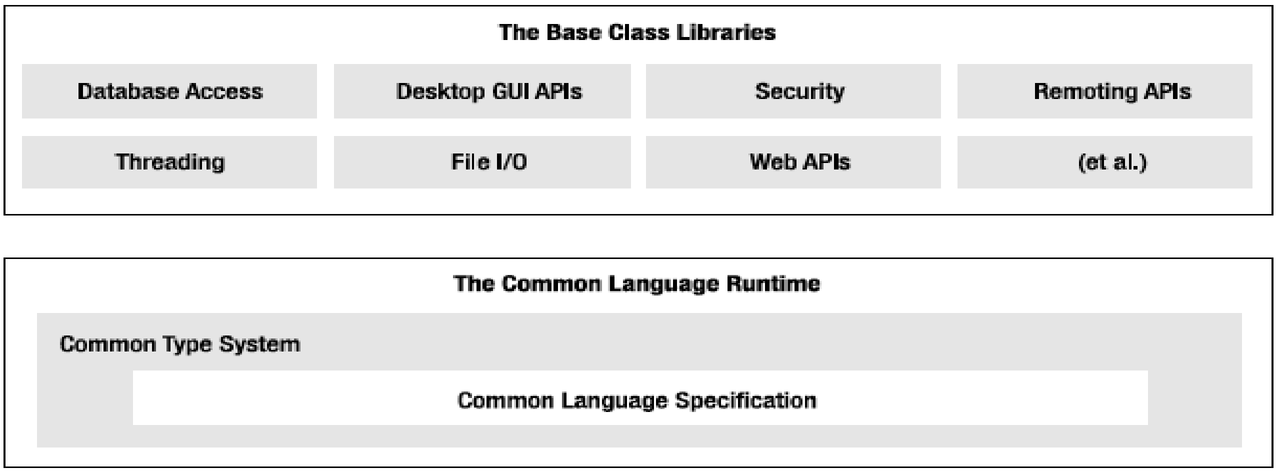


Figure 15: Relationship between CLS, CTS, CLR and base class library

4.2.1 CIL and JIT

When we compile code that uses the .NET Framework library, we don't immediately create operating-system-specific **native code**. Instead, we compile our code into ***Common Intermediate Language (CIL)*** *code*. This code isn't specific to any operating system (OS) and isn't specific to C#. Other .NET languages — Visual Basic .NET, for example — also compile to this language as a first stage. This compilation step is carried out by VS or VCE when we develop C# applications.

Obviously, more work is necessary to execute an application. That is the job of a **just-in-time (JIT)** compiler, which compiles *CIL into native code* that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The just-in-time part of the name reflects the fact that CIL code is compiled only when it is needed.

In the past, it was often necessary to compile our code into several applications, each of which targeted a specific operating system and CPU architecture. Typically, this was a form of optimization (to get code to run faster on an AMD chipset, for example), but at times it was critical (for applications to work in both Win9x and WinNT/2000 environments, for example). This is now unnecessary, because JIT compilers (as their name suggests) use CIL code, which is independent of the *machine, operating system, and CPU.* Several JIT compilers exist, each targeting a different architecture, and the appropriate one is used to create the native code required.

The beauty of all this is that it requires a lot less work on our part - in fact, we can forget about system-dependent details and concentrate on the more interesting functionality of our code.

4.2.2 Assemblies

When we compile an application, the CIL code created is stored in an *assembly*. *Assemblies include both executable application files that we can run directly from Windows without the need for any other programs (these have a* ***.exe*** *file extension) and libraries (which have a* ***.dll*** *extension) for use by other applications.*

In addition to containing CIL, assemblies also include meta information (that is, information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the CIL, such as sound files and pictures). The meta information enables assemblies to be fully self-descriptive. We need no other information to use an assembly, meaning we avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, we can just run an executable file from this directory (assuming the .NET CLR is installed).

Of course, we won't necessarily want to include everything required to run an application in one place. We might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the **global assembly cache (GAC)**. Placing code in the GAC is simple - we just place the assembly containing the code in the directory containing this cache.

4.2.3 Managed Code

The role of the CLR doesn't end after we have compiled our code to CIL and a JIT compiler has compiled that to native code. Code written using the .NET Framework is managed when it is executed (a stage usually referred to as runtime). This means that the CLR looks after our applications by *managing memory, handling security, allowing cross-language debugging*, and so on. By contrast, applications that do not run under the control of the CLR are said to be *unmanaged*, and certain languages such as C++ can be used to write such applications, which, for example, access low-level functions of the operating system. However, in C# we can write only code that runs in a managed environment. We will make use of the managed features of the CLR and allow .NET itself to handle any interaction with the operating system.

4.2.4 Garbage Collection

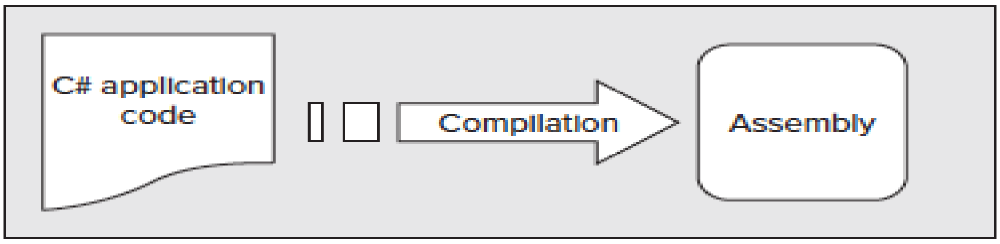
One of the most important features of managed code is the concept of garbage collection. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use. Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of our computer followed by a system crash.

.NET garbage collection works by periodically inspecting the memory of our computer and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but we can rest assured that it will happen.

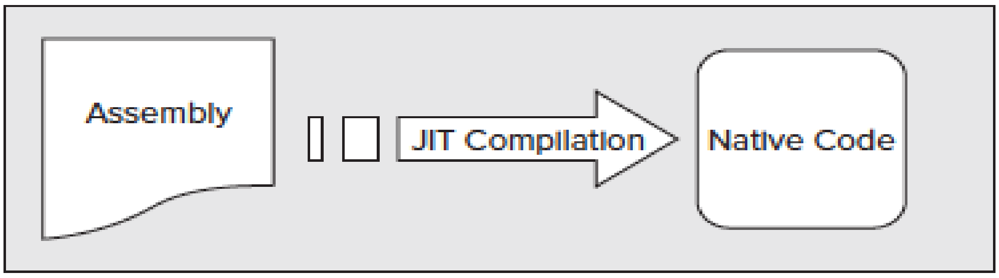
There are some implications for programmers here. Because this work is done for us at an unpredictable time, applications have to be designed with this in mind. Code that requires a lot of memory to run should tidy itself up, rather than wait for garbage collection to happen, but that isn't as tricky as it sounds.

4.3 Summary of steps required to create a .NET application

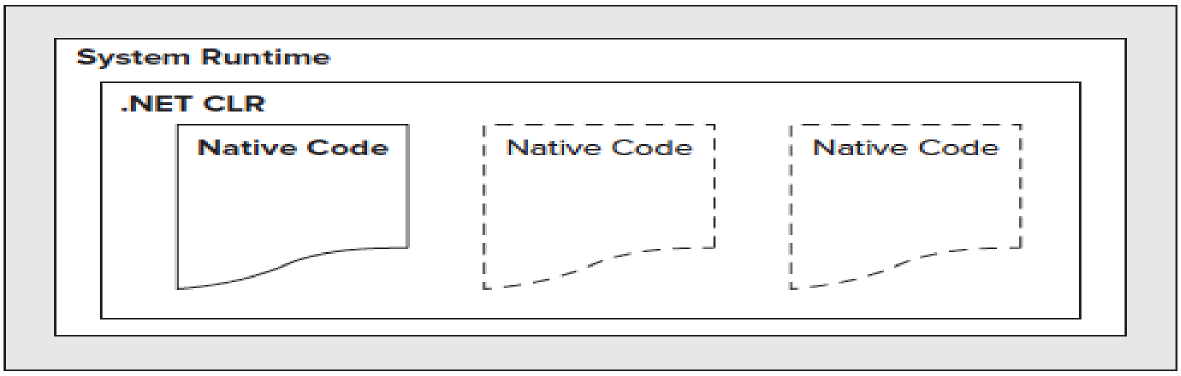
1. Application code is written using a .NET-compatible language such as C#.
2. That code is compiled into CIL, which is stored in an assembly.



3. When this code is executed (either in its own right if it is an executable or when it is used from other code), it must first be compiled into native code using a JIT compiler.



4. The native code is executed in the context of the managed CLR, along with any other running applications or processes.



4.3.1 Linking

The C# code that compiles into CIL in step 2 needn't be contained in a single file. It's possible to split application code across multiple source code files, which are then compiled together into a single assembly. This extremely useful process is known as **linking**. It is required because it is far easier to work with several smaller files than one enormous one. We can separate out logically related code into an individual file so that it can be worked on independently and then practically forgotten about when completed. This also makes it easy to locate specific pieces of code when we need them and enables teams of developers to divide the programming burden into manageable chunks, whereby individuals can ''check out'' pieces of code to work on without risking damage to otherwise satisfactory sections or sections other people are working on.

4.4 Windows Forms

Since the release of the .NET platform (circa 2001), the base class libraries have included a particular API named Windows Forms, represented primarily by the **System.Windows.Forms.dll** assembly. The Windows Forms toolkit provides the types necessary to build desktop graphical user interfaces (GUIs), create custom controls, manage resources (e.g., string tables and icons), and perform other desktop centric programming tasks. In addition, a separate API named GDI+ (represented by the System.Drawing.dll assembly) provides additional types that allow programmers to generate 2D graphics, interact

with networked printers, and manipulate image data.
The Windows Forms (and GDI+) APIs remain alive and well within the .NET 4.0 platform, and they will exist within the base class library for quite some time (arguably forever). However, Microsoft has shipped a brand new GUI toolkit called Windows Presentation Foundation (WPF) since the release of .NET 3.0. WPF provides a massive amount of horsepower that we can use to build bleeding-edge user interfaces, and it has become the preferred desktop API for today's .NET graphical user interfaces. When we need to create more traditional business UIs that do not require an assortment of bells and whistles, the Windows Forms API can often fit the bill.

The Windows Forms Namespaces
The Windows Forms API consists of hundreds of types (e.g., classes, interfaces, structures, enums, and delegates), most of which are organized within various namespaces of the **System.Windows.Forms.dll** assembly.

Far and away the most important Windows Forms namespace is *System.Windows.Forms*. At a high level, we can group the types within this namespace into the following broad categories:

- Core infrastructure: These are types that represent the core operations of a Windows Forms program (e.g., Form and Application) and various types to facilitate interoperability with legacy ActiveX controls, as well as interoperability with new WPF custom controls.
- Controls: These are types used to create graphical UIs (e.g., Button, MenuStrip, ProgressBar, and DataGridView), all of which derive from the Control base class. Controls are configurable at design time and are visible (by default) at runtime.
- Components: These are types that do not derive from the Control base class, but still may provide visual features to a Windows Forms program (e.g., ToolTip and ErrorProvider). Many components (e.g., the Timer and System.ComponentModel.BackgroundWorker) are not visible at runtime, but can be configured visually at design time.
- Common dialog boxes: Windows Forms provides several canned dialog boxes for common operations (e.g., OpenFileDialog, PrintDialog, and ColorDialog).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// The minimum required windows forms namespaces.
using System.Windows.Forms;
namespace SimpleWinFormsApp
{
// This is the application object.
      class Program
      {
            static void Main(string[] args)
            {
                  Application.Run(new MainWindow());
            }
       }
            // This is the main window.
      class MainWindow : Form {}
}
```

This code represents the absolute simplest Windows Forms application and will produce following output.

4.5 Windows Presentation Foundation (WPF)

WPF, or Windows Presentation Foundation, is a graphical system for rendering user interfaces. It provides great flexibility in how we can lay out and interact with our applications. With Common Language Runtime (CLR) at its core, we can use C# or any other CLR language to communicate with user interface elements and develop application logic. The advantages of WPF for our application are its *rich data binding and visualization support and its design flexibility and styling*. WPF enables us to create an application that is more usable to our audience. It gives us the power to design an application that would previously take extremely long development cycles and a calculus genius to implement.

WPF's graphics capabilities make it the perfect choice for data visualization. Take, for instance, the standard drop-down list (or combo box). Its current use is to enable the user to choose a single item from a list of items. For this example, suppose we want the user to select a car model for purchase.
The standard way of displaying this choice is to display a drop-down list of car model names from which users can choose. There is a fundamental usability problem with this common solution: Users are given only a single piece of information from which to base their decision — the text that is used to represent the item in the list.

For the power user (or car fanatic) this may not be an issue, but other users need more than just a model name to make an educated decision on the car they wish to purchase. This is where WPF and its data visualization capabilities come into play.

A template can be provided to define how each item in the drop-down list is rendered. The template can contain any visual element, such as images, labels, text boxes, tooltips, drop shadows, and more.

4.5.1 XAML

WPF enables we to build user interfaces declaratively. XAML forms the foundation of WPF. XAML is similar to HTML in the sense that interface elements are defined using a tag-based syntax. XAML is XML-based and as such it must be **well formed**, meaning all opening tags require closing tags, and all elements and attributes contained in the document must validate strictly against the specified **schemas**. By default, when creating a WPF application in Visual Studio 2010, the following schemas are represented in generated XAML files:
• *http://schemas.microsoft.com/winfx/2006/xaml/presentation*: This schema represents the default Windows Presentation Framework namespace.
• *http://schemas.microsoft.com/winfx/2006/xaml*: This schema represents a set of classes that map to CLR objects. Most CLR objects can be expressed as XAML elements. XAML elements are mapped to classes; attributes are mapped to properties or events.

At runtime when a XAML element is processed, the default constructor for its underlying class is called, and the object is instantiated; its properties and events are set based on the attribute values specified in XAML.

When creating a WPF application from visual studio 2010, an Application template creates two XAML files along with their respective code-behind files: *App.xaml* (*App.xaml.cs*) and *MainWindow.xaml (MainWindow.xaml.cs).*

App.xaml represents the entry-point of the application. This is where application- wide (globally scoped) resources and the startup window are defined. *Resources are a keyed collection of reusable objects. Resources can be created and retrieved using both XAML and C#. Resources can be anything —* ***data templates, arrays of strings, or brushes used to color the background of text boxes.*** *Resources are also scoped, meaning they can be available to the entire Application (global), to the Window, to the User Control, or even to only a specific control.*

```
<Application x:Class=”MyFirstWPFApplication.App”
             xmlns=”http://schemas.microsoft.com/winfx/2006/xaml/presentation”
             xmlns:x=”http://schemas.microsoft.com/winfx/2006/xaml”
             StartupUri=”MainWindow.xaml”>
     <Application.Resources>
     </Application.Resources>
</Application>
```

The above code is the XAML that was generated by the WPF Application template in Visual Studio. Note that the WPF namespaces are defined. The namespace that represents the CLR objects will be distinguished in the XAML file with the **.x** prefix.
The **StartupUri** value defines the window that will be displayed after the application is executed. In this case, the **MainWindow.xaml** window will be displayed. The **x:Class** attribute defines the C# code-behind file of this XAML file. If we open **App.xaml.cs**, we will see that its class name is **App** and it inherits from the **Application** class (which is the root element of the XAML file).

WPF brings not only a dramatic shift to the look and feel of Windows applications but also changes the manner of development. The days of dragging and dropping controls from the toolbox onto a form are long gone. Even though it is still possible to drag and drop in WPF, we will find yourself better off and much happier if we work in XAML directly.

What was once difficult is now relatively simple. For example, in traditional Windows applications, when the user changes the size of the form, the controls typically stay huddled in their corner and a large area of empty canvas is displayed. The only cure for this was a lot of custom code or expensive third-party controls. WPF brings the concept of flow layout from the Web
into the Windows world.

- A Form in WinForms is referred to as a Window in WPF.
- Anything placed on a WinForms Form is called a control, whereas items placed on a WPF Window are referred to as UIElements.
- Panels are WPF UIElements used for layout.
- A Control in WPF is a UIElement that can receive focus and respond to user events.
- A Content control in WPF can contain only a single item, which can in turn be other UIElements.
- The WPF Window class is a specialized Content control.

Instead of depending on screen resolution, WPF measures UI Elements in Device Independent Units (DIUs) that are based on the system DPI. This enables a consistent look between many different hardware configurations.

WPF layout is based on relative values and is adjusted at runtime. When we place controls in a layout container, the rendering engine considers the height and width only as “suggested” values. Location is defined in relation to other controls or the container. Actual rendering is a two-step process that starts with measuring all controls (and querying them for their preferred dimensions) and then arranging them accordingly. If controls could speak, the conversation might go something like this: Layout Engine: “Control, how much space would you like to have?”
<This is the Measure Stage>
Control: “I would like 50 DIUs for height, 100 DIUs for width, and a margin of 3 DIUs in the containing Grid cell.”
Layout Engine asks all other controls and layout containers.
Layout Engine: “Sorry, you can have only 40 DIUs for height, but I can grant the rest of your requests.”
<This is the Arrange Stage>

4.5.2 Arranging elements with layout panel

Designing a Window begins with a Layout control, or Panel. Panels are different than Content controls in that they can hold multiple items, and depending on the Panel, a significant amount of plumbing is taken care of for us. Panels control how *UIElements* relate to each other and to their containing UIElement and do not dictate absolute positioning. Most application Windows requires some combination of Panels to achieve the required user interface, so it’s important to understand them all. WPF ships with six core Panels:
Stack Panel: Stack Panels place UIElements in -wait for it- stacks. Items are placed in either a vertical pile (the default), like a stack of DVDs, or a horizontal arrangement, like books on a shelf. It is important to understand that the order items appear in the XAML is the order they appear in the Panel — the first UIElement in the XAML appears at the top (vertical) or on the far left (horizontal)

✦ **Wrap Panel:** The Wrap Panel automatically wraps overflow content onto the next line(s). This is different than how a typical toolbar works, where overflow items are hidden when there isn’t enough real estate to show them.

✦ **Dock Panel:** The Dock Panel uses attached properties to “dock” child UIElements. An important thing to remember is that child elements are docked in XAML order, which means if we have two items assigned to the left side, the first UIElement as it appears in the XAML gets the far left wall of the Panel, followed by the next item.

✦ **Canvas:** The Canvas is a bit of an anomaly in WPF, since it doesn’t use flow layout, but goes back to fix position layout rendering.

✦ **Uniform Grid:** The Uniform Grid divides the layout area into equally sized cells. The number of Rows and Columns are defined in the UniformGrid XAML tag.

✦ **Grid:** The Grid is in fact the default panel in a Window when we add a new WPF Window to our project. The Grid divides the layout area with rows (RowDefinitions) and columns (ColumnDefinitions). The difference between the Grid and the Uniform Grid is that the Grid allows for sizing of the cells by defining RowDefinition Height and ColumnDefinition Width.

4.6 Windows Communication Foundation

Windows Communication Foundation is just that the foundation for communication between Windows computers. It just so happens that thanks to open standards like SOAP and ReST, WCF can communicate with other software systems.

Although ASMX was really designed to make public services such as, for instance, adding an API to a simple Web application, WCF is a complete distributed computing platform for Windows.

In the early days of .NET, there was a technology called .NET Remoting that replaced DCOM. DCOM was Distributed COM, or the common accepted way to communicate between distributed components. Remoting replaced it when .NET came out. Remoting basically took the principles of DCOM and migrated them to .NET.

WCF isn't like that. WCF is a complete *rethinking of distributed computing*, based on the understanding that computing is becoming more and more distributed. New protocols for communication come out every day.

The goal here then is to define the differences between ASMX and WCF, and see that WCF is a true communications protocol and that ASMX is solely for adding services to Web sites. We can use either technology for both tasks, but one is certainly more suited than the other for each.

First there was DCOM. Then there was .NET Remoting. The path to a distributed computing platform for Microsoft has been a long one. Distributed computing is a hard problem to solve; nothing bad on the Microsoft developers for continuing to hone their platform.
Anyway, there is SOAP, there are Microsoft binary formats, people are creating custom HTTP contexts: the distributed computing platform is a mess. Something needed to happen to enable us to all take our existing code and make it available across the enterprise.
The designers of WCF (largely Doug Purdy, Don Box, and crew) had two diverse issues. On one hand was ASMX, providing SOAP Web service access to business logic. On the other hand was .NET Remoting, providing Microsoft a custom, black box format for the transmission of information between components via known network protocols.

Taking a deeper look at Remoting makes the case for a comprehensive platform even more clear.

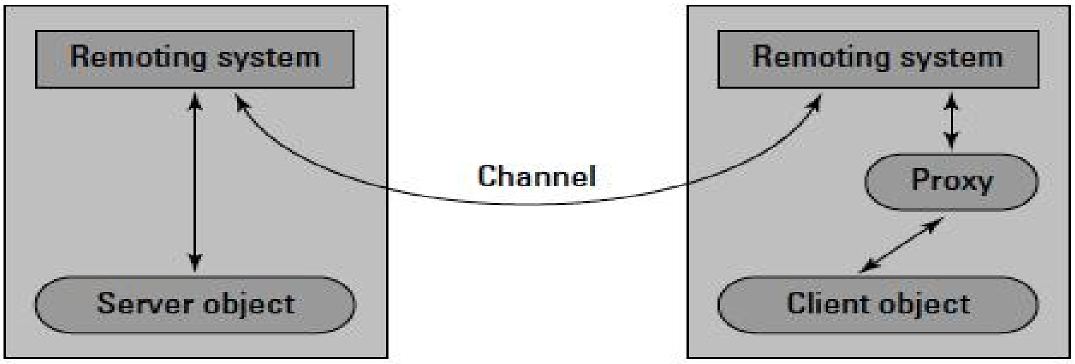


Figure 16: The Remoting System

If each of two systems, say a Client System and a Server System, had a similarly configured Remoting system, then they could communicate. The problem, of course, is that it was never, ever configured correctly (or so it seemed). One side of the equation or the other will make some change, and the whole system will come to a crashing halt. Clearly, there has to be some product that brings the various formats of remote access together under one umbrella. Something that would accept one block of logic and provide multiple service types.
Eventually, WCF was that solution. Starting as an add-on to .NET 2.0, WCF effectively enabled developers to make specific endpoints for a generic connector.

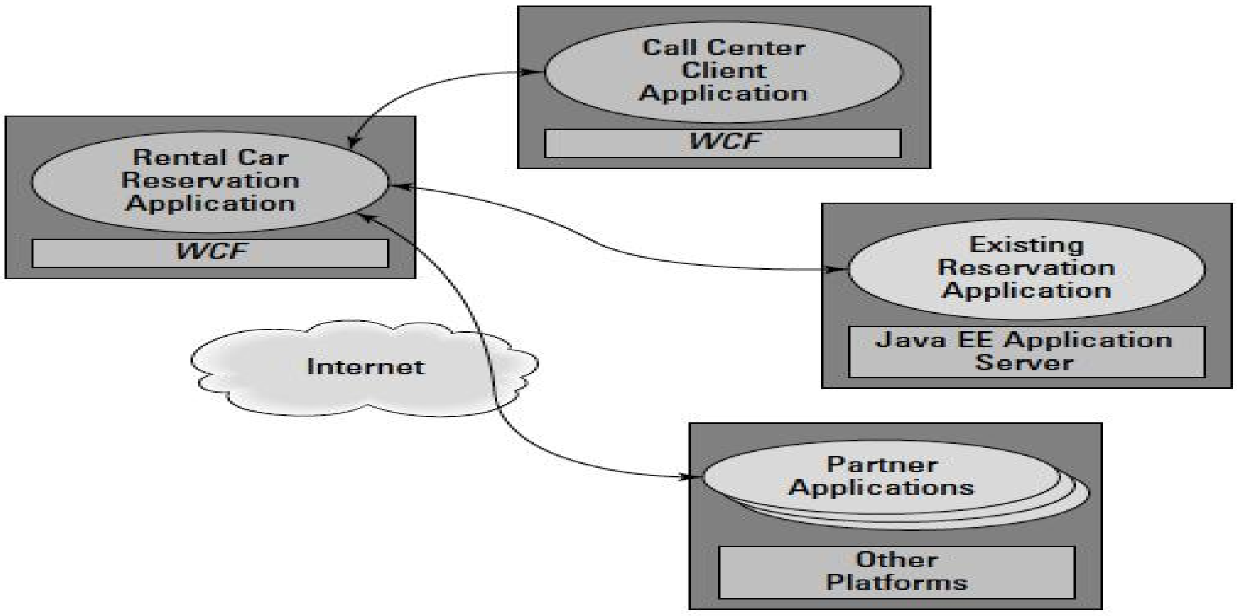

Call Center Client Application
WCF
Rental Car Reservation Application
WCF
Existing Reservation Application
Java EE Application Server
Internet
Partner Applications
Other Platforms

Fig: Original Problem to be solved
A car dealership is trying to build a new reservation application. The business logic needs to be both accessible to outside applications and provide a quality binary transport format for the internal communication. WCF is the answer. It uses configuration to provide various endpoints to consuming applications, from SOAP to ReST to binary associations that resemble DCOM. It doesn't require configuration on both ends of the pipe, only on the server side. If the server serves it, the client can consume it.

4.6.1 Using different endpoints

The whole point of WCF is that we can write one service and then have IIS accept a lot of different protocols calling on the same code. Sure, we can do SOAP, like we did in ASMX. We also can do a lot of other protocols. Here is the breakdown of all the different endpoints, according to MSDN:

1. **BasicHttpBinding:** A binding that is suitable for communicating with WS-Basic Profile conformant Web services (for example, ASP.NET Web services [ASMX]-based services). This binding uses HTTP as the transport and text/XML as the default message encoding.
2. **WSHttpBinding:** A secure and interoperable binding that is suitable for non-duplex service contracts.
3. **WSDualHttpBinding**: A secure and interoperable binding that is suitable for duplex service contracts or communication through SOAP intermediaries.
4. **WSFederationHttpBinding:** A secure and interoperable binding that supports the WS-Federation protocol that enables organizations that is in a federation to efficiently authenticate and authorize users.
5. **NetTcpBinding**: A secure and optimized binding suitable for cross-machine

communication

6. between WCF applications.
7. **NetNamedPipeBinding:** A secure, reliable, optimized binding that is suitable for on-machine communication between WCF applications.
8. **NetMsmqBinding:** A queued binding that is suitable for cross-machine communication between WCF applications.
9. **NetPeerTcpBinding**: A binding that enables secure, multiple machine communication.
10. **MsmqIntegrationBinding:** A binding that is suitable for cross-machine communication between a WCF application and existing Message Queuing applications.
11. **BasicHttpContextBinding**: A binding that is suitable for communicating with WS-Basic Profile conformant Web services that enables HTTP cookies to be used to exchange context.
12. **NetTcpContextBinding**: A secure and optimized binding suitable for cross-machine communication between WCF applications that enables SOAP headers to be used to exchange context.
13. **WebHttpBinding:** A binding used to configure endpoints for WCF Web services that are exposed through HTTP requests instead of SOAP messages.
14. **WSHttpContextBinding:** A secure and interoperable binding that is suitable for non-duplex service contracts that enables SOAP headers to be used to exchange context.

So when do you use what? Well, if you want to have two .NET applications communicate, or th layers of a single application communicate, then use NetTcpBinding. If you are cross-platform communicating, then use WSHttpBinding. Only if for some reason those don’t work do you start to look into the others. Oh, and remember that you can have many endpoints for the same service. Don’t use SOAP for a data layer. Just make a separate SOAP header if you need cross-platform support.

5 Chapter 5: Web and Database Programming with .Net

Active Server Page (ASP): Active Server Pages (ASP) is a server-side scripting technology that can be used to create dynamic and interactive Web applications. An ASP page is an HTML page that contains server-side scripts that are processed by the Web server before being sent to the user's browser. We can combine ASP with Extensible Markup Language (XML), Component Object Model (COM), and Hypertext Markup Language (HTML) to create powerful interactive Web sites.
Server-side scripts run when a browser requests an .asp file from the Web server. ASP is called by the Web server, which processes the requested file from top to bottom and executes any script commands. It then formats a standard Web page and sends it to the browser.
An Example of classical asp page. This example displays the words "Hello World".

```
<%@ Language=VBScript %>
<html> <head> <title>Example 1</title> </head>
        <body>
                <% FirstVar = "Hello world!" %>
                <%=FirstVar%>
        </body>
</html>
```

5.1 ASP.Net

When the first version of the .NET Framework was released nearly a decade ago, it was the start of a radical new direction in software design. Inspired by the best of Java, COM, and the Web, and informed by the mistakes and limitations of previous technologies, Microsoft set out to “hit the reset button” on their development platform. The result was a set of surprisingly mature technologies that developers could use to do everything from building a Windows application to executing a database query, and a web-site-building tool known as ASP.NET.

Today, ASP.NET is as popular as ever, but it’s no longer quite as revolutionary. And, although the basic functionality that sits at the heart of ASP.NET is virtually the same as it was ten years ago, Microsoft has added layers of new features and higher-level coding abstractions. It has also introduced at least one new direction that competes with traditional ASP.NET programming, which is called ASP.NET MVC.

5.1.1 Seven key Pillars of ASP.Net

1. ASP.NET is integrated with the .NET Framework: The .NET Framework is divided into an almost painstaking collection of functional parts, with tens of thousands of types (the .NET term for classes, structures, interfaces, and other core programming ingredients). The massive collection of functionality that the .NET Framework provides is organized in a way those traditional Windows programmers will see as a happy improvement. Each one of the thousands of classes in the .NET Framework is grouped into a logical, hierarchical container called a namespace.

Different namespaces provide different features. Taken together, the .NET namespaces offer functionality for nearly every aspect of distributed development from message queuing to security. This massive toolkit is called the class library.

2. ASP.NET Is Compiled, Not Interpreted: ASP.NET applications, like all .NET applications, are always compiled. In fact, it's impossible to execute C# or Visual Basic code without it being compiled first. .NET applications actually go through two stages of compilation. In the **first stage**, the C# code is compiled into an intermediate language called Microsoft Intermediate Language (MSIL), or just IL. This first step is the fundamental reason that .NET can be language-interdependent. Essentially, all .NET languages (including C#, Visual Basic, and many more) are compiled into virtually identical IL code. This first compilation step may happen automatically when the page is first requested. The compiled file with IL code is an assembly.
The second level of compilation happens just before the page is actually executed. At this point, the IL code is compiled into low-level native machine code. This stage is known as **just in time (JIT)** compilation, and it takes place in the same way for all .NET applications (including Windows Applications, for example). Figure below shows this two-step compilation process.
.NET compilation is decoupled into two steps in order to offer developers the most convenience and the best portability. Before a compiler can create low-level machine code, it needs to know what type of operating system and hardware platform the application will run on (for example, 32-bit or 64-bit Windows). By having two compile stages, we can create a compiled assembly with .NET code and still distribute this to more than one platform.

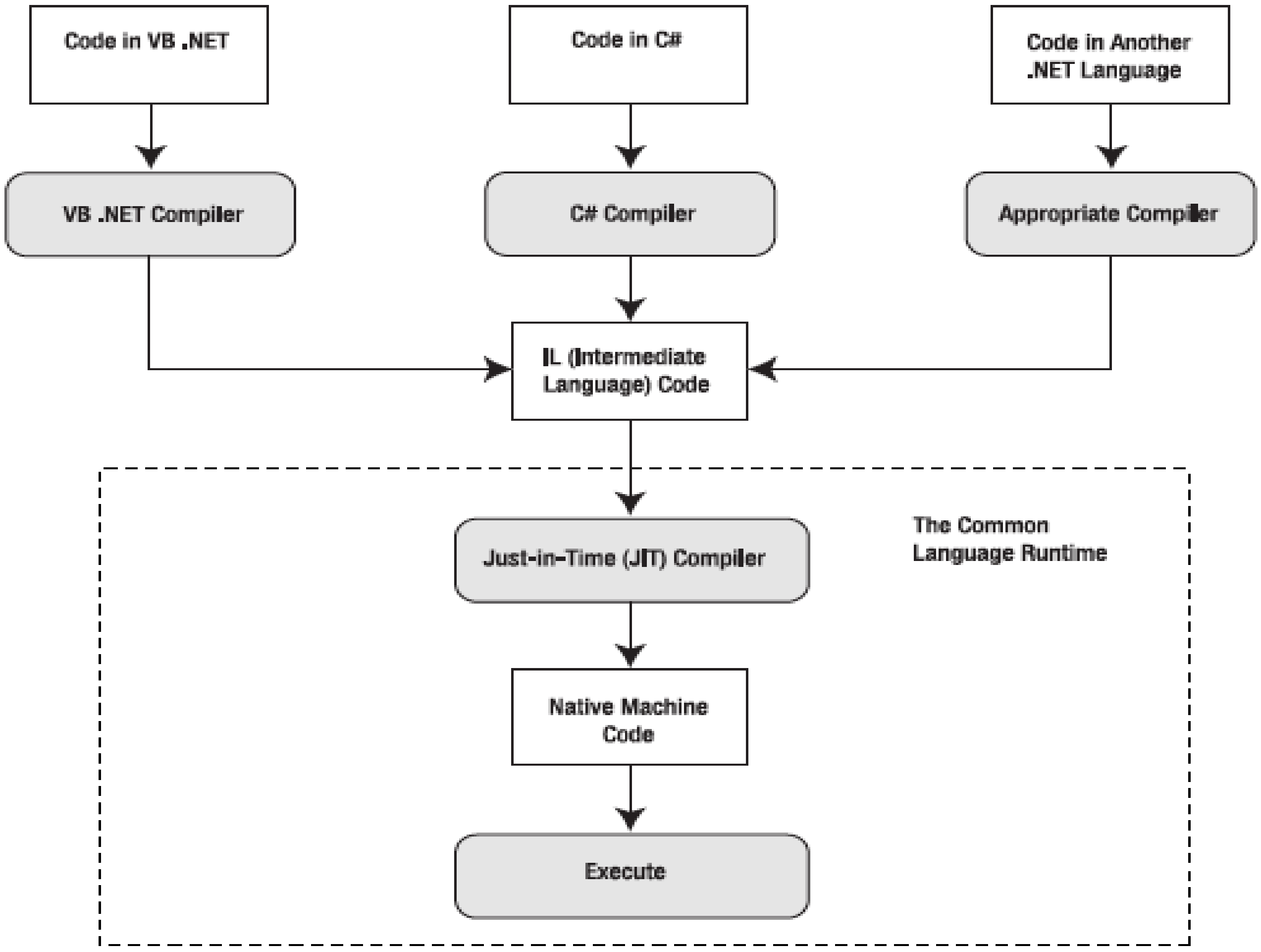

Code in VB .NET
Code in C#
Code in Another .NET Language
VB .NET Compiler
C# Compiler
Appropriate Compiler
IL (Intermediate Language) Code
Just-in-Time (JIT) Compiler
The Common Language Runtime
Native Machine Code
Execute

Figure 17: Compilation of asp.net web page

JIT compilation probably wouldn't be that useful if it needed to be performed every time a user requested a web page from our site. Fortunately, ASP.NET applications don't need to be compiled every time a web page is requested. Instead, the IL code is created once and regenerated only when the source is modified. Similarly, the native machine code files are cached in a system directory that has a path.

3. ASP.NET Is Multilanguage: Many programmers 'll probably opt to use one language over another when one develop an application, that choice won't determine what can accomplish with web applications. That's because no matter what language we use, the code is compiled into IL.

4. ASP.NET Is Hosted by the Common Language Runtime: Perhaps the most important aspect of the ASP.NET engine is that it runs inside the runtime environment of the CLR. The whole of the .NET Framework—that is, all namespaces, applications, and classes—is referred to as managed code. Some benefits of CLR are Automatic memory management and garbage collection, type safety, Extensible metadata, structured error handling, Multithreading etc.

5. ASP.NET Is Object-Oriented: ASP provides a relatively feeble object model. It provides a small set of objects; these objects are really just a thin layer over the raw details of HTTP and HTML. On the other hand, ASP.NET is truly object oriented.
Not only does our code have full access to all objects in the .NET Framework, but we can also exploit all the conventions of an OOP (object-oriented programming) environment. For example, we can create reusable classes, standardize code with interfaces, extend existing classes with inheritance, and bundle useful functionality in a distributable, compiled component.

One of the best examples of object-oriented thinking in ASP.NET is found in server-based controls.Server-based controls are the epitome of encapsulation. Developers can manipulate control objects programmatically using code to customize their appearance, provide data to display, and even react to events. The low-level HTML markup that these controls render is hidden away behind the scenes.

6. ASP.NET supports all Browsers: Different browsers, versions, and configurations differ in their support of XHTML, CSS, and JavaScript. Web developers need to choose whether they should render their content according to the lowest common denominator, and whether they should add ugly hacks to deal with well-known quirks on popular browsers.

ASP.NET addresses this problem in a remarkably intelligent way. Although we can retrieve information about the client browser and its capabilities in an ASP.NET page, ASP.NET actually encourages developers to ignore these considerations and use a rich suite of web server controls.

7. ASP.NET Is Easy to Deploy and Configure: One of the biggest headaches a web developer faces during a development cycle is deploying a completed application to a production server. Not only do the web-page files, databases, and components need to be transferred, but also components need to be registered and a slew of configuration, settings need to be re-created. ASP.NET simplifies this process considerably. Every installation of the .NET Framework provides the same core classes. As a result, deploying an ASP.NET application is relatively simple. For no-frills deployment, we simply need to copy all the files to a virtual directory on a production server (using an FTP program or even a command-line command like XCOPY). As long as the host machine has the .NET Framework, there are no time-consuming registration steps.

5.1.2 ASP.NET File Types

File Type	Description
Ends with .aspx	These are ASP.NET web pages. They contain the user interface and, optionally, the underlying application code. Users request or navigate directly to one of these pages to start web application.
Ends with .ascx	These are ASP.NET user controls. User controls are similar to web pages, except that the user can't access these files directly. Instead, they must be hosted inside an ASP.NET web page. User controls allow us to develop a small piece of user interface and reuse it in as many web forms, as we want without repetitive code.
web.config	This is the XML-based configuration file for ASP.NET application. It includes settings for customizing security, state management, memory management, and much more.
global.asax	This is the global application file. We can use this file to define global variables (variables that can be accessed from any web page in the web application) and react to global events (such as when a web application first starts).
Ends with .cs	These are code-behind files that contain C# code. They allow us to separate the application logic from the user interface of a web page.

Table 13: ASP.Net File Types

5.1.3 ASP.NET Application Directories

Every web application should have a well-planned directory structure.

Directory	Description
App_Browsers	Contains .browser files that ASP.NET uses to identify the browsers that are using our application and determine their capabilities. Usually, browser information is standardized across the entire web server, and we don't need to use this folder.
App_Code	Contains source code files that are dynamically compiled for use in our Application
App_GlobalResources	Stores global resources that are accessible to every page in the web Application. This directory is used in localization scenarios, when we need to have a website in more than one language
App_LocalResources	Serves the same purpose as App_GlobalResources, except these resources are accessible to a specific page only.
App_WebReferences	Stores references to web services, which are remote code routines that a web application can call over a network or the Internet.
App_Data	Stores data, including SQL Server Express database files
App_Themes	Stores the themes that are used to standardize and reuse formatting in your web application.
Bin	Contains all the compiled .NET components (DLLs) that the ASP.NET web application uses.

Table 14: ASP.Net Application Directories

5.1.4 ASP.Net Page Life Cycle

In general terms, the page goes through the stages outlined in the following table. In addition to the page life-cycle stages, there are application stages that occur before and after a request but are not specific to a page.

Stage	Description
Page request	The page request occurs before the page life cycle begins. When the page is requested by a user, ASP.NET determines whether the page needs to be parsed and compiled (therefore beginning the life of a page), or whether a cached version of the page can be sent in response without running the page.
Start	In the start stage, page properties such as Request and Response are set. At this stage, the page also determines whether the request is a postback or a new request and sets the IsPostBack property.
Initialization	During page initialization, controls on the page are available and each control's UniqueID property is set. A master page and themes are also applied to the page if applicable. If the current request is a postback, the postback data has not yet been loaded and control property values have not been restored to the values from view state.
Load	During load, if the current request is a postback, control property are loaded with information recovered from view state and control state.
Postback event handling	If the request is a postback, control event handlers are called. After that, the Validate method of all validator controls is called, which sets the IsValid property of individual validator controls and of the page.
Rendering	Before rendering, view state is saved for the page and all controls. During the rendering stage, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream object of the page's Response property.
Unload	The Unload event is raised after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and cleanup is performed.

Table 15: ASP.Net Page Life Cycle

5.1.5 INTRODUCING SERVER CONTROL

ASP.NET introduces a remarkable new model for creating web pages. In old-style web development, programmers had to master the quirks and details of HTML before they could design a dynamic web page. Pages had to be carefully tailored to a specific task, and the only way to generate additional content was to generate raw HTML tags.
ASP.NET solves this problem with a higher-level model of server controls. These controls are created and configured as objects. *They run on the web server and they automatically provide their own HTML output*. Even better, server controls behave like their Windows counterparts by *maintaining state and raising events* that we can react to in code.

ASP.NET actually provides two sets of server-side controls that we can incorporate into our web forms. These two different types of controls play subtly different roles:
1) HTML server controls: These are server-based equivalents for standard HTML elements. These controls are ideal if we're a seasoned web programmer who prefers to work with familiar HTML tags (at least at first). They are also useful when migrating ordinary HTML pages or ASP pages to ASP.NET, because they require the fewest changes. HTML controls have three main features:
i) ***They generate their own interface***: We set properties in code, and the underlying HTML tag is created automatically when the page is rendered and sent to the client.
ii) ***They retain their state***: Because the Web is stateless, ordinary web pages need to do a lot of work to store information between requests. HTML server controls handle this task automatically. For example, if the user selects an item in a list box, that item remains selected

the next time the page is generated. Or, if our code changes the text in a button, the new text sticks the next time the page is posted back to the web server.
iii) ***They fire server-side events***: For example, buttons fire an event when clicked, text boxes fire an event when the text they contain is modified, and so on. Our code can respond to these events, just like ordinary controls in a Windows application. With event-based programming, we can easily respond to individual user actions and create more structured code. If a given event doesn't occur, the event handler won't be executed.

2) Web controls: These are similar to the HTML server controls, but they provide a richer object model with a variety of properties for style and formatting details. They also provide more events and more closely resemble the controls used for Windows development. Web controls also feature some user interface elements that have no direct HTML equivalent, such as the GridView, Calendar, and validation control. Following are the features of Web Controls.
i) They provide a rich user interface*:* A web control is programmed as an object but doesn't necessarily correspond to a single element in the final HTML page. For example, we might create a single Calendar or GridView control, which will be rendered as dozens of HTML elements in the final page. When using ASP.NET programs, we don't need to know anything about HTML. The control creates the required HTML tags for us.
ii) They provide a consistent object model: HTML is full of quirks and idiosyncrasies. For example, a simple text box can appear as one of three elements, including <*textarea*>, <*input type="text"*>, *and* <*input type="password"*>. With web controls, these three elements are consolidated as a single *TextBox* control. Depending on the properties we set, the underlying HTML element that ASP.NET renders may differ. Similarly, the names of properties don't follow the HTML attribute names. For example, controls that display text, whether it's a caption or a text box that can be edited by the user, expose a Text property.
iii) They tailor their output automatically: ASP.NET server controls can detect the type of browser and automatically adjust the HTML code they write to take advantage of features such as support for JavaScript. We don't need to know about the client because ASP.NET handles that layer and automatically uses the best possible set of features. This feature is known as adaptive rendering.
iv) They provide high-level features: We'll see that web controls allow us to access additional events, properties, and methods that don't correspond directly to typical HTML controls. ASP.NET implements these features by using a combination of tricks.

5.1.6 DATA ACCESS

The .NET Framework includes its own data access technology: ADO.NET. ADO.NET consists of managed classes that allow .NET applications to connect to data sources (usually relational databases), execute commands, and manage disconnected data. The small miracle of ADO.NET is that it allows us to write more or less the same data access code in web applications that we write for client-server desktop applications, or even single-user applications that connect to a local database.

5.1.6.1 Database Access without ADO.NET

In ASP.NET, there are a few ways to get information out of a database without directly using the ADO.NET classes. Depending on needs, we may be able to use one or more of these approaches to supplement our database code (or to avoid writing it altogether).
An options for database access without ADO.NET include the following:
• **The SqlDataSource control**: The SqlDataSource control allows us to define queries declaratively. We can connect the SqlDataSource to rich controls such as the GridView, and give our pages the ability to edit and update data without requiring any ADO.NET code. Best

of all, the SqlDataSource uses ADO.NET behind the scenes, and so it supports any database that has a full ADO.NET provider.

However, the SqlDataSource is somewhat controversial, because it encourages us to place database logic in the markup portion of our page. Many developers prefer to use the *ObjectDataSource* instead, which gives similar data binding functionality but relies on a custom database component. When we use the *ObjectDataSource*, it's up to us to create the database component and write the back-end ADO.NET code.

• **LINQ to Entities:** With LINQ to Entities, we generate a data model with the design support in Visual Studio. The appropriate database logic is generated automatically. LINQ to Entities supports *updates, generates secure and well-written SQL statements*, and provides wide-ranging customizability. LINQ to Entites is also the preferred successor to the simpler LINQ to SQL model, which ASP.NET developers have used in the past.
ASP.NET developers will need to write some ADO.NET code, even if it's only to optimize a performance-sensitive task or to perform a specific operation that wouldn't otherwise be possible. Also, every professional ASP.NET developer needs to understand how the ADO.NET plumbing works in order to evaluate when it's required and when another approach is just as effective.

5.1.6.2 The ADO.NET Architecture

ADO.NET uses a multilayered architecture that revolves around a few key concepts, such as *Connection*, *Command*, and *DataSet* objects. One of the key differences between ADO.NET and some other database technologies is how it deals with the challenge of different data sources. In many previous database technologies, such as classic ADO, programmers use a generic set of objects no matter what the underlying data source is. For example, if we want to retrieve a record from an Oracle database using ADO code, we use the same Connection class we would use to tackle the task with SQL Server. This isn't the case in ADO.NET, which uses a data provider model.

5.1.6.3 ADO.NET Data Providers

A data provider is a set of ADO.NET classes that allows us to access a specific database, execute SQL commands, and retrieve data. Essentially, a *data provider is a bridge between our application and a data source*.
The classes that make up a data provider include the following:
• *Connection*: Use this object to establish a connection to a data source.
• *Command*: Use this object to execute SQL commands and stored procedures.
• *DataReader*: This object provides fast read-only, forward-only access to the data retrieved from a query.
• *DataAdapter*: This object performs two tasks. First, *we can use it to fill a DataSet* (a disconnected collection of tables and relationships) *with information extracted from a data source*. Second, *we can use it to apply changes to a data source, according to the modifications we've made in a DataSet.*
ADO.NET doesn't include generic data provider objects. Instead, it includes different data providers specifically designed for different types of data sources. Each data provider has a specific implementation of the *Connection, Command, DataReader, and DataAdapter* classes that's optimized for a specific RDBMS (relational database management system). For example, if we need to create a connection to a SQL Server database, we'll use a connection class named SqlConnection.

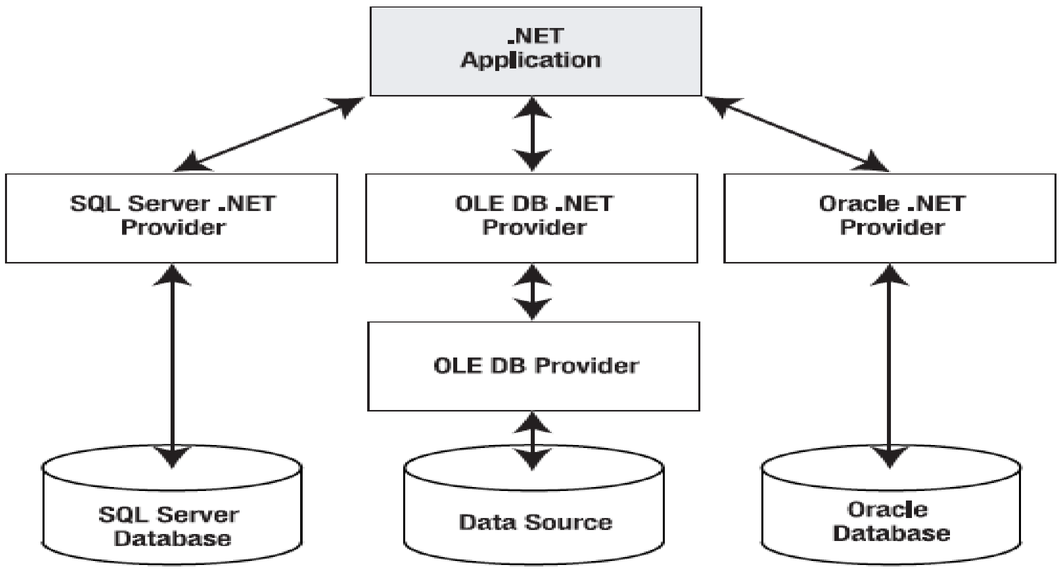


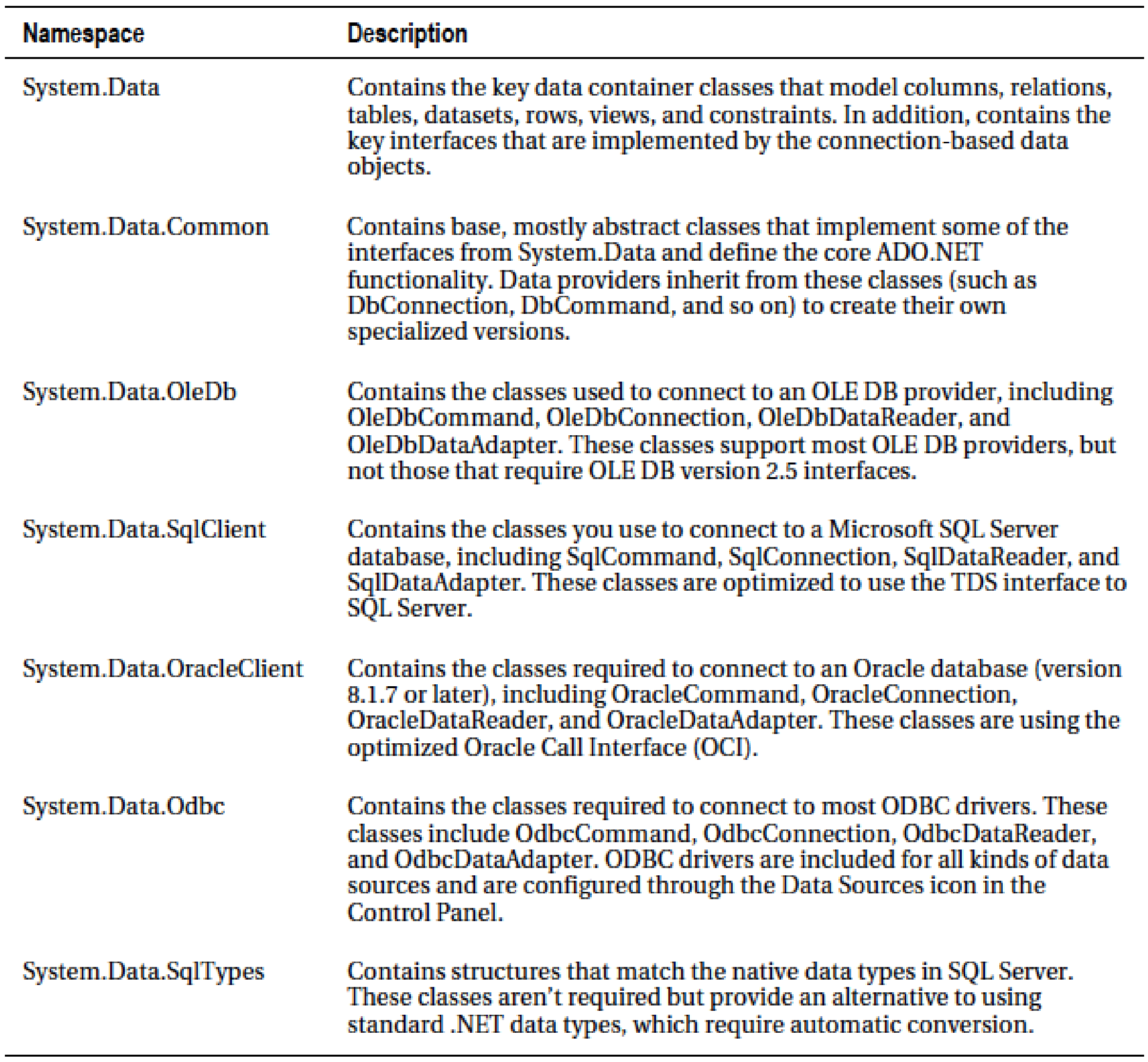
Figure 18: ADO.Net Architecture

5.1.6.4 Fundamental ADO.NET Classes

ADO.NET has two types of objects: connection-based and content-based.

Connection-based objects: These are the data provider objects such as *Connection, Command, DataReader, and DataAdapter*. They allow us to *connect to a database, execute SQL statements, move through a read-only result set, and fill a DataSet*.

Content-based objects: These objects are really just "packages" for data. They include the *DataSet, DataColumn, DataRow, DataRelation, and several others*. They are completely independent of the type of data source and are found in the System.Data namespace.



Namespace	Description
System.Data	Contains the key data container classes that model columns, relations, tables, datasets, rows, views, and constraints. In addition, contains the key interfaces that are implemented by the connection-based data objects.
System.Data.Common	Contains base, mostly abstract classes that implement some of the interfaces from System.Data and define the core ADO.NET functionality. Data providers inherit from these classes (such as DbConnection, DbCommand, and so on) to create their own specialized versions.
System.Data.OleDb	Contains the classes used to connect to an OLE DB provider, including OleDbCommand, OleDbConnection, OleDbDataReader, and OleDbDataAdapter. These classes support most OLE DB providers, but not those that require OLE DB version 2.5 interfaces.
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database, including SqlCommand, SqlConnection, SqlDataReader, and SqlDataAdapter. These classes are optimized to use the TDS interface to SQL Server.
System.Data.OracleClient	Contains the classes required to connect to an Oracle database (version 8.1.7 or later), including OracleCommand, OracleConnection, OracleDataReader, and OracleDataAdapter. These classes are using the optimized Oracle Call Interface (OCI).
System.Data.Odbc	Contains the classes required to connect to most ODBC drivers. These classes include OdbcCommand, OdbcConnection, OdbcDataReader, and OdbcDataAdapter. ODBC drivers are included for all kinds of data sources and are configured through the Data Sources icon in the Control Panel.
System.Data.SqlTypes	Contains structures that match the native data types in SQL Server. These classes aren't required but provide an alternative to using standard .NET data types, which require automatic conversion.

Table 16: The ADO.Net Namespace

5.1.7 THE CONNECTION CLASS

The Connection class allows us to establish a connection to the data source that we want to interact with.

5.1.7.1 Connection Strings

When we create a Connection object, we need to supply a connection string. The connection string is a series of name/value settings separated by semicolons (;). The order of these settings is unimportant, as is the capitalization.

Although connection strings vary based on the RDBMS and provider we are using, a few pieces of information are almost always required:

The server where the database is located: In the examples, the database server is always located on the same computer as the ASP.NET application, so the loopback alias localhost is used instead of a computer name.

The database we want to use: Most of the examples we are use the Northwind database, which is installed with older versions of SQL Server

How the database should authenticate: The Oracle and SQL Server providers give us the choice of supplying authentication credentials or logging in as the current user. The latter choice is usually best, because we don't need to place password information in our code or

configuration files.
For example, here’s the connection string we would use to connect to the Northwind database on the current computer using integrated security (which uses the currently logged-in Windows user to
access the database):
string connectionString = "Data Source=localhost; Initial Catalog=Northwind;" +
"Integrated Security=SSPI";
If integrated security isn’t supported, the connection must indicate a valid user and password combination.
string connectionString = "Data Source=localhost; Initial Catalog=Northwind;" + "user id=sa; password=opensesame";
The <connectionStrings> section of the web.config file is a handy place to store the connection strings.
Here’s an example:

```
<configuration>
<connectionStrings>
<add name="DBConnectionString" connectionString=
"Data Source=localhost; Initial Catalog=Northwind; Integrated Security=SSPI"/>
</connectionStrings>
...
</configuration>
```

We can then retrieve our connection string by name from the
WebConfigurationManager.ConnectionStrings collection

5.1.7.2 Testing a Connection

Once we’ve chosen our connection string, managing the connection is easy—we simply use the Open() and Close() methods.
We can use the following code in the *Page.Load* event handler to test a connection and write its status to a label. To use this code as written, we must import the
`System.Data.SqlClient` namespace.

```
// Create the Connection object.
string connectionString =
WebConfigurationManager.ConnectionStrings["DBConnectionString"].Conn
ectionString;
SqlConnection con = new SqlConnection(connectionString);
try
{
// Try to open the connection.
con.Open();
lblInfo.Text = "<b>Server Version:</b> " + con.ServerVersion;
lblInfo.Text += "<br /><b>Connection Is:</b> " +
con.State.ToString();
}
catch (Exception err)
{
// Handle an error by displaying the information.
lblInfo.Text = "Error reading the database. " + err.Message;
}
finally
{
// Either way, make sure the connection is properly closed.
// Even if the connection wasn't opened successfully,
// calling Close() won't cause an error.
con.Close();
lblInfo.Text += "<br /><b>Now Connection Is:</b> " +
con.State.ToString();
}
```

5.1.8 THE COMMAND AND DATAREADER CLASSES

The Command class allows us to execute any type of SQL statement. Although we can use a Command class to perform data definition tasks (such as creating and altering databases, tables, and indexes), we're much more likely to perform data manipulation tasks (such as retrieving and updating the records in a table).

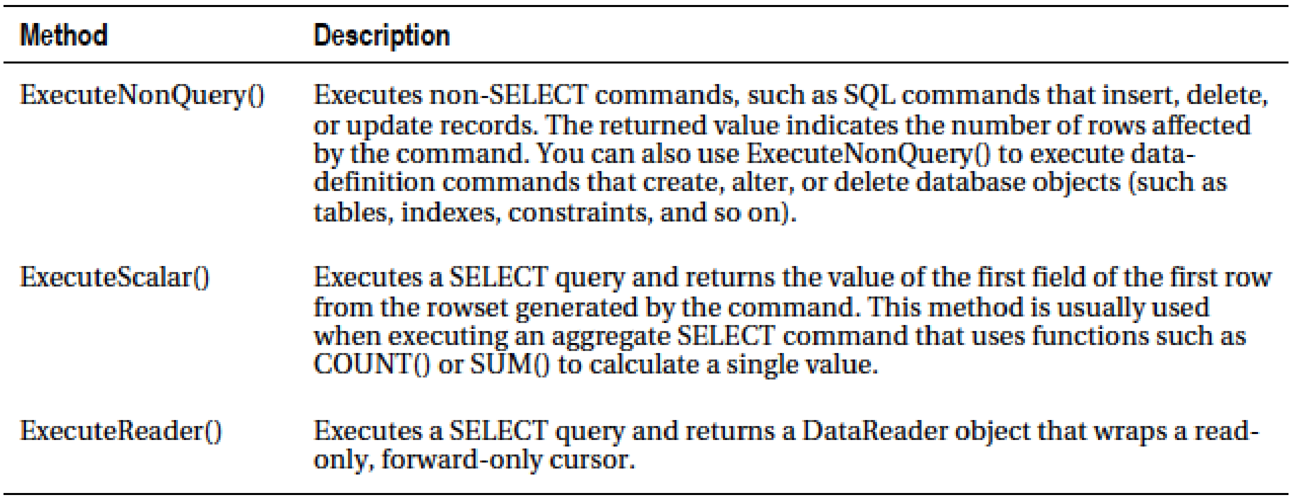
5.1.8.1 Command Basics

Before we can use a command, we need to choose the *command type*, set the *command text*, and *bind the command to a connection*. We can perform this work by setting the corresponding properties (CommandType, CommandText, and Connection), or we can pass the information we need as constructor arguments.
The command text can be a *SQL statement*, a *stored procedure*, or the *name of a table*. It all depends on the type of command we're using. Three types of commands exist.

Value	Description
CommandType.Text	The command will execute a direct SQL statement. The SQL statement is provided in the CommandText property. This is the default value.
CommandType.StoredProcedure	The command will execute a stored procedure in the data source. The CommandText property provides the name of the stored procedure.
CommandType.TableDirect	The command will query all the records in the table. The CommandText is the name of the table from which the command will retrieve the records. (This option is included for backward compatibility with certain OLE DB drivers only. It is not supported by the SQL Server data provider, and it won't perform as well as a carefully targeted query.)

Table 17: Values for the CommandType Enumeration

For example, here's how we would create a Command object that represents a query:
SqlCommand cmd = new SqlCommand();
cmd.Connection = con;
cmd.CommandType = CommandType.Text;
*cmd.CommandText = "SELECT * FROM Employees";*
And here's a more efficient way using one of the Command constructors. Note that we don't need to specify the CommandType, because CommandType.Text is the default.
*SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", con);*
Alternatively, to use a stored procedure,
SqlCommand cmd = new SqlCommand("GetEmployees", con);
cmd.CommandType = CommandType.StoredProcedure;
These examples simply define a Command object; they don't actually execute it. The Command object provides three methods that we can use to perform the command, depending on whether we want to retrieve a full result set, retrieve a single value, or just execute a nonquery command.



Method	Description
ExecuteNonQuery()	Executes non-SELECT commands, such as SQL commands that insert, delete, or update records. The returned value indicates the number of rows affected by the command. You can also use ExecuteNonQuery() to execute data-definition commands that create, alter, or delete database objects (such as tables, indexes, constraints, and so on).
ExecuteScalar()	Executes a SELECT query and returns the value of the first field of the first row from the rowset generated by the command. This method is usually used when executing an aggregate SELECT command that uses functions such as COUNT() or SUM() to calculate a single value.
ExecuteReader()	Executes a SELECT query and returns a DataReader object that wraps a read-only, forward-only cursor.

Table 18: Command Methods

5.1.8.2 *The DataReader Class*

A *DataReader* allows us to read the data returned by a SELECT command one record at a time, in a forward-only, read-only stream. This is sometimes called a firehose cursor. Using a DataReader is the simplest way to get to our data, but it lacks the sorting and relational abilities of the disconnected DataSet.

Methods	Description
Read()	Advances the row cursor to the next row in the stream. This method must also be called before reading the first row of data. The Read() method returns true if there’s another row to be read, or false if it’s on the last row.
GetValue()	Returns the value stored in the field with the specified index, within the currently selected row.
GetValues()	Saves the values of the current row into an array. The number of fields that are saved depends on the size of the array we pass to this method.
GetInt32(),GetChar(), GtDateTime(), GetXxx()	These methods return the value of the field with the specified index in the current row, with the data type specified in the method name.
NextResult()	If the command that generated the DataReader returned more than one rowset, this method moves the pointer to the next rowset.
Close()	Closes the reader.

Table 19: DataReader Class Methods

Example: Filling the ListBox

To start, the connection string is defined as a private variable for the page class and retrieved from the connection string:

private string connectionString =
WebConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;

The list box is filled when the Page.Load event occurs. Because the list box is set to persist its view
state information, this information needs to be retrieved only once—the first time the page is displayed.
It will be ignored on all postbacks. Here’s the code that fills the list from the database:

```
protected void Page_Load(Object sender, EventArgs e)
{
if (!this.IsPostBack)
        {
                FillAuthorList();
        }
}
private void FillAuthorList()
{
        lstAuthor.Items.Clear();
        // Define the Select statement.
        // Three pieces of information are needed: the unique id
        // and the first and last name.
        string selectSQL = "SELECT au_lname, au_fname, au_id FROM
Authors";
// Define the ADO.NET objects.
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand(selectSQL, con);
        SqlDataReader reader;
        // Try to open database and read information.
try
{
        con.Open();
        reader = cmd.ExecuteReader();
        // For each item, add the author name to the displayed
        // list box text, and store the unique ID in the Value
property.
        while (reader.Read())
        {
                ListItem newItem = new ListItem();
                newItem.Text = reader["au_lname"] + ", " +
        reader["au_fname"];
                newItem.Value = reader["au_id"].ToString();
                lstAuthor.Items.Add(newItem);
        }
        reader.Close();
}
        catch (Exception err)
        {
                lblResults.Text = "Error reading list of names. ";
                lblResults.Text += err.Message;
        }
                finally
        {
                con.Close();
        }
}
```

6 Chapter 6: Java Framework

- The Java programming language was originally created in 1995 by James Gosling from Sun Microsystems (currently a subsidiary of Oracle Corporation).
- The goal was to provide a simpler and platform-independent alternative to C++.
- Java is a general-purpose programming language that's used in all industries for almost any type of application.

6.1 The Life Cycle of a Java Program

- Java requires the source code of our program to be compiled first.
- It gets converted to either machine-specific code or a bytecode that is understood by some run-time engine or a virtual machine.
- Not only will the program be checked for syntax errors by a Java compiler, but also some other libraries of Java code can be added (linked) to our program after the compilation is complete (deployment stage).
- Technically we can write the source code of our Java program in any plain text editor that we prefer (Notepad, TextEdit, vi, etc.), but to compile our program we'll need additional tools and code libraries that are included in the Java Development Kit (JDK).

6.2 Java Runtime Environment (JRE)

Java Runtime Environment (JRE) contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If we want to run any java program, we need to have JRE installed in the system.

6.3 Java Development Kit (JDK)

Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc. Compiler converts java code into *byte code*. Java application launcher opens a JRE, loads the class, and invokes its main method. We need JDK, if at all we want to write our own programs, and to compile the program. For running java programs, JRE is sufficient. JRE is targeted for execution of Java files i.e. *JRE = JVM + Java Packages Classes (like util, math, lang, awt,swing etc)+runtime libraries*. JDK is mainly targeted for java development. i.e. we can create a Java file (with the help of Java packages), compile a Java file and run a java file.

6.4 Java Virtual Machine (JVM)

The Java Virtual Machine provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run. But, note that JVM itself not a platform independent. It only helps Java to be executed on the platform-independent way. When JVM has to interpret the byte codes to machine language, then it has to use some native or operating system specific language to interact with the system. One has to be very clear on platform independent concept. Even there are many JVMs written on Java, however they too have little bit of code specific to the operating systems.

As we all aware when we compile a Java file, output is not an 'exe' but it's a '.class' file. '.class' file consists of Java byte codes which are understandable by JVM. Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc.JVM is platform dependent.

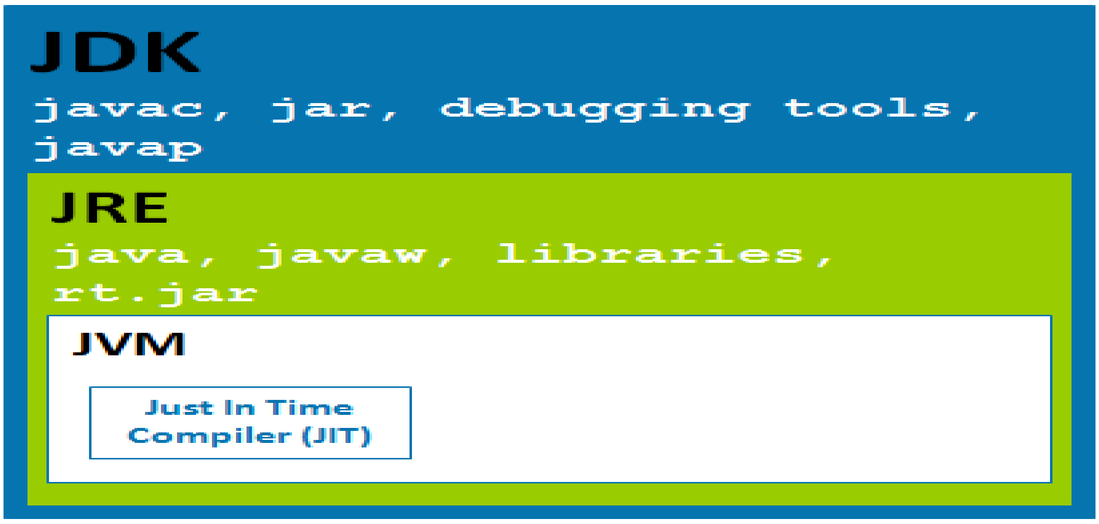

JDK
javac, jar, debugging tools, javap
JRE
java, javaw, libraries, rt.jar
JVM
Just In Time Compiler (JIT)

Figure 19: JVM, JRE and JDK

The JVM is called "virtual" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and operating system is a cornerstone of the *write-once run-anywhere* value of Java programs.
There are different JVM implementations are there. These may differ in things like performance, reliability, speed, etc. These implementations will differ in those areas where Java specification doesn't mention how to implement the features, like how the garbage collection process works is JVM dependent, Java spec doesn't define any specific way to do this.

6.5 Difference between JRE, JDK and JVM

In short here are few differences between JRE, JDK and JVM:

1) JRE and JDK come as installer while JVM are bundled with them.
2) JRE only contain environment to execute java program but doesn't contain other tool for compiling java program.
3) JVM comes along with both JDK and JRE and created when we execute Java program by giving "java" command.

6.6 Just in Time Compiler (JIT)

Initially Java has been accused of poor performance because it's both compiles and interprets instruction. Since compilation or Java file to class file is independent of execution of Java program. Here compilation word is used for byte code to machine instruction translation. JIT are advanced part of Java Virtual machine which optimize byte code to machine instruction conversion part by compiling similar byte codes at same time and thus reducing overall execution time. JIT is part of Java Virtual Machine and also performs several other optimizations such as in-lining function.

7 Chapter 7: Java Exception Handling

7.1 Exception

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually-typically through the use of error codes, and so on. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world. An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the
- JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. Based on these, we have three categories of Exceptions.

Checked exceptions: A checked exception is an exception that occurs at the compile time, these are also called as *compile time exceptions*. These exceptions cannot simply be ignored at the time of compilation; the programmer should take care of (handle) these exceptions.

Unchecked exceptions: An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in our code because we can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

7.1.1 Error and Exception

Both Error and Exception are derived from *java.lang.Throwable* in Java but main difference between **Error** and **Exception** is kind of error they represent. *java.lang.Error* represent errors which are generally can not be handled and usually refer catastrophic failure e.g. *running out of System resources*, some examples of Error in Java are *java.lang.OutOfMemoryError* or *Java.lang.NoClassDefFoundError* and *java.lang.UnSupportedClassVersionError*. On the other hand *java.lang.Exception* represent errors which can be catch and dealt e.g. IOException which comes while performing I/O operations i.e. reading files and directories.

7.1.2 Exception-Handling Fundamentals

A Java *exception* is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an *object* representing that exception is created and thrown in the method that caused the error. That method may choose

to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the *Java run-time* system, or they can be manually generated by our code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
Java exception handling is managed via five keywords: ***try, catch, throw, throws, and finally***. Briefly, here is how they work.

- Program statements that we want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.
- The code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword ***throw***.
- Any exception that is thrown out of a method must be specified as such by a ***throws*** clause.
- Any code that absolutely must be executed after a try block completes is put in a ***finally*** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

7.1.3 Exception Types

All exception types are subclasses of the built-in class ***Throwable***. Thus, *Throwable* is at the top of the exception class hierarchy. Immediately below *Throwable* are two subclasses that partition exceptions into two distinct branches. One branch is headed by ***Exception***. This class is used for exceptional conditions that user programs should catch. This is also the class that we will subclass to create our own custom exception types. There is an important subclass of Exception, called *RuntimeException*. Exceptions of this type are automatically defined for the programs that we write and include things such as division by zero and invalid array indexing.

The other branch is topped by ***Error***, which defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type *Error are used by the Java run-time* system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

The Exception class has two main subclasses: ***IOException*** class and ***RuntimeException*** Class.

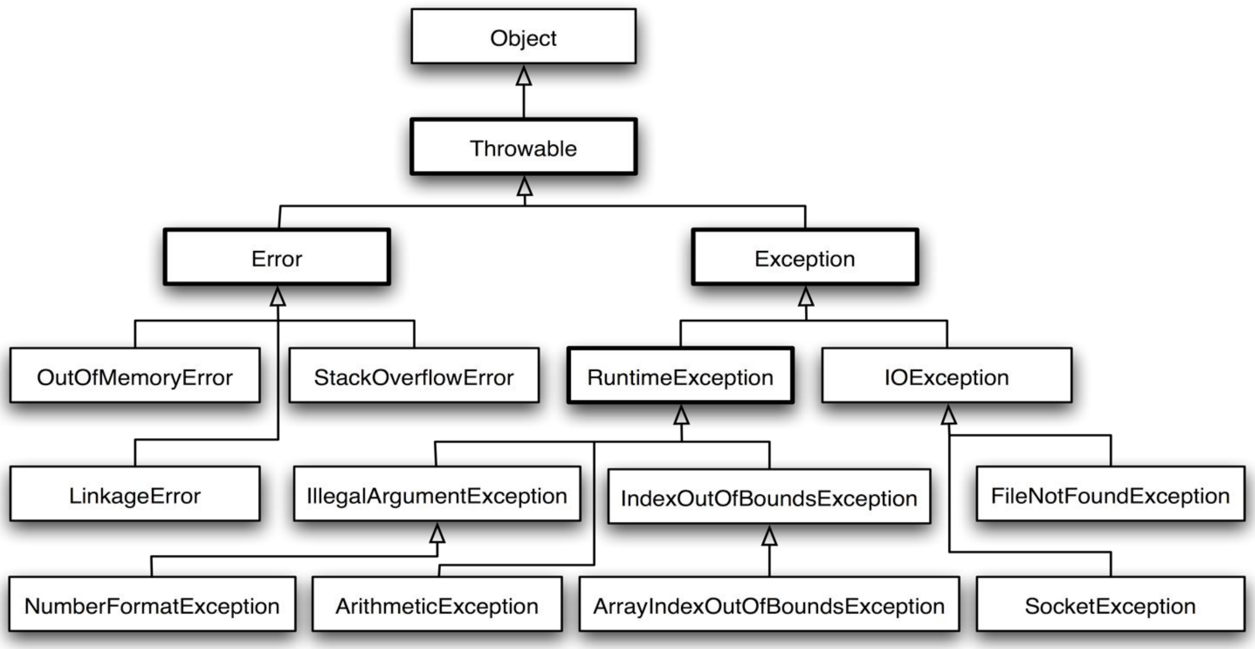


Figure 20: Java Exception Hierarchy

7.1.4 Built-in Exceptions

Java defines several exception classes inside the standard package **java.lang**.The most general of these exceptions are subclasses of the standard type ***RuntimeException***. Since *java.lang* is implicitly imported into all Java programs, most exceptions derived from ***RuntimeException*** are automatically available.Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java **Unchecked** *RuntimeException*.

SN	Exception	Description
1	ArithmeticException	Arithmetic error, such as divide-by-zero.
2	ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
3	ArrayStoreException	Assignment to an array element of an incompatible type.
4	ClassCastException	Invalid cast.
5	IllegalArgumentException	Illegal argument used to invoke a method.
6	IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
7	IllegalStateException	Environment or application is in incorrect state.
8	IllegalThreadStateException	Requested operation not compatible with the current thread state.
9	IndexOutOfBoundsException	Some type of index is out-of-bounds.
10	NegativeArraySizeException	Array created with a negative size.
11	NullPointerException	Invalid use of a null reference.
12	NumberFormatException	Invalid conversion of a string to a numeric format.
13	SecurityException	Attempt to violate security.
14	StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
15	UnsupportedOperationException	An unsupported operation was encountered.

Table 20: Java's Unchecked RuntimeException Subclasses Defined in java.lang

Following is the list of Java **Checked Exceptions** Defined in *java.lang*.

SN	Exception	Description
1	ClassNotFoundException	Class not found.
2	CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException	Access to a class is denied.
4	InstantiationException	Attempt to create an object of an abstract class or interface.
5	InterruptedException	One thread has been interrupted by another thread.
6	NoSuchFieldException	A requested field does not exist.
7	NoSuchMethodException	A requested method does not exist.

Table 21: Java's Checked Exceptions Defined in java.lang

7.1.5 Uncaught Exceptions

It is useful to see what happens when we don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to *divide by zero*, it constructs a new exception object and then throws this exception. This causes the execution of *Exc0* to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the default handler provided by the Java run-time system catches the exception. The default handler will ultimately process any exception that is not caught by our program. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

7.1.6 Using *try* and *catch*

Although the default exception handler provided by the *Java run-time system* is useful for debugging, we usually want to handle an exception our self. Doing so provides two benefits. First, *it allows us to fix the error*. Second, *it prevents the program from automatically terminating*. To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that we wish to catch. Following block of code shows that the *ArithmeticException* generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
} //End of Catch
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

7.1.7 Multiple *catch* Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
class MultiCatch {
      public static void main(String args[]) {
      try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
            } catch(ArithmeticException e) {
                  System.out.println("Divide by 0: " + e);
            } catch(ArrayIndexOutOfBoundsException e) {
                  System.out.println("Array index oob: " + e);
            }
      System.out.println("After try/catch blocks.");
      }
}
```

This program will cause a division-by-zero exception if it is started with no commandline arguments, since a will equal zero. It will survive the division if we provide a command-line argument, setting **a** to something larger than zero. But it will cause an *ArrayIndexOutOfBoundsException*, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When using multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. *Further, in Java, unreachable code is an error*. For example, consider the following program:

```
class SuperSubCatch {
      public static void main(String args[]) {
            try {
                  int a = 0;
```

```
                int b = 42 / a;
          }catch(Exception e) {
                System.out.println("Generic Exception catch.");
          }
          /*    This    catch    is    never    reached    because
      ArithmeticException is a subclass of Exception. */
          catch(ArithmeticException e) { // ERROR - unreachable
                System.out.println("This is never reached.");
          }
      }
}
```

If we try to compile this program, we will receive an error message stating that the *second catch statement is unreachable because the exception has already been caught*. Since *ArithmeticException* is a subclass of Exception, the first catch statement will handle all Exception-based errors, including *ArithmeticException*. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

7.1.8 Nested *try* Statements

The *try* statement can be nested. That is, a *try* statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

```
// An example of nested try statements.
class NestTry {
      public static void main(String args[]) {
            try {
                  int a = args.length;
/* If no command-line args are present,the following statement will
generate a divide-by-zero exception. */
                  int b = 42 / a;
                  System.out.println("a = " + a);
                  try { // nested try block
/* If one command-line arg is used,then a divide-by-zero exception
will be generated by the following code. */
                        if(a==1)
                              a = a/(a-a); // division by zero
/* If two command-line args are used,then generate an out-of-bounds
exception. */
                        if(a==2) {
                              int c[] = { 1 };
                              c[42]=99;//generate    an    out-of-bounds
                        exception
                        }
                  } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array    index    out-of-
                  bounds:"+ e);
                  }
                  } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
            }
      }
}
```

The program works as follows. When we execute the program with no command-line arguments, a *divide-by-zero exception* is generated by the outer try block. Execution of the program with one command-line argument generates a ***divide-by-zero*** exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

7.1.9 throw

However, it is possible for our program to throw an exception explicitly, using the ***throw*** statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, *ThrowableInstance* must be an object of type *Throwable* or a subclass of *Throwable*. Primitive types, such as *int* or *char*, as well as *non-Throwable* classes, such as String and Object, cannot be used as exceptions. There are two ways that we can obtain a *Throwable* object: using a parameter in a *catch* clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
// Demonstrate throw.
class ThrowDemo {
      static void demoproc() {
            try {
                  throw new NullPointerException("demo");
                  } catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                  }
            }
      public static void main(String args[]) {
      try {
            demoproc();
            } catch(NullPointerException e) {
                  System.out.println("Recaught: "+e);
            }
      }
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another *exceptionhandling* context and immediately throws a new instance of *NullPointerException*, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

7.1.10 throws

If a method does not handle a checked exception, the method must declare it using the ***throws*** keyword. The throws keyword appears at the end of a method's signature. If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a ***throws*** clause in the method's declaration. A ***throws*** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type ***Error or RuntimeException***, or any of their subclasses. All other exceptions that a method can throw must be declared in the ***throws*** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
} //End of catch
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

7.1.11 finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

The ***finally*** block follows a ***try*** block or a ***catch*** block. A *finally* block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows us to run any cleanup type statements that we want to execute, no matter what happens in the protected code. The *finally* block will execute whether or not an exception is thrown. If an exception is thrown, the *finally* block will execute even if no *catch* statement matches the exception. Any time a method is about to return to the caller from inside a *try/catch* block, via an uncaught exception or an explicit return statement, the *finally* clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The *finally* clause is optional. However, each try statement requires at least one catch or a finally clause.

Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

// Demonstrate finally.

```
class FinallyDemo {
      // Through an exception out of the method.
      static void procA() {
            try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
            } finally {
            System.out.println("procA's finally");
            }
            }
            // Return from within a try block.
            static void procB() {
                  try {
                  System.out.println("inside procB");
                  return;
                  } finally {
                  System.out.println("procB's finally");
                  }
            }
            // Execute a try block normally.
            static void procC() {
                  try {
                  System.out.println("inside procC");
                  } finally {
                  }
            }
            public static void main(String args[]) {
            try {
                  procA();
                  } catch (Exception e) {
                  System.out.println("Exception caught");
            }
            procB();
            procC();
      }
}
```

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The *finally* clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed.
Here is the output generated by the preceding program:

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

7.1.12 User-defined Exceptions

We can create our own exceptions in Java. Following is the key points to understand before we write our own exception classes:

- All exceptions must be a child of ***Throwable***.
- If we want to write a checked exception that is automatically enforced by the Handle or
  Declare Rule, we need to extend the **Exception** class.
- If we want to write a *runtime exception*, we need to extend the ***RuntimeException*** class.

Syntax

```
class MyException extends Exception{
    //class definition
}
```

The following example declares a new subclass of **Exception** and then uses that subclass to signal an **error** condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}

class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

This example defines a subclass of Exception called **MyException**. This subclass is quite simple: it has only a constructor plus an overloaded **toString( )** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10. The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 10) and an illegal one to show both paths through the code.

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

7.2 Debugging

7.2.1 Loop Bug

The two most common kinds of loop errors are unintended *infinite loops* and *off-by-one errors*.

Infinite Loop: A while loop, do-while loop, or for loop does not terminate as long as the controlling Boolean expression evaluates to true . This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable eventually is changed in a way that makes the Boolean expression false and therefore

terminates the loop. However, if we make a mistake and write our program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an **infinite loop**.
off-by-one error: The loop repeats the loop body *one too many or one too few times*. These sorts of errors can result from carelessness in designing a controlling Boolean expression. For example, if we use less-than when we should use less-than-or-equal, this can easily make our loop iterate the body the wrong number of times.

Use of == to test for equality in the controlling Boolean expression of a loop can often lead to an off-by-one error or an infinite loop. This sort of equality testing can work satisfactorily for integers and characters, but is not reliable for floating-point numbers. This is because the floating-point numbers are approximate quantities, and == tests for exact equality. The result of such a test is unpredictable. When comparing floating-point numbers, always use something involving less-than or greater-than, such as <= ; do not use == or != . Using == or != to test floating-point numbers can produce an off-by-one
error or an unintended infinite loop or even some other type of error. Even when using integer variables, it is best to avoid using == and != and to instead use something involving less-than or greater-than.

7.2.2 Tracing Variables
One good way to discover errors in a loop or any kind of code is to trace some key variables. Tracing variables means watching the variables change value while the program is running. Most programs do not output each variable's value every time the variable changes, but being able to see all of these variable changes can help us to debug your program.

7.2.3 General Debugging Techniques
Tracing errors can sometimes be a difficult and time-consuming task. It is not uncommon to spend more time debugging a piece of code than it took to write the code in the first place. If we are having difficulties finding the source of our errors, then there are some general debugging techniques to consider. If the program is giving incorrect output values, then we should examine the source code, different test cases using a range of input and output values, and the logic behind the algorithm itself.

Determining the precise cause and location of a bug is one of the first steps in fixing the error. Examine the input and output behavior for different test cases to try to localize the error. A related technique is to trace variables to show what code the program is executing and what values are contained in key variables. We might also focus on code that has recently changed or code that has had errors before.

Finally, we can also try removing code. If we comment out blocks of code and the error still remains, then the culprit is in the uncommented code. The process can be repeated until the location of the error can be pinpointed. The /* and */ notation is particularly useful to comment out large blocks of code. After the error has been fixed, it is easy to remove the comments and reactivate the code.

The first mistakes we should look for are common errors that are easy to make. Examples of common
errors include off-by-one errors, comparing floating-point types with == , adding extra semicolons that terminate a loop early, or using == to compare strings for equality.

8 Chapter 8: Applets and Application

8.1 Applet Fundamentals

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity. Applets differ from console-based applications in several key areas.

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
      public void paint(Graphics g) {
      g.drawString("A Simple Applet", 20, 20);
      }
}
```

1. This applet begins with two ***import*** statements. The first imports the *Abstract Window Toolkit* (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface.
2. The second ***import*** statement imports the ***applet*** package, which contains the class ***Applet***. Every applet that we create must be a subclass of ***Applet***.
3. The next line in the program declares the class ***SimpleApplet***. This class must be declared as ***public***, because it will be accessed by code that is outside the program.
4. Inside ***SimpleApplet***, ***paint( )*** is declared. This method is defined by the AWT and must be overridden by the *applet*. paint( ) is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, *the window in which the applet is running can be overwritten by another window and then uncovered*. Or, *the applet window can be minimized and then restored*. ***paint( )*** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
5. Inside paint( ) is a call to ***drawString( ),*** which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:
   void drawString(String message, int x, int y)
6. Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to ***drawString( )*** in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.
7. Notice that the applet ***does not have a main( )*** method. Unlike Java programs, applets do not begin execution at main( ). In fact, most applets don’t even have a main( ) method. Instead, an applet begins execution ***when the name of its class is passed to an applet viewer*** or to a network browser.
8. After we enter the source code for ***SimpleApplet***, compile in the same way that we have been compiling programs. However, running ***SimpleApplet*** involves a different process. In fact, there are two ways in which you can run an applet:
   - Executing the applet within a Java-compatible web browser.
   - Using an applet viewer, such as the standard tool, ***appletviewer***. An applet viewer executes our applet in a window. This is generally the fastest and easiest way to test our applet.

To execute an applet in a web browser, we need to write a short HTML text file that contains a tag that loads the applet. Currently, Sun recommends using the APPLET tag for this purpose. Here is the HTML file that executes ***SimpleApplet***:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. After the creation of this file, we can execute our browser and then load this file, which causes ***SimpleApplet*** to be executed.

To execute ***SimpleApplet*** with an applet viewer, we may also execute the HTML file shown earlier. For example, if the preceding HTML file is called ***RunApp.html***, then the following command line will run SimpleApplet:

```
      C:\>appletviewer RunApp.html
```

However, a more convenient method exists that we can use to speed up testing. Simply include a comment at the head of Java source code file that contains the APPLET tag. By doing so, our code is documented with a prototype of the necessary HTML statements, and we can test our compiled applet merely by starting the applet viewer with our Java source code file. The ***SimpleApplet*** source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
      public void paint(Graphics g) {
      g.drawString("A Simple Applet", 20, 20);
      }
}
```

Key Points

1. Applets do not need a main( ) method.
2. Applets must be run under an applet viewer or a Java-compatible browser.
3. User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.
4. All applets are subclasses (either directly or indirectly) of Applet.
5. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.
6. Output is handled with various AWT methods, such as ***drawString()***,which outputs a string to a specified X,Y location, not by ***system.out.println()***.
7. A Java-enabled web browser will execute the applet when it encounters the APPLET tag within the HTML file.

8.2 Two Types of Applet

There are two varieties of applets.

1. The first type of applets uses the ***Abstract Window Toolkit (AWT)*** to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.
2. The second type of applets is those based on the Swing class ***JApplet***. Swing applets use the Swing classes to provide the GUI. *Swing offers a richer and often easier-to-use user interface than does the AWT*. Thus, Swing-based applets are now the most

popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid. Because ***JApplet*** inherits ***Applet***, all the features of ***Applet*** are also available in ***JApplet***.

8.3 The Applet Class

The ***Applet*** class defines the methods shown in following table. ***Applet*** provides all necessary support for applet execution, such as *starting and stopping*. It also provides methods that *load and display images*, and *methods that load and play audio clips*. ***Applet*** extends the ***AWT class Panel***. In turn, ***Panel*** extends ***Container***, which extends ***Component***. These classes provide support for Java's window-based, graphical interface. Thus, ***Applet*** provides all of the necessary support for window-based activities.

Methods	**Description**
void destroy( )	Called by the browser just before an applet is terminated.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
Locale getLocale( )	Returns a Locale object that is used by various locale sensitive classes and methods.
String getParameter(String paramName)	Returns the parameter associated with ***paramName.null*** is returned if the specified parameter is not found.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns true if the applet has been started. It returns false if the applet has been stopped.
void play(URL url)	If an audio clip is found at the location specified by url, the clip is played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after init( ) when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start( ).
void showStatus(String str)	Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.

And many more.. (refer to text book)

Table 22: The Methods Defined by Applet

8.4 Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs. First, applets are event driven. It is important to understand in a general way that the event-driven architecture impacts the design of an applet.
An applet resembles a set of *interrupt service routines*. Here is how the process works. An applet waits until an ***event*** occurs. The ***run-time system*** notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, *it must perform specific actions in response to events and then return control to the run-time system*. In those situations in which applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), we must start an additional thread of execution.

Second, the user initiates interaction with an applet. These interactions are sent to the applet *as events to which the applet must respond*. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a *keypress* event is generated. ***Applets*** can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

8.5 Applet Life Cycle Methods

There are five applet methods that are called by the applet container from the time the applet is loaded into the browser to the time it's terminated by the browser. These methods correspond to various aspects of an applet's life cycle.

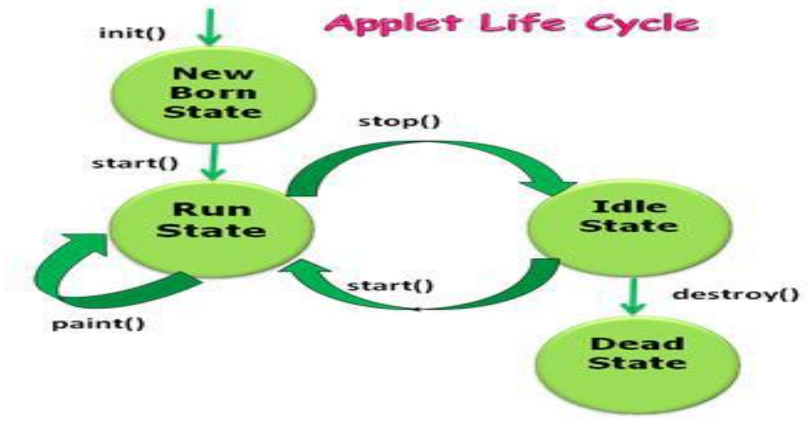


Figure 21: Applet Life Cycle

Methods	Description
public void init()	Called once by the ***applet container*** when an applet is loaded for execution. This method initializes an applet. Typical actions performed here are initializing fields, creating GUI components, loading sounds to play, loading images to display and creating threads.
public void start()	Called by the ***applet container*** after method ***init*** completes execution. In addition, if the user browses to another website and later returns to the applet's HTML page, method **start** is called again. The method performs any tasks that ***must be completed*** when the applet is loaded for the first time and that must be performed every time the applet's

	HTML page is revisited. Actions performed here might include starting an animation or starting other threads of execution.
public void paint(Graphics g)	Called by the ***applet container*** after methods ***init*** and ***start***. Method paint is also called when the applet needs to be repainted. For example, if the user covers the applet with another open window on the screen and later uncovers it, the ***paint*** method is called. Typical actions performed here involve drawing with the Graphics object **g** that's passed to the paint method by the applet container.
public void stop()	The ***applet container*** calls this method when the user leaves the applet's web page by browsing to another web page. Since it's possible that the user might return to the web page containing the ***applet***, method ***stop*** performs tasks that might be required to suspend the applet's execution, so that the applet does not use computer processing time when it's not displayed on the screen. Typical actions performed here would stop the execution of animations and threads.
public void destroy()	The ***applet container*** calls this method when the applet is being removed from memory. This occurs when the user exits the browsing session by closing all the browser windows and may also occur at the browser's discretion when the user has browsed to other web pages. The method performs any tasks that are required to ***clean up*** resources allocated to the applet.

8.6 An Applet Skeleton

Four of these methods, **init( ), start( ), stop( ),** and **destroy( ),** apply to all applets and are defined by Applet. Default implementations for all of these methods are provided.

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
        // Called first.
        public void init() {
        // initialization
        }
/* Called second, after init(). Also called whenever the applet is restarted. */
        public void start() {
        // start or resume execution
        }
        // Called when the applet is stopped.
        public void stop() {
        // suspends execution
        }
        /* Called when applet is terminated. This is the last method executed. */
        public void destroy() {
        // perform shutdown activities
        }
        // Called when an applet's window must be restored.
        public void paint(Graphics g) {
        // redisplay contents of window
        }
}
```

8.7 The HTML APPLET Tag

The APPLET tag be used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional while ***code***, ***width*** and ***height*** attributes are required.

```
<APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels
HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

- **CODEBASE** is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file. The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
- **CODE** is a required attribute that gives the name of the file containing applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
- **ALT** The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.
- **NAME** is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use ***getApplet( )***, which is defined by the ***AppletContext*** interface.
- **WIDTH** and **HEIGHT** are required attributes that give the size (in pixels) of the applet display area.
- **ALIGN** is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.
- **VSPACE** and **HSPACE** are optional attributes. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE

attributes.

- **PARAM NAME** and **VALUE** The PARAM tag allows us to specify applet-specific arguments in an HTML page. Applets access their attributes with the ***getParameter( )*** method.

8.8 Passing Parameters to Applets

The **APPLET tag** in HTML allows us to pass parameters to our applet. To retrieve a parameter, we need to use the ***getParameter( )*** method. It returns the value of the specified parameter in the form of a ***String*** object. Thus, for ***numeric*** and ***boolean*** values, we will need to convert their string representations into their internal formats.

```
// Use Parameters
/*
<applet code="ParamDemo" width="400" height="100">
<param name="fontName" value="Times new Roman">
<param name="fontSize" value="15">
<param name="leading" value="3">
<param name="accountEnabled" value="true">
</applet>
*/

import java.awt.*;
import java.applet.*;

public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
      String param;
      fontName = getParameter("fontName");
      if(fontName == null)
      fontName = "Not Found";
      param = getParameter("fontSize");
      try {
            if(param != null) // if not found
            fontSize = Integer.parseInt(param);
            else
                  fontSize =0;
      } catch(NumberFormatException e) {
      fontSize = -1;
      }
      param = getParameter("leading");
      try {
      if(param != null) // if not found
      leading = Float.valueOf(param).floatValue();
      else
      leading = 0;
      } catch(NumberFormatException e) {
      leading = -1;
      }
      param = getParameter("accountEnabled");
      if(param != null)
      active = Boolean.valueOf(param).booleanValue();
      }
// Display parameters.
public void paint(Graphics g) {
      g.drawString("Font name: " + fontName, 0, 10);
      g.drawString("Font size: " + fontSize, 0, 26);
      g.drawString("Leading: " + leading, 0, 42);
      g.drawString("Account Active: " + active, 0, 58);
}
}
```

8.9 Applet Security Basics

Because applets are designed to be loaded from a remote site and then executed locally, security becomes vital. If a user enables Java in the browser, the browser will download all the applet code on the web page and execute it immediately. The user never gets a chance to confirm or to stop individual applets from running. For this reason, applets (unlike applications) are restricted in what they can do. The applet security manager throws a ***SecurityException*** whenever an applet attempts to violate one of the access rules. What can applets do on all platforms? They can show images and *play sounds, get keystrokes and mouse clicks from the user, and send user input back to the host* from which they were loaded. That is enough functionality to show facts and figures or to get user input for placing an order. The restricted execution environment for applets is often called the "sandbox." Applets playing in the "sandbox" cannot alter the user's system or spy on it.

In particular, when running in the sandbox:

- Applets can never run any local executable program.
- Applets cannot communicate with any host other than the server from which they were downloaded; that server is called the ***originating host***. This rule is often called "applets can only phone home." This protects applet users from applets that might try to spy on intranet resources.
- Applets cannot read from or write to the local computer's file system.
- Applets cannot find out any information about the local computer, except for the Java version used, the name and version of the operating system, and the characters used to separate files (for instance, / or \), paths (such as : or ;), and lines (such as *\n or \r\n*). In particular, applets cannot find out the user's name, e-mail address, and so on.
- All windows that an applet pops up carry a warning message.

All this is possible only because applets are interpreted by the Java Virtual Machine and not directly executed by the CPU on the user's computer. Because the interpreter checks all critical instructions and program areas, a hostile (or poorly written) applet will almost certainly not be able to crash the computer, overwrite system memory, or change the privileges granted by the operating system.

These restrictions are too strong for some situations. For example, on a corporate intranet, we can certainly imagine an applet wanting to access local files. To allow for different levels of security under different situations, we can use ***signed applets***. A signed applet carries with it a certificate that indicates the identity of the signer. Cryptographic techniques ensure that such a certificate cannot be forged. If we trust the signer, we can choose to give the applet additional rights.

The point is that if we trust the signer of the applet, we can tell the browser to give the applet more privileges. The ***configurable Java security model*** allows the continuum of privilege

levels we need. We can give completely trusted applets the same privilege levels as local applications. Programs from vendors that are known to be somewhat flaky can be given access to some, but not all, privileges. Unknown applets can be restricted to the sandbox.
To sum up, Java has three separate mechanisms for enforcing security:

1. Program code is interpreted by the Java Virtual Machine, not executed directly.
2. A security manager checks all sensitive operations in the Java runtime library.
3. Applets can be signed to identify their origin.

8.10 Applet vs Application

Features	**Application**	**Applet**
main() method	Present	Not present
Execution	Requires JRE	Requires a browser or applet viewer
Nature	Called as stand-alone application as application can be executed from command prompt	Requires some third party tool help like a browser to execute
Restrictions	Can access any data or software available on the system	cannot access any thing on the system except browser's services
Security	Does not require any security	Requires highest security for the system as they are untrusted

Table 23: Applet Vs Application:

8.11 Advantages of Applet

- Execution of applets is easy in a Web browser and does not require any installation or deployment procedure in real-time programming (where as servlets require).
- Writing and displaying (just opening in a browser) graphics and animations is easier than applications.
- In GUI development, constructor, size of frame, window closing code etc. are not required (but are required in applications).

8.12 Application Conversion to Applets

It is easy to convert a graphical Java application that is an application that uses the AWT and that we can start with the java program launcher into an applet that we can embed in a web page. Here are the specific steps for converting an application to an applet.
Make an HTML page with the appropriate tag to load the applet code.

- Supply a subclass of the ***JApplet*** class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the ***main*** method in the application. Do not construct a frame window for the application. Our application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the **init** method of the applet. We don't need to explicitly construct the applet object. The browser instantiates it for us and calls the ***init*** method.

- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to **setDefaultCloseOperation**. An applet cannot be closed; it terminates when the browser exits.
- If the application calls **setTitle**, eliminate the call to the method. Applets cannot have title bars. (We can, of course, title the web page itself, using the HTML title tag).
- Don't call **setVisible(true)**. The applet is displayed automatically.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within
the applet, we use the **drawImage** method found in the ***java.awt.Graphics*** class. Following is the example showing all the steps to show images:

```
import  java.applet.*;
import  java.awt.*;
import  java.net.*;
public class ImageDemo extends Applet
{
      private Image  image;
      private AppletContext context;
      public void  init()
      {
            context = this.getAppletContext();
            String  imageURL = this.getParameter("image");
            if(imageURL == null)
            {
                  imageURL = "java.jpg";
            }
            try
            {
             URL url = new  URL(this. getDocumentBase(),imageURL);
             image =  context.getImage(url);
            }catch(MalformedURLException  e)
            {
                  e. printStackTrace();
                  // Display in browser status bar
             context.showStatus("Could not load image!");
            }
      }
      public void  paint(Graphics  g)
      {
       context.showStatus("Displaying image");
       g.drawImage( image, 0, 0, 200, 84, null);
       g.drawString("www.appletdemos.com", 35, 100);
      }
}
```

Now, let us call this applet as follows:

```
<html>
      <title>The ImageDemo applet</title>
      <hr>
            <applet code="ImageDemo.class" width="300" height="200">
                  <param nam e="image" value="java.jpg">
            </applet>
      <hr>
</html>
```

Playing Audio

An applet can play an audio file represented by the **AudioClip** interface in the **java.applet** package. The **AudioClip** interface has three methods, including:

public void play: Plays the audio clip one time, from the beginning.
public void loop: Causes the audio clip to replay continually.
public void stop: Stops playing the audio clip.

To obtain an **AudioClip** object, we must invoke the **getAudioClip** method of the Applet class. The
getAudioClip method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.
Following is the example showing all the steps to play an audio:

```
import  java. applet.* ;
import  java. awt.* ;
import  java. net.* ;
public class AudioDemo extends Applet
{
      private AudioClip  clip;
      private AppletContext  context;
      public void  init()
      {
            context = this.getAppletContext();
            String  audioURL = this.getParameter("audio");
            if( audioURL == null)
            {
                  audioURL = "default.au";
            }
            try
            {
                  URL url=new URL(this.getDocumentBase(),audioURL);
                  clip=context.getAudioClip(url);
            }catch(MalformedURLException  e)
            {
                  e.printStackTrace();
                  context.showStatus("Could not load audio file!");
            }
      }
      public void  start()
      {
            if( clip != null)
            {
                  clip.loop();
            }
      }
      public void  stop()
      {
            if( clip != null)
            {
                  clip.stop();
            }
      }
}
```

Now, let us call this applet as follows:

```
<html>
      <title>The AudioDemo applet</title>
      <hr>
```

```
    <applet code="AudioDemo.class" width="0" height="0">
    <param nam e="audio" value="test.wav">
    </applet>
    <hr>
</html>
```

8.13 Converting a Java applet to an application

1. Build a Java **JApplet** (or Applet) in Eclipse or NetBeans. (Let's assume you name the class **AppletToApplication**).
2. Change the name of your applet's **init()** method to be the same as the name of your class (in this case: **AppletToApplication**). This is now the constructor method for the class.
3. Delete the word **void** in the header of your new **AppletToApplication** constructor, since a constructor has no return type.
4. Alter the class header so that it extends **Frame** rather than **Applet** (or **JApplet**) .
5. Create a new method called **main**. The header for this method will be:
   public static void main (String[] args)
   This method should create a Frame object as an instance of the class. So, if your class is named **AppletToApplication**, the main method should look like the following (where the size will be your original applet size):

```
public static void main(String[] args)
{
  AppletToApplication f = new AppletToApplication ();
  f.setSize(300,200);
  f.setVisible(true);
  f.setLayout(new FlowLayout());
}
```

6. Delete the **import** for the class **Applet** , since it is now redundant.
7. Add window methods (e.g., **windowClosing** to handle the event which is the user clicking on the close window button, and others). This also involves adding **implements WindowListener** and **this.addWindowListener(this)**; to the new constructor method you created in Step #2 above - in order to register the event handler.

```
public void windowClosing(WindowEvent e)
{
   dispose();
   System.exit(0);
}
public void windowOpened(WindowEvent e)
{ }
public void windowIconified(WindowEvent e)
{ }
public void windowClosed(WindowEvent e)
{ }
public void windowDeiconified(WindowEvent e)
{ }
public void windowActivated(WindowEvent e)
{ }
public void windowDeactivated(WindowEvent e)
{ }
```

8. Make sure that the program does not use any of the methods that are special to the Applet class – methods including **getAudioClip** , **getCodeBase** , **getDocumentBase** , and **getImage**.

Applet Program to perform addition of two User input numbers.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class UserInputDemo extends Applet implements ActionListener
{
    int a = 0, b = 0;
    String str1, str2, str;
    TextField T1;
    TextField T2;
    TextField T3;
    Button btnAdd;

 public void init() {
        // TODO start asynchronous download of heavy resources
        T1 = new TextField(10);
        T2 = new TextField(10);
        T3 = new TextField(10);
        btnAdd = new Button("Add");
        T1.setText("0");
        T2.setText("0");
        T3.setText("0");

        add(T1);
        add(T2);
        add(T3);
        add(btnAdd);
        btnAdd.addActionListener(this);
    }
   @Override
   public void actionPerformed(ActionEvent ae)
   {
         str1 = T1.getText();
         a = Integer.parseInt(str1);
         str2 = T2.getText();
         b = Integer.parseInt(str2);
         T3.setText("sum= "+(a+b));
    }
}
```

9 Chapter 9: Events, Handling Events and AWT/Swing

9.1 Event Handling Fundamental

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running. Each program then decides what, if anything, to do in response to these events. Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs. Any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Events are supported by a number of packages, including ***java.util***, ***java.awt***, and ***java.awt.event***.
Most events to which our program will respond are generated when the user interacts with a ***GUI-based program***. They are passed to our program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the ***mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.***
Events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

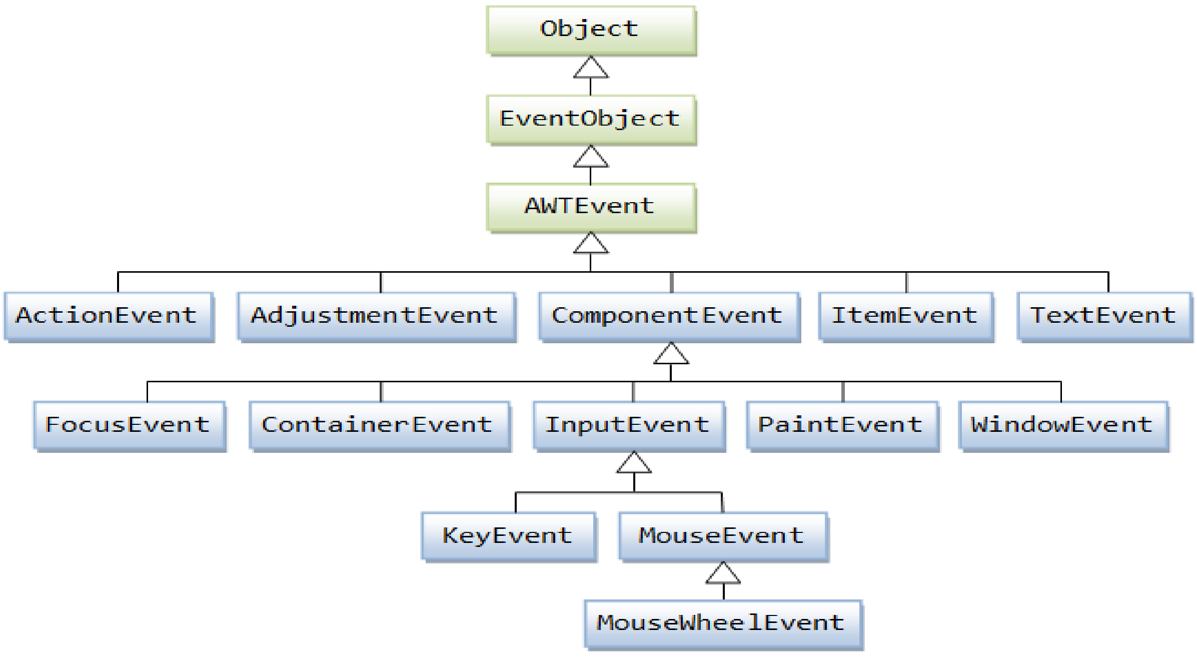


Figure 22: AWT Event Class Diagram

9.2 Events

An event is an *object* that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are *pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse*.

9.2.1 Types of Event

The events can be broadly classified into two categories:
Foreground Events - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in

Graphical User Interface. For example, *clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page* etc.
Background Events - Those events that require the interaction of end user are known as background events. *Operating system interrupts, hardware or software failure, timer expires, an operation completion* are the example of background events.

9.2.2 What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as **event handler** that is executed when an event occurs. Java Uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to *generate and handle the events*. Its concept is quite simple: ***a source generates an event and sends it to one or more listeners***. In this scheme, *the listener simply waits until it receives an event*. Once an event is received, *the listener processes the event and then returns*. The advantage of this design is that the *application logic that processes events is cleanly separated from the user interface logic that generates those events*. A user interface element is able to "*delegate*" the processing of an event to a separate piece of code. In the delegation event model, *listeners must register with a source in order to receive an event notification*. This provides an important benefit: *notifications are sent only to listeners that want to receive them.*

The Delegation Event Model has the following key participants namely:
Source: The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provide as with classes for source object.
Listener: It is also known as **event handler**. Listener is responsible for generating response to an event. From java implementation point of view, the listener is also an object. *Listener waits until it receives an event. Once the event is received, the listener processes the event an then returns*.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, *Listener needs to be registered with the source object so that the listener can receive the event notification*. This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

9.2.2.1 Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

Points to remember about listener
In order to design a listener class, we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
If we do not implement the any if the predefined interfaces, then our class cannot act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces. If a component wants some listener will listen to its events the the source must register itself to the listener.

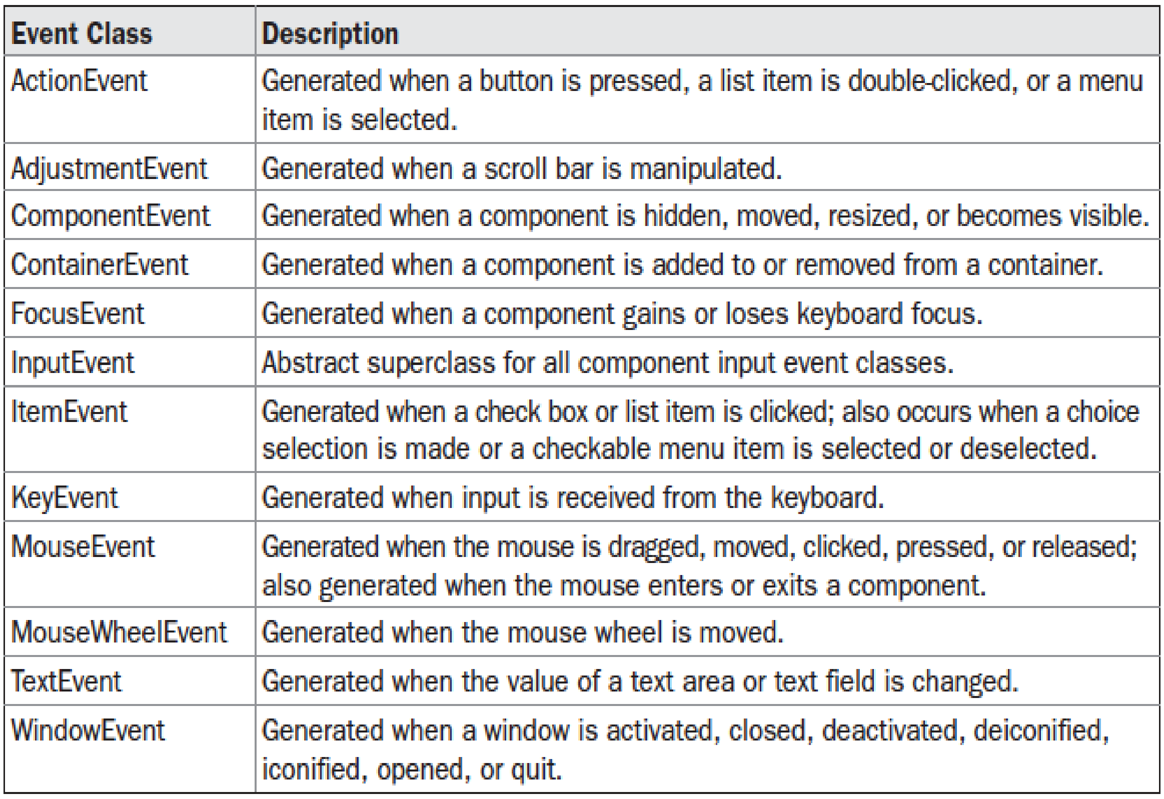
9.3 Event Classes

The classes that represent events are at the core of Java's *event handling mechanism*. Thus, a discussion of event handling must begin with the event classes. The most widely used events are those defined by the AWT and those defined by Swing.

At the root of the Java event class hierarchy is *EventObject,* which is in *java.util.* It is the superclass for all events.

- EventObject is a superclass of all events.
- ***AWTEvent*** is a superclass of all AWT events that are handled by the delegation event model.

The package *java.awt.event* defines many types of events that are generated by various user interface elements. Following table shows several commonly used event classes and provides a brief description of when they are generated.



Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24: Main Event Classes in java.awt.event

9.3.1 The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. ActionEvent class contains the following methods:

String getActionCommand() – Used to obtain the command name for the invoking ActionEvent object.

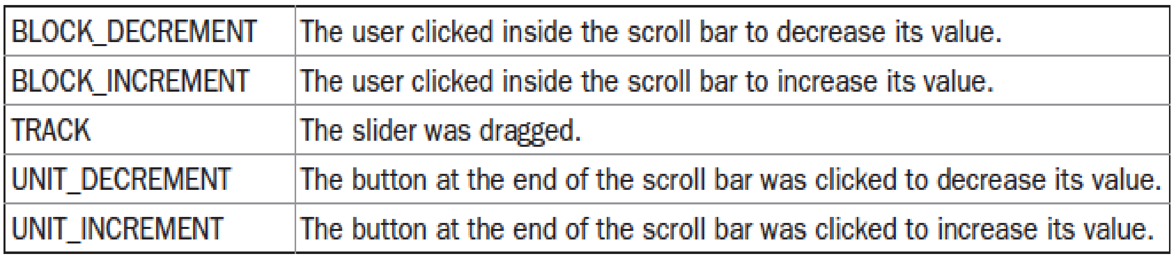
int getModifiers() – This method returns an integer that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed.

long getWhen() – Returns the time when the event was generated.

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

9.3.2 The AdjustmentEvent Class

An ***AdjustmentEvent*** is generated by a scroll bar. There are five types of adjustment events. The *AdjustmentEvent* class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

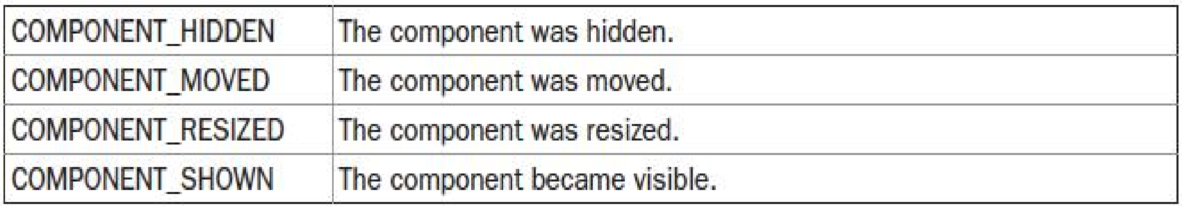


BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The *getAdjustable( )* method returns the object that generated the event. Its form is shown here:
The type of the adjustment event may be obtained by the *getAdjustmentType( )* method. It returns one of the constants defined by *AdjustmentEvent*.
The amount of the adjustment can be obtained from the *getValue( )* method.

9.3.3 The ComponentEvent Class

A *ComponentEvent* is generated when the *size, position, or visibility* of a component is changed. There are four types of component events. The *ComponentEvent* class defines integer constants that can be used to identify them.



COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:
ComponentEvent(Component src, int type)
Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.
ComponentEvent is the superclass either directly or indirectly of *ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.*
The *getComponent( )* method returns the component that generated the event.
Component getComponent( )

9.3.4 The ContainerEvent Class

A *ContainerEvent* is generated when a component is added to or removed from a container.
There are two types of container events. The *ContainerEvent* class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED. They indicate that a component has been added to or removed from the container.
ContainerEvent is a subclass of *ComponentEvent* and has this constructor:
Following are the methods available in the *ContainerEvent* class:
Container getContainer() – Returns the reference of the container that generated the event.

Component getChild() – Returns the reference of the component that is added to or removed from the container.

9.3.5 The FocusEvent Class

A *FocusEvent* is generated when a component gains or loses input focus. These events are identified by the integer constants FOCUS_GAINED and FOCUS_LOST.
FocusEvent is a subclass of *ComponentEvent* and has these constructors:
Following are the methods available in the FocusEvent class:
Component getOppositeComponent() – Returns the other component that gained or lost focus.
boolean isTemporary() – Method returns true or false that indicates whether the change in focus is temporary or not.

9.3.6 The InputEvent Class

The abstract class *InputEvent* is a subclass of *ComponentEvent* and is the superclass for component input events. Its subclasses are *KeyEvent* and *MouseEvent*.
InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the *InputEvent* class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

When writing new code, it is recommended that we use the new, extended modifiers rather than the original modifiers.
To test if a modifier was pressed at the time an event is generated, use the *isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ), and isShiftDown( )* methods. The forms of these methods are shown here:
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
We can obtain a value that contains all of the original modifier flags by calling the *getModifiers( )* method.
int getModifiers( )
We can obtain the extended modifiers by calling *getModifiersEx( ),* which is shown here:
int getModifiersEx( )

9.3.7 The ItemEvent Class

An *ItemEvent* is generated when a *check box or a list item is clicked or when a checkable menu item is selected or deselected*. There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, *ItemEvent* defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state. *ItemEvent* has this constructor:
Following are the methods available in the ItemEvent class:
Object getItem() – Returns a reference to the item whose state has changed.
ItemSelectable getItemSelectable() – Returns the reference of *ItemSelectable*object that raised the event.
int getStateChange() – Returns the status of the state (whether SELECTED or DESELECTED)

9.3.8 The KeyEvent Class

A *KeyEvent* is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated.
There are many other integer constants that are defined by *KeyEvent*. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The VK constants specify virtual key codes and are independent of any modifiers, such as *control*, *shift*, *or alt*. *KeyEvent* is a subclass of *InputEvent*. KeyEvent is a sub class of InputEvent.
Following are the methods available in KeyEvent class:
char getKeyChar() – Returns the character entered from the keyboard
int getKeyCode() – Returns the key code

9.3.9 The MouseEvent Class

There are eight types of mouse events. The *MouseEvent* class defines the following integer constants that can be used to identify them:

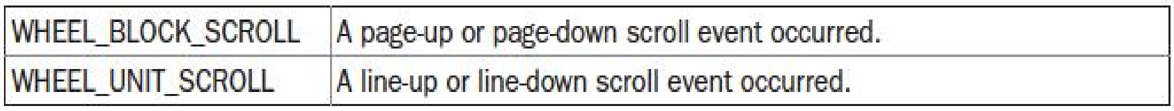
MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

MouseEvent is a subclass of *InputEvent*. Here is one of its constructors:
Following are some of the methods available in MouseEvent class:
int getX() – Returns the x-coordinate of the mouse at which the event was generated.

int getY() – Returns the y-coordinate of the mouse at which the event was generated.
Point getPoint() – Returns the x and y coordinates of mouse at which the event was generated.
int getClickCount() – This method returns the number of mouse clicks for an event.
int getButton() – Returns an integer that represents the buttons that caused the event. Returned value can be either NOBUTTON, BUTTON1 (left mouse button), BUTTON2 (middle mouse button), or BUTTON3 (right mouse button).
Following methods are available to obtain the coordinates relative to the screen:
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()

9.3.10 The MouseWheelEvent Class

The *MouseWheelEvent* class encapsulates a mouse wheel event. It is a subclass of *MouseEvent*. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. *MouseWheelEvent* defines these two integer constants:



WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by *MouseWheelEvent*:
MouseWheelEvent(Component src, int type, long when, int modifiers,
int x, int y, int clicks, boolean triggersPopup,
int scrollHow, int amount, int count)
Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in x and y. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL. The number of units to scroll is passed in amount. The count parameter indicates the number of rotational units that the wheel moved.

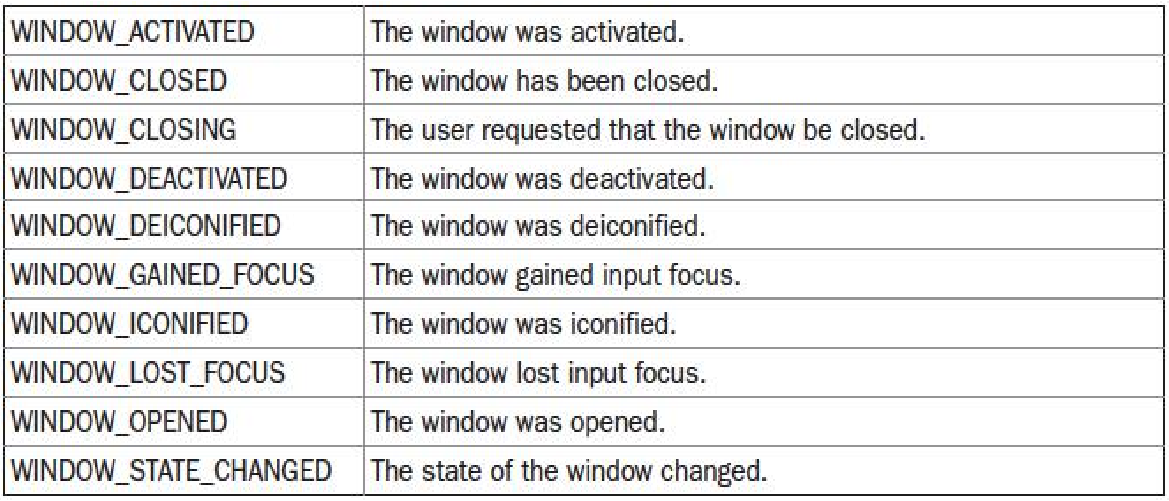
9.3.11 The TextEvent Class

Instances of this class describe text events. These are generated by *text fields and text areas* when characters are entered by a user or program. *TextEvent* defines the integer constant TEXT_VALUE_CHANGED.
The one constructor for this class is shown here:
TextEvent(Object src, int type)
Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.
The *TextEvent* object does not include the characters currently in the text component that generated the *event*. Instead, our program must use other methods associated with the text component to retrieve that information.

9.3.12 The WindowEvent Class

A window event is generated when a user performs any one of the window related events which are identified by the following constants as shown in table:

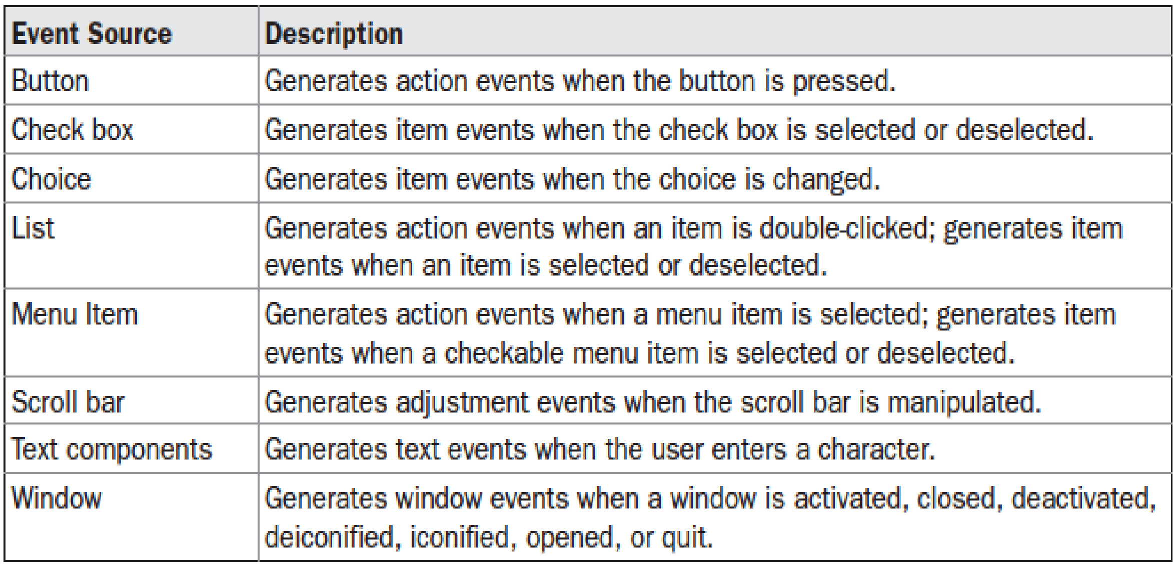
Following are some of the methods available in the WindowEvent class:
Window getWindow() – Returns the reference of the window on which the event is generated.
Window getOppositeWindow() – Returns the reference to the previous window when a focus event of activation event has occurred.
int getOldState() – Returns an integer indicating the old state of the window.
int getNewState() – Returns an integer indicating the new state of the window.



WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

9.4 Sources of Events

Following table lists some of the user interface components that can generate the events. In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events. For example, we can receive key and mouse events from an applet.



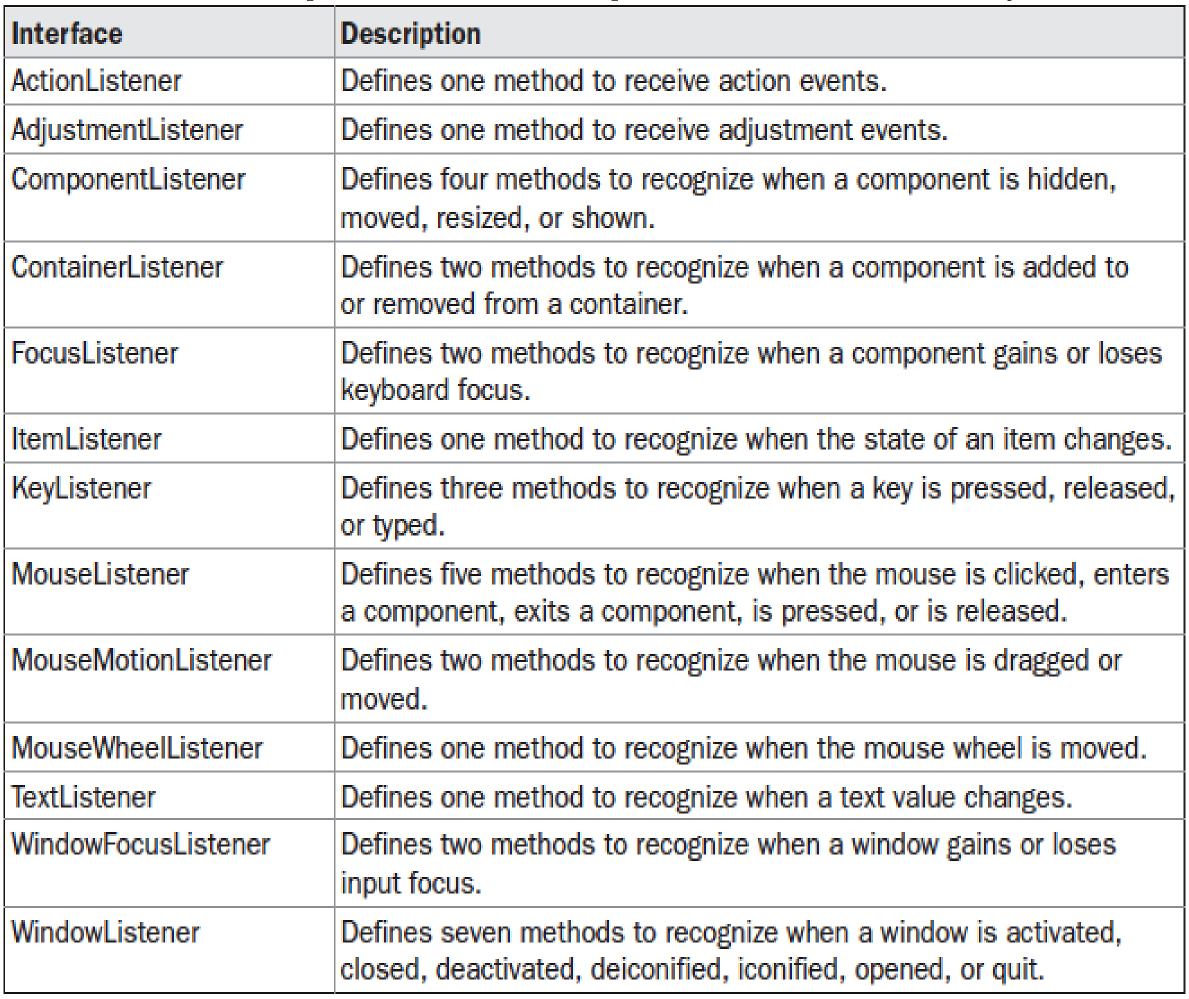
Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table: Sources of Events

9.5 Event Listener Interfaces

The delegation event model has two parts: *sources and listeners*. Listeners are created by implementing one or more of the interfaces defined by the *java.awt.event* package.

When an event occurs, the event source invokes the appropriate method defined by the *listener* and provides *an event object* as its argument. Below table lists commonly used *listener* interfaces and provides a brief description of the methods that they define.



Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 25: Commonly Used Event Listener Interfaces

9.6 Handling Mouse Event

Handling mouse event using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

To handle mouse events, we must implement the *MouseListener* and the *MouseMotionListener* interfaces.
The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, *mouseX* and

mouseY, store the location of the mouse when a *mouse pressed, released, or dragged event occurs*. These coordinates are then used by *paint( )* to display output at the point of these occurrences.

// Demonstrate the mouse event handlers.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener,
MouseMotionListener {
      String msg = "";
      int mouseX = 0, mouseY = 0; // coordinates of mouse
      public void init() {
            addMouseListener(this);
            addMouseMotionListener(this);
      }
      // Handle mouse clicked.
      public void mouseClicked(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse clicked.";
            repaint();
      }
      // Handle mouse entered.
      public void mouseEntered(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse entered.";
            repaint();
      }
      // Handle mouse exited.
      public void mouseExited(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse exited.";
            repaint();
      }
      // Handle button pressed.
      public void mousePressed(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "Down";
            repaint();
      }
      // Handle button released.
      public void mouseReleased(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "Up";
            repaint();
      }
      // Handle mouse dragged.
      public void mouseDragged(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "*";
            showStatus("Dragging mouse at " + mouseX + ", " +
            mouseY);
            repaint();
      }
      // Handle mouse moved.
      public void mouseMoved(MouseEvent me) {
            // show status
            showStatus("Moving mouse at " + me.getX() + ", " +
            me.getY());
      }
      // Display msg in applet window at current X,Y location.
      public void paint(Graphics g) {
            g.drawString(msg, mouseX, mouseY);
      }
}
```

The ***MouseEvents*** class extends ***Applet*** and implements both the ***MouseListener*** and ***MouseMotionListener*** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the ***applet*** is both the source and the listener for these events. This works because Component, which supplies the ***addMouseListener()*** and ***addMouseMotionListener()*** methods, is a superclass of ***Applet***. Being both the source and the listener for events is a common situation for applets. Inside ***init( ),*** the applet registers itself as a listener for mouse events. This is done by using ***addMouseListener( )*** and ***addMouseMotionListener( ),*** which, are the members of Component.

9.7 Swing

Swing API is set of extensible GUI Components to ease developer's life to create JAVA based Front
End/ GUI Applications. It is built upon top of AWT API and acts as replacement of AWT API as it has almost every control corresponding to AWT controls. Swing component follows a ***Model-View-***
Controller architecture to fulfill the following criteria.

- A single API is to be sufficient to support multiple look and feel.
- API is to model driven so that highest level API is not required to have the data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers to use it.

MVC Architecture

Swing API architecture follows loosely based MVC architecture in the following manner.

- A Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.

- Swing component have Model as a separate element and View and Controller part are clubbed in User Interface elements. Using this way, Swing has pluggable look-and-feel architecture.

As a programmer using Swing components, we generally don't need to think about the model-view-controller architecture. Each user interface has a *wrapper class* (such as **JButton** or **JTextField**) that stores the model and the view. When we want to inquire about the contents (for example, the text in a text field), *the wrapper class asks the model and returns the answer to us*. When we want to change the view (for example, move the caret position in a text field), the *wrapper class forwards that request to the view*. However, there are occasions where the wrapper class doesn't work hard enough on forwarding commands. Then, we have to ask it to retrieve the model and work directly with the model. (We don't have to work directly with the view-that is the job of the look-and-feel code.)

Besides being "*the right thing to do*," the ***model-view-controller*** pattern was attractive for the Swing designers because it allowed them to implement pluggable look and feel. The model of a button or text field is independent of the look-and-feel. But of course the visual representation is completely dependent on the user interface design of a particular look and feel. The controller can vary as well.

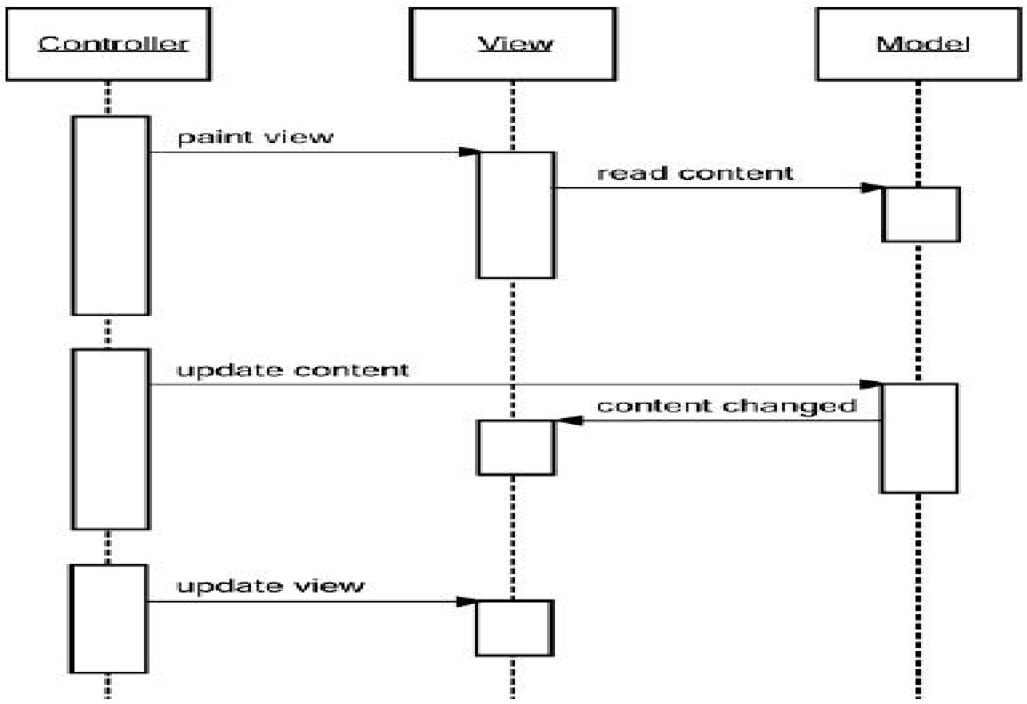


Figure 23: Interaction Between Model, View and Controller objects

9.7.1 Swing Features

- **Light Weight** - Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, table controls.
- **Highly Customizable** - Swing controls can be customized in very easy way as visual appearance is independent of internal representation.

- **Pluggable look and feel**- SWING based GUI Application look and feel can be changed at run time based on available values.

9.7.2 Java Swing VS Java AWT

Java AWT	Java Swing
AWT components are **platform-dependent**.	Java swing components are **platform-independent.**
AWT components are **heavyweight.**	Swing components are **lightweight.**
AWT doesn't support **pluggable look and feel**.	Swing supports **pluggable look and feel.**
AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT doesn't follows MVC where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC.

Table 26: Java AWT vs Java Swing

9.7.3 Swing Controls

Every user interface considers the following three main aspects:

- **UI elements**: These are the core visual elements the user eventually sees and interacts with. AWT provides a huge list of widely used and common elements varying from basic to complex.
- **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).
- **Behaviour**: These are events which occur when the user interacts with UI elements.

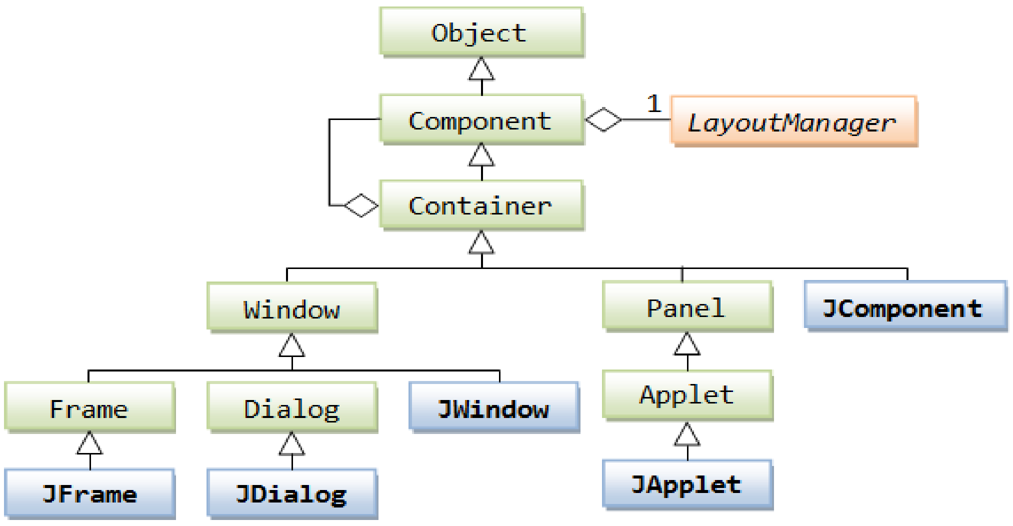


Figure 24: Swing Container Class Diagram

Every SWING controls inherits properties from Component class hierarchy.

1. **Component:** A Container is the abstract base class for the non-menu user-interface controls of SWING. Component represents an object with graphical representation
2. **Container:** A Container is a component that can contain other SWING components.
3. **JComponent:** A JComponent is a base class for all swing UI components. In order to use a swing component that inherits from JComponent, component must be in a

containment hierarchy whose root is a top-level Swing container.

9.7.3.1 Swing UI Elements

Following is the list of commonly used controls while designed GUI using SWING.

SN	Control & Description
1	**JLabel**: A JLabel object is a component for placing text in a container.
2	**JButton:** This class creates a labeled button.
3	**JColorChooser:** A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.
4	**JCheck Box:** A JCheckBox is a graphical component that can be in either an on true or off false state.
5	**JRadioButton**: The JRadioButton class is a graphical component that can be in either an on true or off false state. in a group.
6	**JList:** A JList component presents the user with a scrolling list of text items.
7	**JComboBox**: A JComboBox component presents the user with a to show up menu of choices.
8	**JTextField:** A JTextField object is a text component that allows for the editing of a single line of text.
9	**JPasswordField:** A JPasswordField object is a text component specialized for password entry.
10	**JTextArea:** A JTextArea object is a text component that allows for the editing of a multiple lines of text.
11	**ImageIcon:** A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
12	**JScrollbar:** A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
13	**JOptionPane:** JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
14	**JFileChooser:** A JFileChooser control represents a dialog window from which the user can select a file.
15	**JProgressBar:** As the task progresses towards completion, the progress bar displays the task's percentage of completion.
16	**JSlider:** A JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
17	**JSpinner:** A JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

Table 27: commonly used controls in Swing

Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Simple Swing Program

```
import javax.swing.*;
public class FirstSwingExample extends JFrame{
    FirstSwingExample()
    {
        JFrame  frm=new  JFrame("First  Swing  Example");//creating
instance of JFrame
        JButton btn = new JButton("Click Me");//creating instance of
JButton
        btn.setBounds(130, 100, 100, 40);//x axis, y axis, width,
height
        frm.add(btn);//adding button in JFrame
        frm.setSize(400, 500);//400 width and 500 height
        frm.setLayout(null);//using no layout managers
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frm.setVisible(true);//making the frame visible
    }
    public static void main(String[] args) {
        new FirstSwingExample();
    }
}
```

Another Example

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;
public class SwingExample {

    public static void main(String[] args) {
        // Creating instance of JFrame
        JFrame frame = new JFrame("Swing Example");
        // Setting the width and height of frame
        frame.setSize(350, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      /*Creating panel. Inside panels we can add text  fields,
buttons and other components. */
        JPanel panel = new JPanel();
        // adding panel to frame
        frame.add(panel);
/* calling user defined method for adding components to the panel.
*/
        placeComponents(panel);
        // Setting the frame visibility to true
        frame.setVisible(true);
    }
    private static void placeComponents(JPanel panel) {
        panel.setLayout(null);
        // Creating JLabel
        JLabel userLabel = new JLabel("User");
   /*This method specifies the location and size of component.
setBounds(x, y, width, height) here (x,y) are cordinates from the
top left corner and remaining two arguments are the width and height
of the component. */
        userLabel.setBounds(10,20,80,25);
        panel.add(userLabel);

        /* Creating text field where user is supposed to enter user
name. */
        JTextField userText = new JTextField(20);
        userText.setBounds(100,20,165,25);
        panel.add(userText);
        // Same process for password label and text field.
        JLabel passwordLabel = new JLabel("Password");
        passwordLabel.setBounds(10,50,80,25);
        panel.add(passwordLabel);

        JPasswordField passwordText = new JPasswordField(20);
        passwordText.setBounds(100,50,165,25);
        panel.add(passwordText);

        // Creating login button
        JButton loginButton = new JButton("login");
        loginButton.setBounds(10, 80, 80, 25);
        panel.add(loginButton);
    }

}
```

9.7.4 Swing Event Handling Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingControlDemo {
    private JFrame mainFrame;
    private JLabel headerLabel;
    private JLabel statusLabel;
    private JPanel controlPanel;

    public SwingControlDemo() {
        prepareGUI();
    }
    public static void main(String[] args) {
        SwingControlDemo scd = new SwingControlDemo();
        scd.showEventDemo();
    }
    private void prepareGUI() {
        mainFrame = new JFrame("Java SWING Examples");
        mainFrame.setSize(400, 400);
        mainFrame.setLayout(new GridLayout(3, 1));
        headerLabel = new JLabel("", JLabel.CENTER);
        statusLabel = new JLabel("", JLabel.CENTER);
        statusLabel.setSize(350, 100);

        controlPanel = new JPanel();
        controlPanel.setLayout(new FlowLayout());
        mainFrame.add(headerLabel);
        mainFrame.add(controlPanel);
        mainFrame.add(statusLabel);
        mainFrame.setVisible(true);
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    private void showEventDemo() {
        headerLabel.setText("Control in action: Button");
        JButton okButton = new JButton("OK");
        JButton submitButton = new JButton("Submit");
        JButton cancelButton = new JButton("Cancel");
        okButton.setActionCommand("OK");
        submitButton.setActionCommand("Submit");
        cancelButton.setActionCommand("Cancel");
        okButton.addActionListener(new ButtonClickListener());
        submitButton.addActionListener(new ButtonClickListener());
        cancelButton.addActionListener(new ButtonClickListener());
        controlPanel.add(okButton);
        controlPanel.add(submitButton);
        controlPanel.add(cancelButton);
        mainFrame.setVisible(true);
    }
    private class ButtonClickListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            String command = e.getActionCommand();
            if (command.equals("OK")) {
                statusLabel.setText("Ok Button clicked.");
            } else if (command.equals("Submit")) {
                statusLabel.setText("Submit Button clicked.");
            } else {
                statusLabel.setText("Cancel Button clicked.");
            }
        }
    }
}
```

JRadioButton example with event handling

```
import javax.swing.*;
import java.awt.event.*;

class RadioExample extends JFrame implements ActionListener {
    JRadioButton rb1, rb2;
    JButton b;

    RadioExample() {
        rb1 = new JRadioButton("Male");
        rb1.setBounds(100, 50, 100, 30);

        rb2 = new JRadioButton("Female");
        rb2.setBounds(100, 100, 100, 30);

        ButtonGroup bg = new ButtonGroup();
        bg.add(rb1);
        bg.add(rb2);

        b = new JButton("click");
        b.setBounds(100, 150, 80, 30);
        b.addActionListener(this);

        add(rb1);
        add(rb2);
        add(b);

        setSize(300, 300);
        setLayout(null);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (rb1.isSelected()) {
            JOptionPane.showMessageDialog(this, "You are male");
        }
        if (rb2.isSelected()) {
            JOptionPane.showMessageDialog(this, "You are female");
        }
    }
    public static void main(String args[]) {
        new RadioExample();
    }
}
```

9.7.5 A Model-View-Controller Analysis of Swing Buttons

Buttons are about the simplest user interface elements, so they are a good place to become comfortable with the *model-view-controller* pattern. For most components, the model class implements an interface whose name ends in ***Model***. Thus, there is an interface called ***ButtonModel***. Classes implementing that interface can define the state of the various kinds of buttons. Actually, buttons aren't all that complicated, and the Swing library contains a single class, called ***DefaultButtonModel***, that implements this interface.

`getActionCommand()`	The action command string associated with this button
`getMnemonic()`	The keyboard mnemonic for this button
`isArmed()`	`true` if the button was pressed and the mouse is still over the button
`isEnabled()`	`true` if the button is selectable
`isPressed()`	`true` if the button was pressed but the mouse button hasn't yet been released
`isRollover()`	`true` if the mouse is over the button
`isSelected()`	`true` if the button has been toggled on (used for check boxes and radio buttons)

Table: The accessor methods of the ***ButtonModel*** interface

Each ***JButton*** object stores a button model object, which we can retrieve.

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

9.7.5.1 Building GUI with Swing

The following examples demonstrate the use of ***JButton*** in which whenever we click one of the buttons, the appropriate action listener changes the background color of the panel.

```
import java.awt.*;
 import java.awt.event.*;
 import javax.swing.*;

 public class ButtonTest
 {
      public static void main(String[] args)
      {
           ButtonFrame frame = new ButtonFrame();
           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.show();
      }
 }

 /**
 A frame with a button panel  */
 class ButtonFrame extends JFrame
 {
     public ButtonFrame()
     {
           setTitle("ButtonTest");
           setSize(WIDTH, HEIGHT);

     // add panel to frame

      ButtonPanel panel = new ButtonPanel();
      Container contentPane = getContentPane();
      contentPane.add(panel);
     }

     public static final int WIDTH = 300;
     public static final int HEIGHT = 200;
 }

 /** A panel with three buttons.  */
 class ButtonPanel extends JPanel
 {
     public ButtonPanel()
     {
            // create buttons

      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
     JButton redButton = new JButton("Red");

     // add buttons to panel

      add(yellowButton);
      add(blueButton);
      add(redButton);

     // create button actions

     ColorAction yellowAction = new ColorAction(Color.yellow);
     ColorAction blueAction = new ColorAction(Color.blue);
      ColorAction redAction = new ColorAction(Color.red);
     // associate actions with buttons
```

```
      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
     redButton.addActionListener(redAction);
   }

 /** An action listener that sets the panel's background color. */
      private class ColorAction implements ActionListener
     {
           public ColorAction(Color c)
           {
                backgroundColor = c;
           }
           public void actionPerformed(ActionEvent event)
           {
                setBackground(backgroundColor);
                repaint();
           }
           private Color backgroundColor;
     }
}
```

9.7.6 Layout Management

The **LayoutManagers** are used to arrange components in a particular manner. **LayoutManager** is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

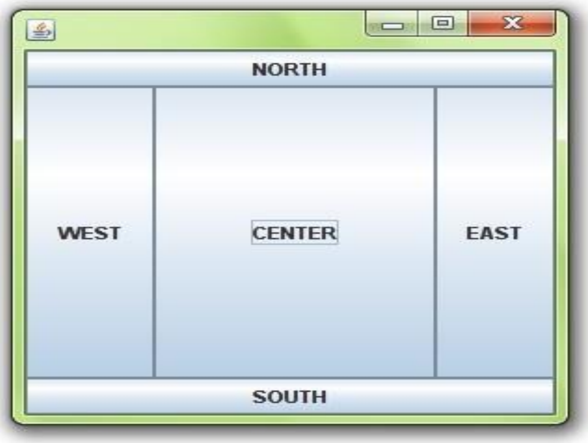
1. BorderLayout
2. FlowLayout
3. GridLayout
4. CardLayout
5. GridBagLayout
6. BoxLayout
7. GroupLayout
8. ScrollPaneLayout
9. SpringLayout etc.

9.7.6.1 BorderLayout:

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region: **NORTH ,SOUTH, EAST, WEST, CENTER**

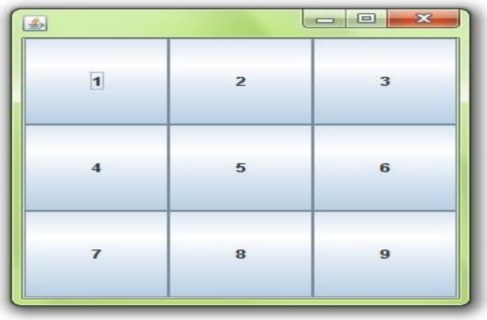
Constructors of BorderLayout class

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.



```
import java.awt.*;
import javax.swing.*;

public class Border extends JFrame {
    JFrame f;
    Border() {
        f = new JFrame();
        JButton b1 = new JButton("NORTH");;
        JButton b2 = new JButton("SOUTH");;
        JButton b3 = new JButton("EAST");;
        JButton b4 = new JButton("WEST");;
        JButton b5 = new JButton("CENTER");;

        f.add(b1, BorderLayout.NORTH);
        f.add(b2, BorderLayout.SOUTH);
        f.add(b3, BorderLayout.EAST);
        f.add(b4, BorderLayout.WEST);
        f.add(b5, BorderLayout.CENTER);
        f.setSize(300, 300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new Border();
    }
}
```

9.7.6.2 GridLayout



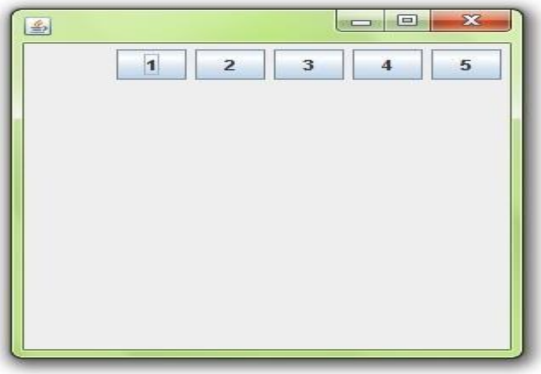
The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class:

- **GridLayout():** creates a grid layout with one column per component in a row.
- **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
- **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

```
import java.awt.*;
import javax.swing.*;

public class MyGridLayout {
JFrame f;
MyGridLayout() {
        f = new JFrame();

        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        JButton b5 = new JButton("5");
        JButton b6 = new JButton("6");
        JButton b7 = new JButton("7");
        JButton b8 = new JButton("8");
        JButton b9 = new JButton("9");

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);
        f.add(b7);
        f.add(b8);
        f.add(b9);

        f.setLayout(new GridLayout(3, 3));
        //setting grid layout of 3 rows and 3 columns

        f.setSize(300, 300);
        f.setVisible(true);
    }

    public static void main(String[] args) {
        new MyGridLayout();
    }
}
```

9.7.6.3 FlowLayout



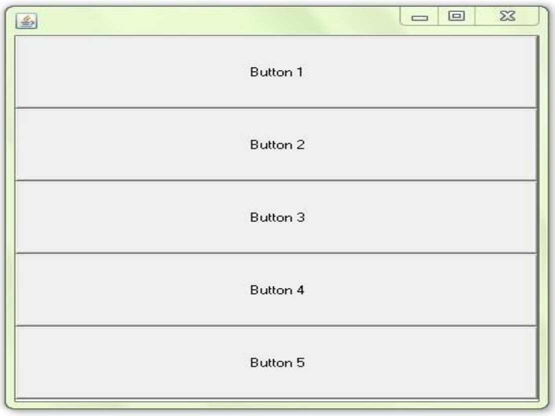
The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel. The FlowLayout provides five constants: **LEFT, RIGHT, CENTER, LEADING, TRAILING.**

Constructors of FlowLayout class:

- **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap
- **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap
- **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout {
    JFrame f;
    MyFlowLayout() {
        f = new JFrame();

        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        JButton b5 = new JButton("5");

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.setLayout(new FlowLayout(FlowLayout.RIGHT));
        //setting flow layout of right alignment

        f.setSize(300, 300);
        f.setVisible(true);
    }

    public static void main(String[] args) {
        new MyFlowLayout();
    }
}
```

9.7.6.4 BoxLayout class:

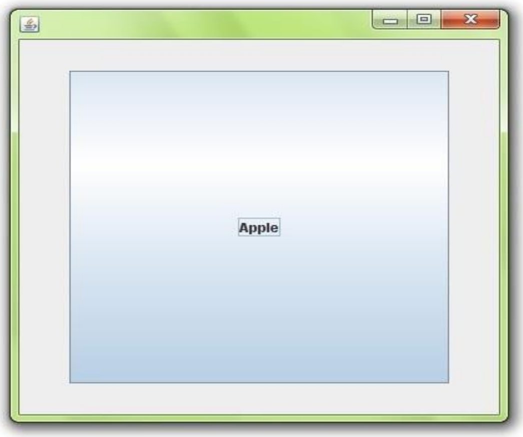


The BoxLayout is used to arrange the components either vertically or horizontally. For this purpose, BoxLayout provides four constants. They are as follows: **X_AXIS, Y_AXIS, LINE_AXIS, PAGE_AXIS.**
Note: BoxLayout class is found in javax.swing package.

Constructor of BoxLayout class:

- **BoxLayout(Container c, int axis):** creates a box layout that arranges the components with the given axis

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample extends Frame {

    Button buttons[];

    public BoxLayoutExample() {
        buttons = new Button[5];

        for (int i = 0; i < 5; i++) {
            buttons[i] = new Button("Button " + (i + 1));
            add(buttons[i]);
        }
        setLayout(new
BoxLayout(this, BoxLayout.Y_AXIS));
        setSize(400, 400);
        setVisible(true);
    }
    public static void main(String
args[]) {
        BoxLayoutExample b = new
BoxLayoutExample();
    }
}
```



9.7.6.5 CardLayout class

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class

CardLayout(): creates a card layout with zero horizontal and vertical gap.
CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class:

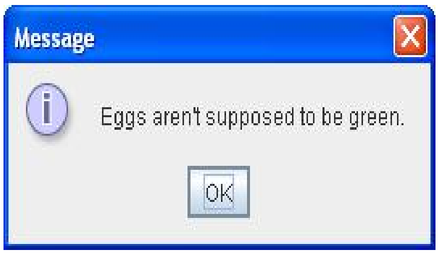
1. **public void next(Container parent):** is used to flip to the next card of the given container.
2. **public void previous(Container parent):** is used to flip to the previous card of the given container.
3. **public void first(Container parent):** is used to flip to the first card of the given container.
4. **public void last(Container parent):** is used to flip to the last card of the given container.
5. **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements
ActionListener {
    CardLayout card;
    JButton b1, b2, b3;
    Container c;
    CardLayoutExample() {
        c = getContentPane();
        card = new CardLayout(40, 30);
//create CardLayout object with 40 hor space and 30 ver space
        c.setLayout(card);
        b1 = new JButton("Apple");
        b2 = new JButton("Boy");
        b3 = new JButton("Cat");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        c.add("a", b1);
        c.add("b", b2);
        c.add("c", b3);
    }
    public void actionPerformed(ActionEvent e) {
        card.next(c);
    }
    public static void main(String[] args) {
        CardLayoutExample cl = new CardLayoutExample();
        cl.setSize(400, 400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

9.7.7 Dialog Box

A Dialog window is an independent sub window meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

Several Swing component classes can directly instantiate and display *dialogs*. To create simple, standard dialogs, we use the `JOptionPane` class. The `ProgressMonitor` class can put up a dialog that shows the progress of an operation. Two other classes, `JColorChooser` and `JFileChooser`, also supply standard dialogs.



The code for simple dialogs can be minimal. For example, here is an informational dialog:

Code:

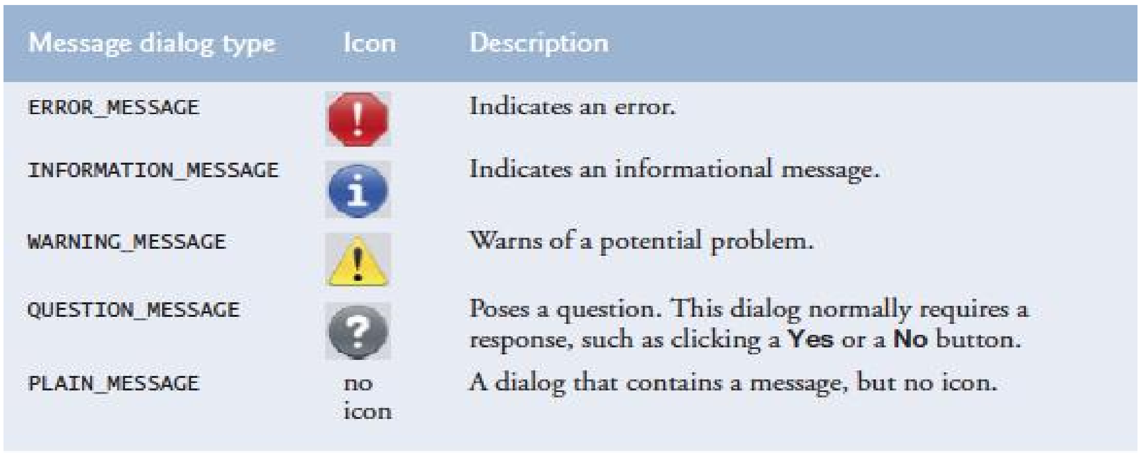
```
JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.");
```

Simple Example program of JOptionPane

```
import javax.swing.JOptionPane;

public class NameDialog
{
   public static void main( String[] args )
   {
      // prompt user to enter name
      String name =
         JOptionPane.showInputDialog( "What is your name?" );
      // create the message
      String message =
         String.format( "Welcome, %s, to Java Programming!", name );

      // display the message to welcome the user by name
      JOptionPane.showMessageDialog( null, message );
   } // end main
} // end class NameDialog
```

Addition Using JOptionPane

```
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane
public class Addition
{
   public static void main( String[] args )
   {
      // obtain user input from JOptionPane input dialogs
      String firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );
      String secondNumber =
          JOptionPane.showInputDialog( "Enter second integer" );
      // convert String inputs to int values for use in a
calculation
      int number1 = Integer.parseInt( firstNumber );
      int number2 = Integer.parseInt( secondNumber );
      int sum = number1 + number2; // add numbers
      // display result in a JOptionPane message dialog
      JOptionPane.showMessageDialog( null, "The sum is " + sum,
         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
   } // end method main
} // end class Addition
```



Message dialog type	Icon	Description
ERROR_MESSAGE	❗	Indicates an error.
INFORMATION_MESSAGE	ℹ	Indicates an informational message.
WARNING_MESSAGE	⚠	Warns of a potential problem.
QUESTION_MESSAGE	❓	Poses a question. This dialog normally requires a response, such as clicking a **Yes** or a **No** button.
PLAIN_MESSAGE	no icon	A dialog that contains a message, but no icon.

Table 28: JOptionPane static constants for message dialogs.

9.7.8 Scroll Bar

A `JScrollPane` provides a scrollable view of a component. When screen real estate is limited, use a scroll pane to display a component that is large or one whose size can change dynamically. Other containers used to save screen space include split panes and tabbed panes. Here's the code that creates the text area, makes it the scroll pane's client, and adds the scroll pane to a container:

```
//In a container that uses a BorderLayout:
textArea = new JTextArea(5, 30);
...
JScrollPane scrollPane = new JScrollPane(textArea);
...
setPreferredSize(new Dimension(450, 110));
...
add(scrollPane, BorderLayout.CENTER);
```

Additional Program

```
import java.awt.BorderLayout;
import java.awt.GridLayout;

import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JScrollPane;

public class ScrollDemo extends JFrame {

    JScrollPane scrollpane;

    public ScrollDemo() {
        super("JScrollPane Demonstration");
        setSize(300, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        init();
        setVisible(true);
    }

    public void init() {
        JRadioButton form[][] = new JRadioButton[12][5];
        String  counts[]  =  {"",  "0-1",  "2-5",  "6-10",  "11-100",
"101+"};
        String  categories[]  =  {"Household",  "Office",  "Extended
Family",
            "Company (US)", "Company (World)", "Team", "Will",
            "Birthday  Card  List",  "High  School",  "Country",
"Continent",
            "Planet"};
        JPanel p = new JPanel();
        p.setSize(600, 400);
        p.setLayout(new GridLayout(13, 6, 10, 0));
        for (int row = 0; row < 13; row++) {
            ButtonGroup bg = new ButtonGroup();
            for (int col = 0; col < 6; col++) {
                if (row == 0) {
                    p.add(new JLabel(counts[col]));
                } else {
                    if (col == 0) {
                        p.add(new JLabel(categories[row - 1]));
                    } else {
                        form[row - 1][col - 1] = new JRadioButton();
                        bg.add(form[row - 1][col - 1]);
                        p.add(form[row - 1][col - 1]);
                    }
                }
            }
        }
        scrollpane = new JScrollPane(p);
        getContentPane().add(scrollpane, BorderLayout.CENTER);
    }

    public static void main(String args[]) {
        new ScrollDemo();
    }
}
```

9.7.9 Menus

A menu provides a space-saving way to let the user choose one of several options. Other components with which the user can make a one-of-many choices include *combo boxes*, *lists*, *radio buttons*, *spinners*, and *tool bars*.

Menus are unique in that, by convention, they aren't placed with the other components in the UI. Instead, a menu usually appears either in a *menu bar* or as a *popup menu*. A menu bar contains one or more menus and has a customary, platform-dependent location , usually along the top of a window. A popup menu is a menu that is invisible until the user makes a platform-specific mouse action, such as pressing the right mouse button, over a popup-enabled component. The popup menu then appears under the cursor.

```
import javax.swing.*;
import java.awt.event.*;

public class Notepad implements ActionListener {

    JFrame f;
    JMenuBar mb;
    JMenu file, edit, help;
    JMenuItem cut, copy, paste, selectAll;
    JTextArea ta;

    Notepad() {
        f = new JFrame();

        cut = new JMenuItem("cut");
        copy = new JMenuItem("copy");
        paste = new JMenuItem("paste");
        selectAll = new JMenuItem("selectAll");

        cut.addActionListener(this);
        copy.addActionListener(this);
        paste.addActionListener(this);
        selectAll.addActionListener(this);

        mb = new JMenuBar();
        mb.setBounds(5, 5, 400, 40);

        file = new JMenu("File");
        edit = new JMenu("Edit");
        help = new JMenu("Help");

        edit.add(cut);
        edit.add(copy);
        edit.add(paste);
        edit.add(selectAll);


        mb.add(file);
        mb.add(edit);
        mb.add(help);

        ta = new JTextArea();
        ta.setBounds(5, 30, 460, 460);

        f.add(mb);
        f.add(ta);

        f.setLayout(null);
        f.setSize(500, 500);
        f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == cut) {
            ta.cut();
        }
        if (e.getSource() == paste) {
            ta.paste();
        }
        if (e.getSource() == copy) {
            ta.copy();
        }
        if (e.getSource() == selectAll) {
            ta.selectAll();
        }
    }
    public static void main(String[] args) {
        new Notepad();
    }
}
```

9.7.10 Progress Bar

Sometimes a task running within a program might take a while to complete. A user-friendly program provides some indication to the user that the task is occurring, how long the task might take, and how much work has already been done. One way of indicating work, and perhaps the amount of progress, is to use an animated image.

Commonly used Constructors of JProgressBar class

- **JProgressBar():** is used to create a horizontal progress bar but no string text.
- **JProgressBar(int min, int max):** is used to create a horizontal progress bar with the specified minimum and maximum value.
- **JProgressBar(int orient):** is used to create a progress bar with the specified orientation, it can be either Vertical or Horizontal by using **SwingConstants.VERTICAL** and **SwingConstants.HORIZONTAL** constants.
- **JProgressBar(int orient, int min, int max):** is used to create a progress bar with the specified orientation, minimum and maximum value.

Additional Program

```
import javax.swing.*;
public class MyProgress extends JFrame {
    JProgressBar jb;
    int i = 0, num = 0;

    MyProgress() {
        jb = new JProgressBar(0, 2000);
        jb.setBounds(40, 40, 200, 30);

        jb.setValue(0);
        jb.setStringPainted(true);

        add(jb);
        setSize(400, 400);
        setLayout(null);
    }

    public void iterate() {
        while (i <= 2000) {
            jb.setValue(i);
            i = i + 20;
            try {
                Thread.sleep(150);
            } catch (Exception e) {
            }
        }
    }

    public static void main(String[] args) {
        MyProgress m = new MyProgress();
        m.setVisible(true);
        m.iterate();
    }
}
```

10 Chapter 10: Java Server Pages (JSP)/Servlet Technology

10.1Applets, Servlets, and Java Server Pages

When we instruct our Web browser to view a page from a Web server on the Internet, our Web browser requests the page from the Web server, the Web server processes the request (which may involve reading the requested page from a file on the hard drive), and then the Web server sends the requested page to our Web browser. Our Web browser *formats*, or *renders*, the *received data to fit on our computer screen*. This interaction is a specific case of the ***client/server*** model. Our Web browser is the client program, our computer is the *client computer*, the remote website is the *server computer*, and the Web server software running on the remote website is the *server program*.

In the context of a Web application, the *client/server* model is important because Java code can run in two places: on the ***client*** or on ***the server***. There are trade-offs to both approaches. *Server-based programs have easy access to information that resides on the server*, such as customer orders or inventory data. Because all of the computation is done on the server and results are transmitted to the client as HTML, a client does not need a powerful computer to run a server-based program. On the other hand, a *client-based* program may require a more powerful client computer, because all computation is performed locally. However, richer interaction is possible, because the client program has access to local resources, such as the *graphics display* (e.g., perhaps using Swing) or the operating system. Many systems today are constructed using code that runs on both the client and the server to reap the benefit of both approaches.

Web applications built with Java include ***Java applets, Java servlets***, and ***Java Server Pages (JSP)***. Java applets run on the *client computer*. *JavaScript*, which is a different language than Java despite its similar name, also runs on the client computer as part of the Web browser. ***Java servlets*** and ***Java Server Pages*** run on the ***server***. *Java Server pages* which are a dynamic version of *Java servlets*. *Servlets must be compiled before they can run, just like a normal Java program*. In contrast, *JSP code is embedded with the corresponding HTML and is compiled* "*on the fly*" into a servlet when the page is requested. This flexibility can make it easier to develop Web applications using JSP than with Java servlets. The following fig1, fig2 and fig2 shows that the running a Java applet, Java Servlet and JSP program.

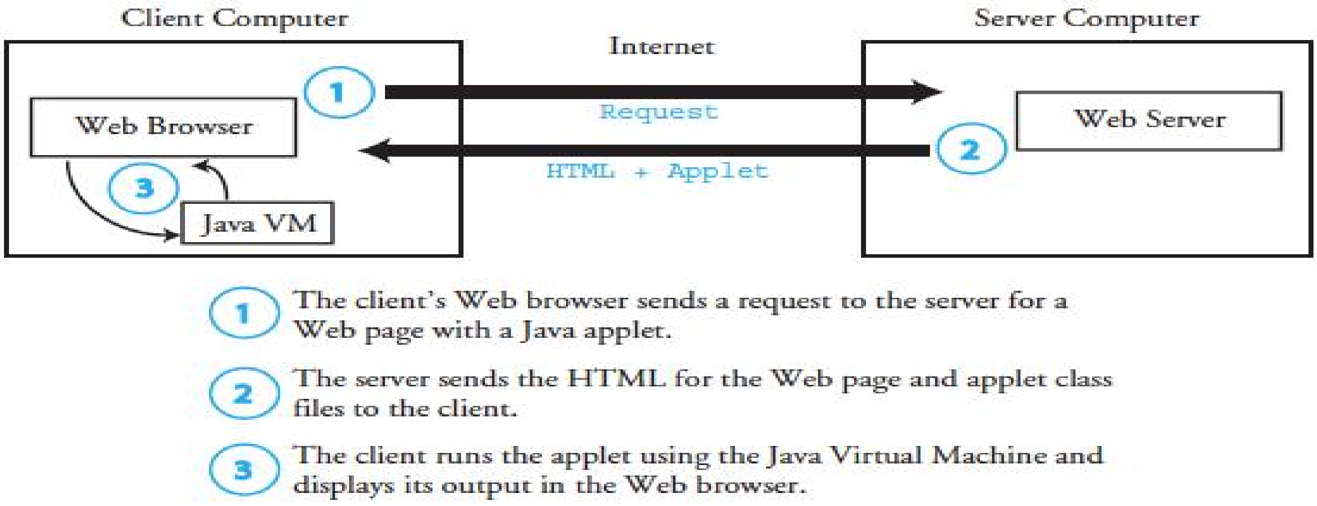
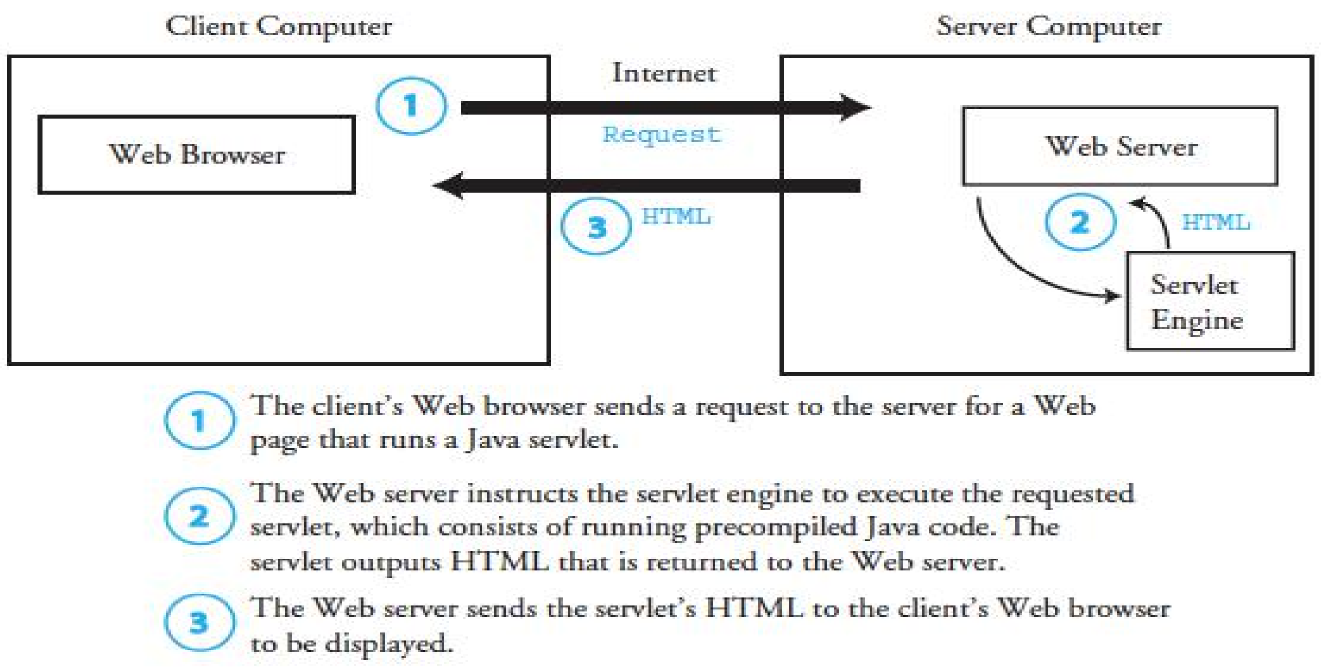

Client Computer
Server Computer
Internet
Web Browser
Web Server
Java VM
Request
HTML + Applet
1
2
3
1 The client's Web browser sends a request to the server for a Web page with a Java applet.
2 The server sends the HTML for the Web page and applet class files to the client.
3 The client runs the applet using the Java Virtual Machine and displays its output in the Web browser.

Figure 25: Running a Java Applet

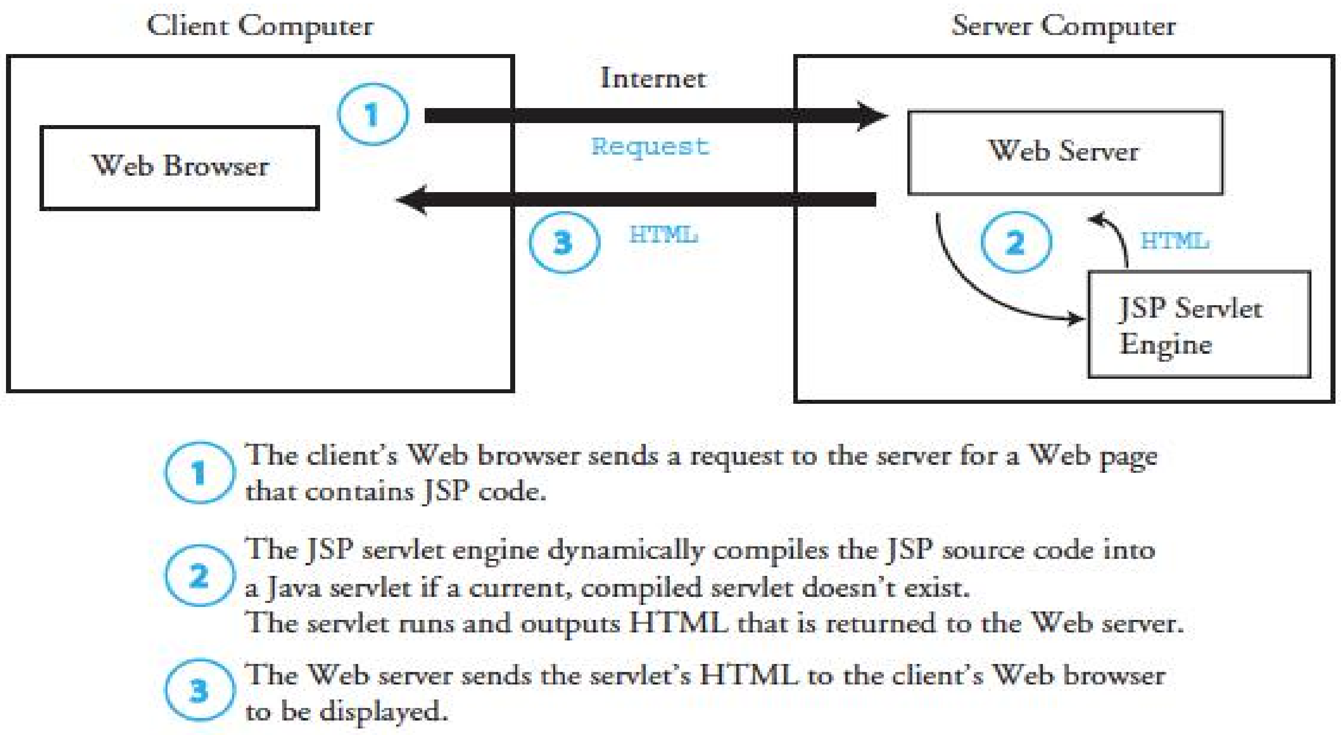


1 The client's Web browser sends a request to the server for a Web page that runs a Java servlet.

2 The Web server instructs the servlet engine to execute the requested servlet, which consists of running precompiled Java code. The servlet outputs HTML that is returned to the Web server.

3 The Web server sends the servlet's HTML to the client's Web browser to be displayed.

Figure 26: Running a Java Servlet



1 The client's Web browser sends a request to the server for a Web page that contains JSP code.

2 The JSP servlet engine dynamically compiles the JSP source code into a Java servlet if a current, compiled servlet doesn't exist. The servlet runs and outputs HTML that is returned to the Web server.

3 The Web server sends the servlet's HTML to the client's Web browser to be displayed.

Figure 27: Running a Java Server Page (JSP) Program

10.2 Servlet

Servlets offer several advantages in comparison with traditional system.

1. ***First***, *performance is significantly better*. Servlets execute within the *address space* of a web server. It is *not necessary* to *create a separate process to handle each client request*.
2. **Second**, *servlets are platform-independent because they are written in Java*.
3. **Third**, *the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine*.
4. **Finally**, *the full functionality of the Java class libraries is available to a servlet*. It can

communicate with *applets, databases, or other software* via the sockets and RMI mechanisms.

10.2.1 The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are ***init( ), service( )***, and ***destroy( ).*** They are implemented by every *servlet* and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

1. **First**, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.
2. **Second**, the web server receives this HTTP request. The server maps this request to a particular ***servlet***. The servlet is dynamically retrieved and loaded into the address space of the server.
3. **Third**, the server invokes the ***init( )*** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.
4. **Fourth**, the server invokes the ***service( )*** method of the servlet. This method is called *to process the HTTP request*. It may also formulate an HTTP response for the client. The servlet remains in the *server's address space* and is available to process any other HTTP requests received from clients. The ***service( )*** method is called for each HTTP request.
5. **Finally**, the server may decide to ***unload*** the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the ***destroy( )*** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

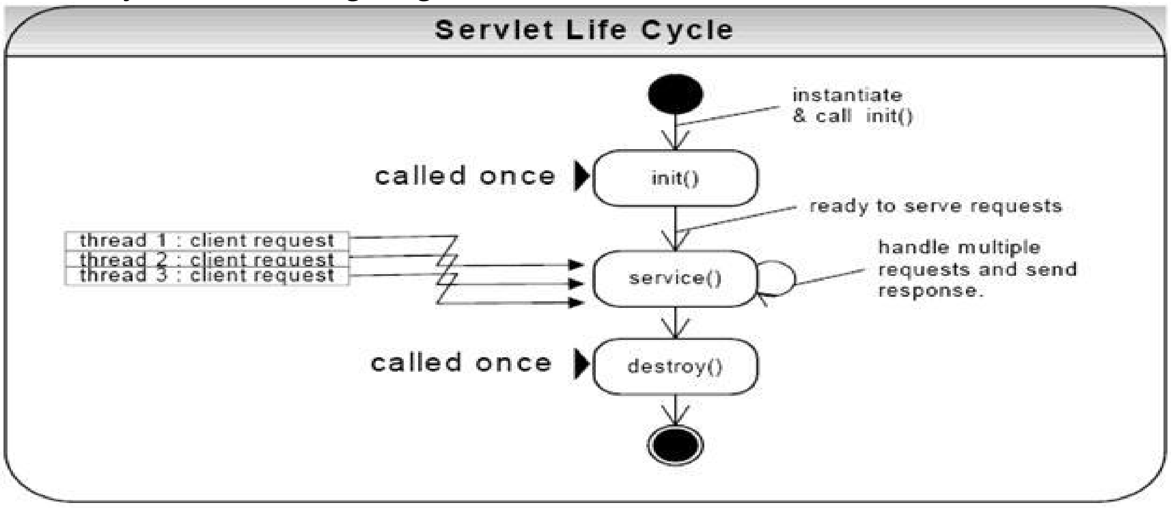

Servlet Life Cycle
instantiate & call init()
called once
init()
ready to serve requests
thread 1 : client request
thread 2 : client request
thread 3 : client request
service()
handle multiple requests and send response.
called once
destroy()

Figure 28: Servlet Life cycle

10.2.2 A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper ***web.xml*** file.
2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named ***HelloServlet.java*** that contains the following program:

```
import java.io.*;
import javax.servlet.*;
public class HelloServlet extends GenericServlet {
      public void service(ServletRequest request, ServletResponse
response) throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter pw = response.getWriter();
      pw.println("<B>Hello!");
      pw.close();
      }
}
```

Let's look closely at this program. First, note that it imports the ***javax.servlet*** package. This package contains the classes and interfaces required to build servlets. Next, the program defines ***HelloServlet*** as a subclass of ***GenericServlet***. The ***GenericServlet*** class provides functionality that simplifies the *creation of a servlet*. For example, it provides versions of ***init***( ) and ***destroy***( ), which may be used as is. We need supply only the ***service***( ) method. Inside ***HelloServlet***, the ***service***( ) method (which is inherited from ***GenericServlet***) is overridden. This method handles requests from a client. Notice that the first argument is a ***ServletRequest*** object. This enables the servlet to read data that is provided via the client request. The second argument is a ***ServletResponse*** object. This enables the servlet to formulate a response for the client.
The call to ***setContentType***( ) establishes the MIME type of the HTTP response. In this program, the MIME type is ***text/html.*** This indicates that the browser should interpret the content as HTML source code. Next, the ***getWriter***( ) method obtains a ***PrintWriter***. Anything written to this stream is sent to the client as part of the HTTP response. Then ***println***( ) is used to write some simple HTML source code as the HTTP response.
Compile this source code and place the ***HelloServlet.class*** file in the proper Tomcat directory. Also, add ***HelloServlet*** to the ***web.xml*** file.

Start Tomcat

Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet

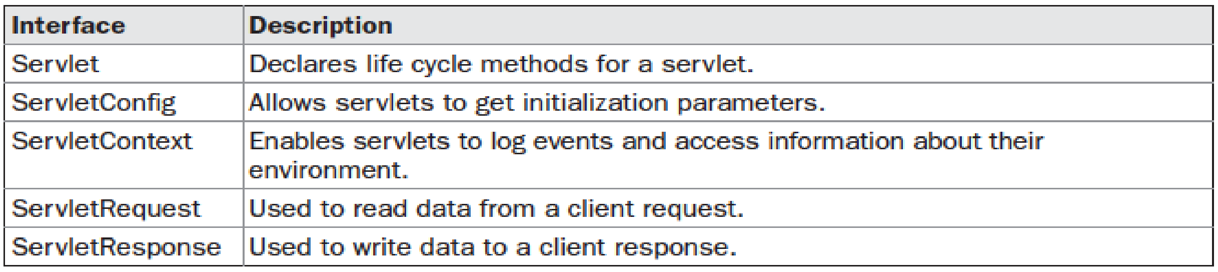
Start a web browser and enter the URL shown here:
http://localhost:8080/servlets-examples/servlet/HelloServlet
Alternatively, we may enter the URL shown here:
http://127.0.0.1:8080/servlets-examples/servlet/HelloServlet
This can be done because 127.0.0.1 is defined as the IP address of the local machine. We will observe the output of the servlet in the browser display area. It will contain the string ***Hello***! in bold type.

10.3 The Servlet API

Two packages contain the classes and interfaces that are required to build servlets. These are ***javax.servlet*** and ***javax.servlet.http***. They constitute the Servlet API.

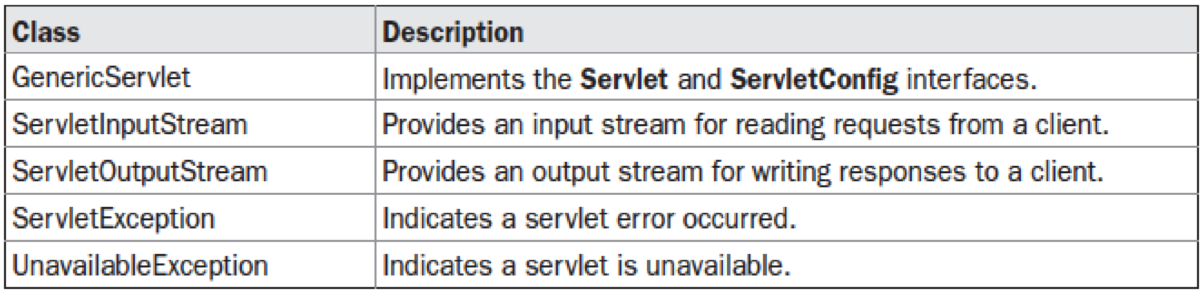
10.3.1 The javax.servlet Package

The ***javax.servlet*** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes the core interfaces that are provided in this package. The most significant of these is ***Servlet***. All servlets must implement this interface or extend a class that implements the interface. The ***ServletRequest*** and ***ServletResponse*** interfaces are also very important.



Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the ***javax.servlet*** package:



Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

10.3.1.1 The Servlet Interface

All servlets must implement the *Servlet interface*. It declares the ***init***( ), ***service***( ), and ***destroy***( ) methods that are called by the server during the *life cycle of a servlet*. A method is also provided that allows a servlet to obtain any initialization parameters. The ***getServletConfig( )*** method is called by the servlet to obtain initialization parameters. A servlet developer overrides the ***getServletInfo( )*** method to provide a string with useful information (for example, author, version, date, copyright). The server invokes this method also.

10.3.1.2 The ServletRequest Interface

The *ServletRequest* interface enables a servlet to obtain information about a client request. Several of its methods are summarized in following table.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

Table 29: Various methods defined by ServletRequest

10.3.1.3 The ServletResponse Interface

The *ServletResponse* interface enables a servlet to formulate a response for a client. Several of its methods are summarized in following table.

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

Table 30: Various methods defined by ServletResponse

10.3.2 The GenericServlet Class

The ***GenericServlt*** class provides implementations of the basic life cycle methods for a servlet. ***GenericServlet*** implements the ***Servlet*** and ***ServletConfig*** interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

void log(String s)

void log(String s, Throwable e)
Here, s is the string to be appended to the log, and e is an exception that occurred.

10.4 Reading Servlet Parameters

The ***ServletRequest*** interface includes methods that allow us to read the names and values of parameters that are included in a client request. The example contains two files. A web page is defined in ***PostParameters.htm***, and a servlet is defined in ***PostParametersServlet.java***. The HTML source code for ***PostParameters.htm*** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
      <body>
      <center>
      <form name="Form1" method="post"
      action="http://localhost:8080/servlets-
      examples/servlet/PostParametersServlet">
            <table>
            <tr>
                  <td><B>Employee</td>
                  <td><input type=textbox name="e" size="25"
                  value=""></td>
            </tr>
            <tr>
                  <td><B>Phone</td>
                  <td><input type=textbox name="p" size="25"
                  value=""></td>
            </tr>
            </table>
      <input type=submit value="Submit">
      </body>
</html>
```

The source code for ***PostParametersServlet.java*** is shown in the following listing. The ***service***( ) method is overridden to process client requests. The ***getParameterNames***( ) method returns an enumeration of the parameter names. These are processed in a loop. We can see that the parameter name and value are output to the client. The parameter value is obtained via the ***getParameter***( ) method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
public class PostParametersServlet extends GenericServlet {
      public void service(ServletRequest request, ServletResponse
response)
      throws ServletException, IOException {
      // Get print writer.
      PrintWriter pw = response.getWriter();
      // Get enumeration of parameter names.
      Enumeration e = request.getParameterNames();
      // Display parameter names and values.
            while(e.hasMoreElements()) {
                  String pname = (String)e.nextElement();
                  pw.print(pname + " = ");
                  String pvalue = request.getParameter(pname);
                  pw.println(pvalue);
            }
      pw.close();
      }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the ***web.xml*** file. Then, perform these steps to test this example:
1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

10.5 The javax.servlet.http Package

The ***javax.servlet.http*** package contains a number of interfaces and classes that are commonly used by servlet developers. The following table summarizes the core interfaces that are provided in this package:

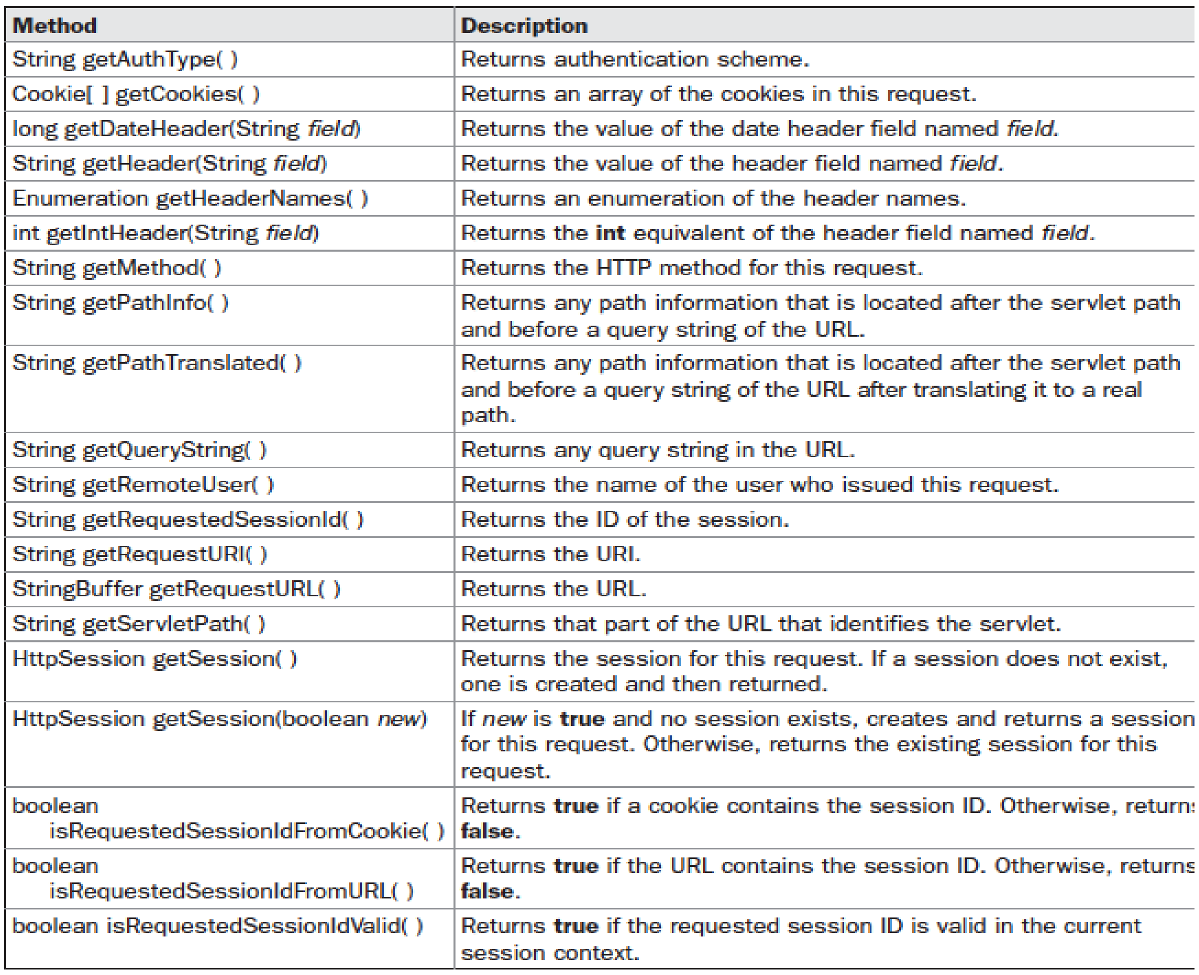
Interface	**Description**
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.
HttpSessionBindingListener	Informs an object that it is bound to or unbound from a session.

The following table summarizes the core classes that are provided in this package. The most important of these is ***HttpServlet***. Servlet developers typically extend this class in order to process HTTP requests.

Class	**Description**
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

10.5.1 The HttpServletRequest Interface

The ***HttpServletRequest*** interface enables a servlet to obtain information about a client request. Several of its methods are shown in following table.

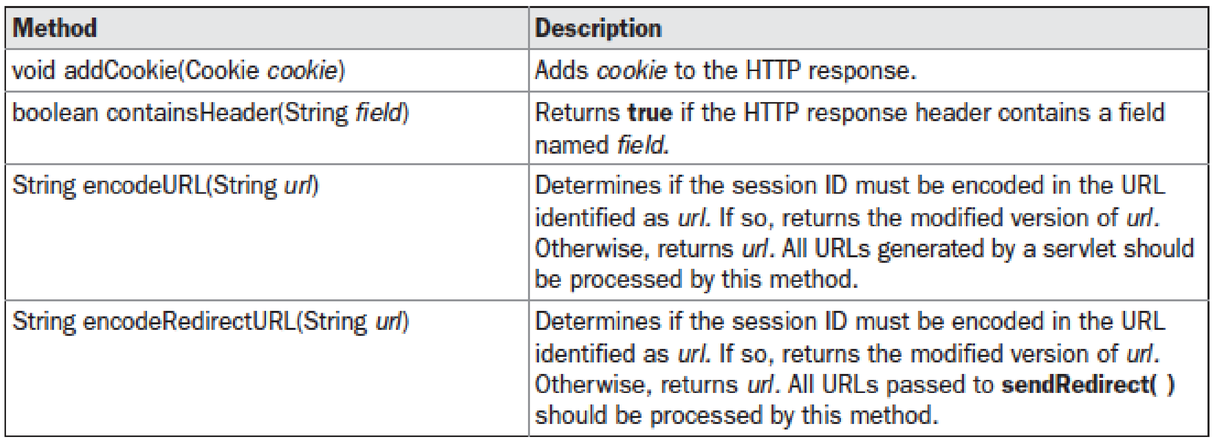


Method	Description
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, return **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

Table 31: Various methods defined by HttpServletRequest

10.5.2 The HttpServletResponse Interface

The ***HttpServletResponse*** interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, SC_OK indicates that the HTTP request succeeded, and SC_NOT_FOUND indicates that the requested resource is not available. Several methods of this interface are summarized in table below.



Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.

Method	Description
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

Table 32: Various methods defined by HttpServletResponse

10.5.3 The HttpSession Interface

The *HttpSession* interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table below. All of these methods throw an *IllegalStateException* if the session has already been invalidated.

Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

Table 33: The methods defined by HttpSession

10.5.4 The HttpSessionBindingListener Interface

The ***HttpSessionBindingListener*** interface is implemented by objects that need to be notified when they are bound to or unbound from an HTTP session. The methods that are invoked when an object is bound or unbound are
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
Here, e is the event object that describes the binding.

10.5.5 The Cookie Class

The Cookie class encapsulates a cookie. A **cookie** is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store. A servlet can write a cookie to a user's machine via the ***addCookie***( ) method of the ***HttpServletResponse*** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.
The names and values of cookies are stored on the user's machine. Some of the information

that is saved for each cookie includes the following:

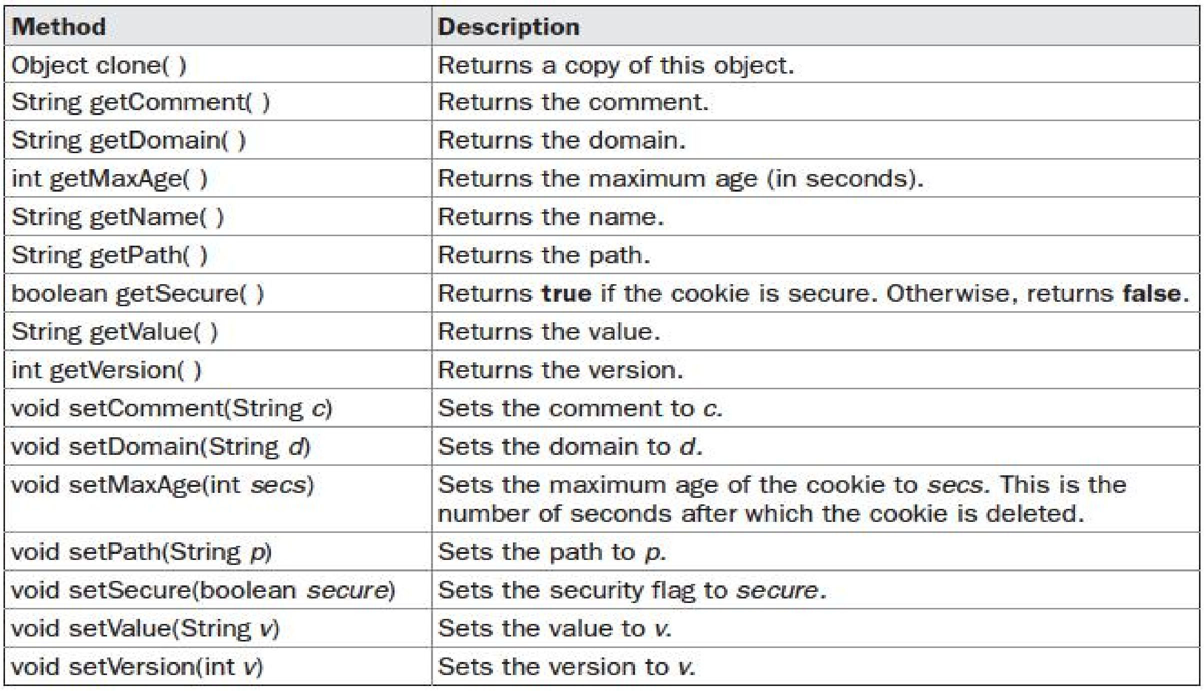
- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user's machine.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server. Otherwise, it is not. There is one constructor for Cookie. It has the signature shown here:

Cookie(String name, String value).

Here, the name and value of the cookie are supplied as arguments to the constructor.



Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

Table 34: Methods defined by Cookie

10.6 Handling HTTP Requests and Responses

The ***HttpServlet*** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are ***doDelete( ), doGet( ), doHead( ), doOptions( ), doPost( ), doPut( )***, and ***doTrace( ).*** The GET and POST requests are commonly used when handling form input.

10.6.1 Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in ***ColorGet.htm***, and a servlet is defined in ***ColorGetServlet.java***. The HTML source code for ***ColorGet.htm*** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
```

```
      <body>
            <center>
            <form name="Form1"action="http://localhost:8080/servlets-
            examples/servlet/ColorGetServlet">
            <B>Color:</B>
            <select name="color" size="1">
                  <option value="Red">Red</option>
                  <option value="Green">Green</option>
                  <option value="Blue">Blue</option>
            </select>
            <br><br>
            <input type=submit value="Submit">
            </form>
      </body>
</html>
```

The source code for *ColorGetServlet.java* is shown in the following listing. The ***doGet( )*** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the ***getParameter( )*** method of ***HttpServletRequest*** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException {
      String color = request.getParameter("color");
      response.setContentType("text/html");
      PrintWriter pw = response.getWriter();
      pw.println("<B>The selected color is: ");
      pw.println(color);
      pw.close();
}
}
```

After completing these steps, the browser will display the response that is dynamically generated by the servlet. One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is
http://localhost:8080/servlets-examples/servlet/ColorGetServlet?color=Red
The characters to the right of the question mark are known as the *query string*.

10.6.2 Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in ***ColorPost.htm***, and a servlet is defined in ***ColorPostServlet.java***.
The HTML source code for ***ColorPost.htm*** is shown in the following listing. It is identical to ***ColorGet.htm*** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
      <body>
      <center>
      <form name="Form1" method="post"
      action="http://localhost:8080/servlets-
```

```
	examples/servlet/ColorPostServlet">
		<B>Color:</B>
		<select name="color" size="1">
		<option value="Red">Red</option>
		<option value="Green">Green</option>
		<option value="Blue">Blue</option>
		</select>
		<br><br>
		<input type=submit value="Submit">
	</form>
	</body>
</html>
```

The source code for *ColorPostServlet.java* is shown in the following listing. The ***doPost( )*** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the ***getParameter***( ) method of *HttpServletRequest* to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorPostServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
	String color = request.getParameter("color");
	response.setContentType("text/html");
	PrintWriter pw = response.getWriter();
	pw.println("<B>The selected color is: ");
	pw.println(color);
	pw.close();
}
}
```

Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is
http://localhost:8080/servlets-examples/servlet/ColorPostServlet.
The parameter names and values are sent in the body of the HTTP request.

10.7 Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.
A session can be created via the ***getSession***( ) method of ***HttpServletRequest***. An ***HttpSession*** object is returned. This object can store a set of bindings that associate names with objects. The ***setAttribute***( ), ***getAttribute( ), getAttributeNames( ),*** and ***removeAttribute( )*** methods of ***HttpSession*** manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

The following servlet illustrates how to use session state. The *getSession*( ) method gets the current session. A new session is created if one does not already exist. The *getAttribute*( ) method is called to obtain the object that is bound to the name “date”. That objects is a Date object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A Date object encapsulating the current date and time is then created. The *setAttribute*( ) method is called to bind the name “date” to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DateServlet extends HttpServlet {
      public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
      IOException {
      // Get the HttpSession object.
      HttpSession hs = request.getSession(true);
      // Get writer.
      response.setContentType("text/html");
      PrintWriter pw = response.getWriter();
      pw.print("<B>");
      // Display date/time of last access.
      Date date = (Date)hs.getAttribute("date");
      if(date != null) {
      pw.print("Last access: " + date + "<br>");
      }
      // Display current date/time.
      date = new Date();
      hs.setAttribute("date", date);
      pw.println("Current date: " + date);
}
}
```

When we first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

10.8 Java Server Page (JSP)

- *Java Server Pages (JSP)* is a technology for developing web pages that support dynamic content which helps developers *insert java code in HTML* pages by making use of special JSP tags, most of which start with <% and end with %>.
- A ***Java Server*** Pages component is a type of *Java servlet* that is designed to fulfill the role of a user interface for a Java web application.
- Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.
- Using JSP, we can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.
- JSP tags can be used for a variety of purposes, such as retrieving information from a database or *registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages* etc.

10.8.1 Why Use JSP?

Java Server Pages often serve the same purpose as programs implemented using the

Common Gateway Interface (CGI). But JSP offers several advantages in comparison with the CGI.

- Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having a separate CGI files.
- JSP are always compiled before it's processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
- Java Server Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP etc.
- JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

- Finally, JSP is an integral part of J2EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

10.8.2 Advantages of JSP:

Following is the list of other advantages of using JSP over other technologies:

- **vs. Active Server Pages (ASP):** The advantages of JSP are two fold. ***First***, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. ***Second***, it is portable to other operating systems and non-Microsoft Web servers.
- **vs. Pure Servlets:** It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.
- **vs. Server-Side Includes (SSI):** SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.
- **vs. JavaScript:** JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.
- **vs. Static HTML:** Regular HTML, of course, cannot contain dynamic information.

10.8.3 JSP Life Cycle

A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

The following are the paths followed by a JSP

- Compilation
- Initialization
- Execution
- Cleanup

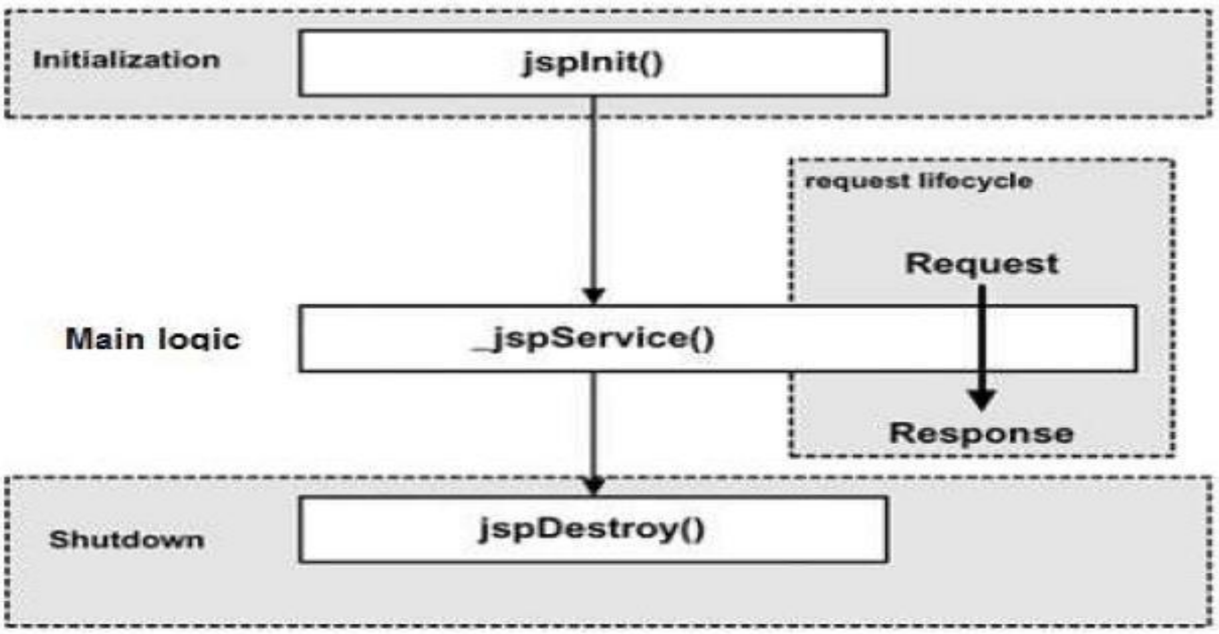


Figure 29: JSP Life Cycle

JSP Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page. The compilation process involves three steps:

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization

When a container loads a JSP it invokes the ***jspInit()*** method before servicing any requests. If we need to perform JSP-specific initialization, override the ***jspInit()*** method:

public void jspInit(){
// Initialization code...
}

Typically initialization is performed only once and as with the servlet ***init*** method, we generally initialize database connections, open files, and create lookup tables in the ***jspInit*** method.

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed. Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP. The ***_jspService()*** method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

void _jspService(HttpServletRequest request,
HttpServletResponse response)
{
// Service handling code...
}

The *_jspService()* method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

JSP Cleanup
The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container. The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. We need to override ***jspDestroy*** to perform any cleanup, such as releasing database connections or closing open files.
The ***jspDestroy***() method has the following form:
public void jspDestroy()
{
// Your cleanup code goes here.
}

10.8.4 JSP Syntax
A JSP Web page is created the same way you make an HTML file, except JSP code is added along with the HTML code. Additionally, instead of naming the file with an extension of **.HTM or .HTML** , the extension is **.JSP** .
The JSP elements are ***declarations***, ***expressions*** , ***scriptlets*** , and ***directives*** . Their own tags identify all of these elements.

10.8.4.1 JSP element declaration
The declarations tag allows us to define ***variables*** and ***methods***. The variables and methods are accessible
from any ***scriptlets*** and ***expressions*** on the same page. Variable declarations are compiled as instance variables for a class that corresponds to the JSP page. Declarations are defined with the syntax
<%!
Declarations
%>
For example, the following defines an instance variable named ***count*** and a method named ***incrementCount*** () that increments the count variable:
```
<%!
      private int count = 0;
      private void incrementCount()
      {
            count++;
      }
%>
```

10.8.4.2 JSP Expression
We can access variables defined in declarations with expression. The syntax to embed an expression is as follows:
<%=
Expression
%>
Expressions are embedded directly into the HTML. The Web browser will display the value of the expression in place of the tag. For example, we can output the value of the ***count*** variable in bold type with the following piece of HTML:
The value of count is <b> <%= count %> </b>

10.8.4.3 JSP Scriptlet
Blocks of Java code can be embedded in a ***scriptlet***. The syntax for a ***scriptlet*** is as follows:

<%
Java code
%>
If you wish to output HTML within a ***scriptlet***, then this is done using ***out.println()***, which is used in the same manner as ***System.out.println( )***.The variable ***out*** is already defined for us and is of type ***javax.servlet.jsp.JspWriter***. Also note that ***System.out.println( )*** will output to the console, which is useful for debugging purposes, while ***out.println( )*** will output to the browser. The following ***scriptlet*** invokes the ***incrementCount( )*** method and then outputs the value in count :

```
<%
out.println("The counter's value is " + count + "<br />");
incrementCount();
%>
```

Following is a JSP page with a ***declaration***, ***expression***, and ***scriptlet*** that outputs text inside a header tag from levels 1 to 6. The identifier LASTLEVEL is declared as the last heading level that is to be displayed. LASTLEVEL is modified by static final, because it is intended as a constant. A loop inside the ***scriptlet*** outputs sample text for each level.

```
<html>
      <title>
       Displaying Heading Tags with JSP
       </title>
       <body>
            <%!
                 private static final int LASTLEVEL = 6;
            %>
            <p>
            This page uses JSP to display Heading Tags from
            Level 1 to Level <%= LASTLEVEL %>
            </p>
            <%
            int i;
            for (i = 1; i <= LASTLEVEL; i++)
            {
            out.println("<H" + i + ">" +
            "This text is in Heading Level " + i +
            "</H" + i + ">");
            }
            %>
       </body>
 </html>
```

10.8.4.4 JSP Directives

Finally, let us introduce one more JSP tag, the ***directive***. In general terms, directives instruct the compiler how to process a JSP program. Examples include the definition of our own tags, including the source code of other files, and importing packages. The syntax for directives is as follows:
<%@
page import="java.util.,java.sql.*"*
%>

10.8.5 Handling Parameters in JSP

To make a JSP page more interactive, we can read and process the data entered in an HTML form. One way to read these values is to call the ***request.getParameter()*** method.This method takes a String parameter as input that identifies the name of an HTML form element and returns the value entered by the user for that element on the form. For example, if there is a *textbox* named *AuthorID*, then we can retrieve the value entered in that textbox with the following ***scriptlet*** code:

String value = request.getParameter("AuthorID");

If the user leaves the field blank, then *getParameter* returns an empty string. The following Html file uses a EditURL.jsp in order to process request.

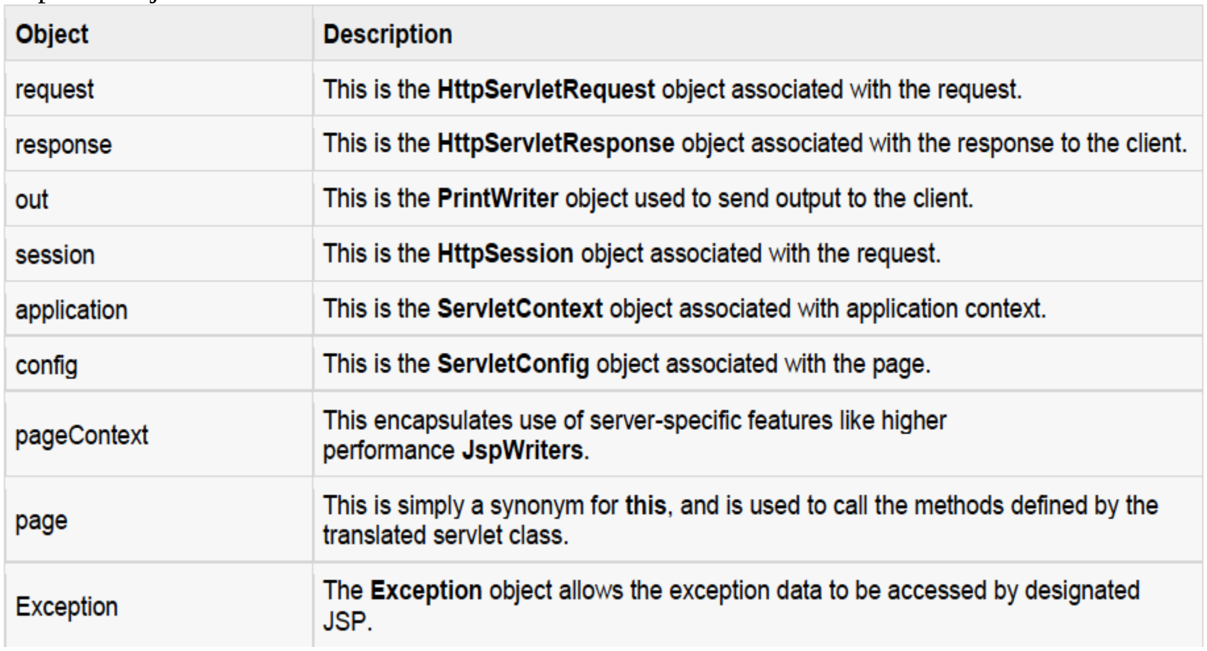
```
<html>
      <head>
            <title>Change Author's URL</title>
      </head>
      <body>
            <h1>Change Author's URL</h1>
            <p>
            Enter the ID of the author you would like to change
            along with the new URL.
            </p>
            <form ACTION = "EditURL.jsp" METHOD = POST>
                  Author ID:
                  <input TYPE = "TEXT" NAME = "AuthorID" VALUE = ""
                  SIZE = "4" MAXLENGTH = "4">
                  <br/>
                  New URL:
                  <input TYPE = "TEXT" NAME = "URL"
                  VALUE = "http://" SIZE = "40" MAXLENGTH = "200">
                  <p>
                  INPUT TYPE="SUBMIT" VALUE="Submit">
                  </p>
            </form>
      </body>
</html>
```

The following JSP program echoes back the data entered by the user in above html. The name of the JSP file must match the value supplied for the ACTION tag of the form.

```
<html>
      <title>Edit URL: Echo submitted values</title>
      <body>
            <h2>Edit URL>/h2>
            <p>
            This version of EditURL.jsp simply echoes back to the
            user the values that were entered in the textboxes.
            </p>
            <%
            String url = request.getParameter("URL");
            String stringID = request.getParameter("AuthorID");
            int author_id = Integer.parseInt(stringID);
            out.println("The submitted author ID is: " + author_id);
            out.println("<br/>");
            out.println("The submitted URL is: " + url);
            %>
      </body>
</html>
```

10.8.6 JSP Implicit Objects

JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables. JSP supports nine Implicit Objects which are listed below:



Object	Description
request	This is the **HttpServletRequest** object associated with the request.
response	This is the **HttpServletResponse** object associated with the response to the client.
out	This is the **PrintWriter** object used to send output to the client.
session	This is the **HttpSession** object associated with the request.
application	This is the **ServletContext** object associated with application context.
config	This is the **ServletConfig** object associated with the page.
pageContext	This encapsulates use of server-specific features like higher performance **JspWriters**.
page	This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
Exception	The **Exception** object allows the exception data to be accessed by designated JSP.

1. The ***request object*** is an instance of a ***javax.servlet.http.HttpServletRequest*** object. Each time a client requests a page the JSP engine creates a new object to represent that request. The *request* object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.
2. The ***response object*** is an instance of a ***javax.servlet.http.HttpServletResponse*** object. Just as the server creates the request object, it also creates an object to represent the response to the client. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.
3. The ***out*** implicit object is an instance of a ***javax.servlet.jsp.JspWriter*** object and is used to send content in a response. The initial ***JspWriter*** object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the buffered='false' attribute of the page directive. The ***JspWriter*** object contains most of the same methods as the ***java.io.PrintWriter*** class. However, ***JspWriter*** has some additional methods designed to deal with buffering. Unlike the *PrintWriter* object, *JspWriter* throws *IOExceptions*.
4. The ***session object*** is an instance of ***javax.servlet.http.HttpSession*** and behaves exactly the same way that session objects behave under Java Servlets. The session object is used to track client session between client requests.
5. The ***application object*** is direct wrapper around the ***ServletContext*** object for the generated Servlet and in reality an instance of a ***javax.servlet.ServletContext*** object. This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the ***jspDestroy***() method.
6. The ***config object*** is an instantiation of ***javax.servlet.ServletConfig*** and is a direct wrapper around the ***ServletConfig*** object for the generated servlet. This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.
7. The ***pageContext*** object is an instance of a ***javax.servlet.jsp.PageContext*** object. The ***pageContext*** object is used to represent the entire JSP page. This object is intended as a means to access information about the page while avoiding most of the implementation

details. This object stores references to the request and response objects for each request. The *application*, *config*, *session*, and *out* objects are derived by accessing attributes of this object.

8. ***Page*** object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The page object is really a direct synonym for the **this** object.
9. The ***exception*** object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

10.8.7 JavaServer Pages Standard Tag Library (JSTL)

JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications. JSTL has support for *common, structural tasks* such as *iteration and conditionals*, *tags for manipulating XML documents*, *internationalization tags*, and *SQL tags*. It also provides a framework for integrating existing custom tags with JSTL tags.

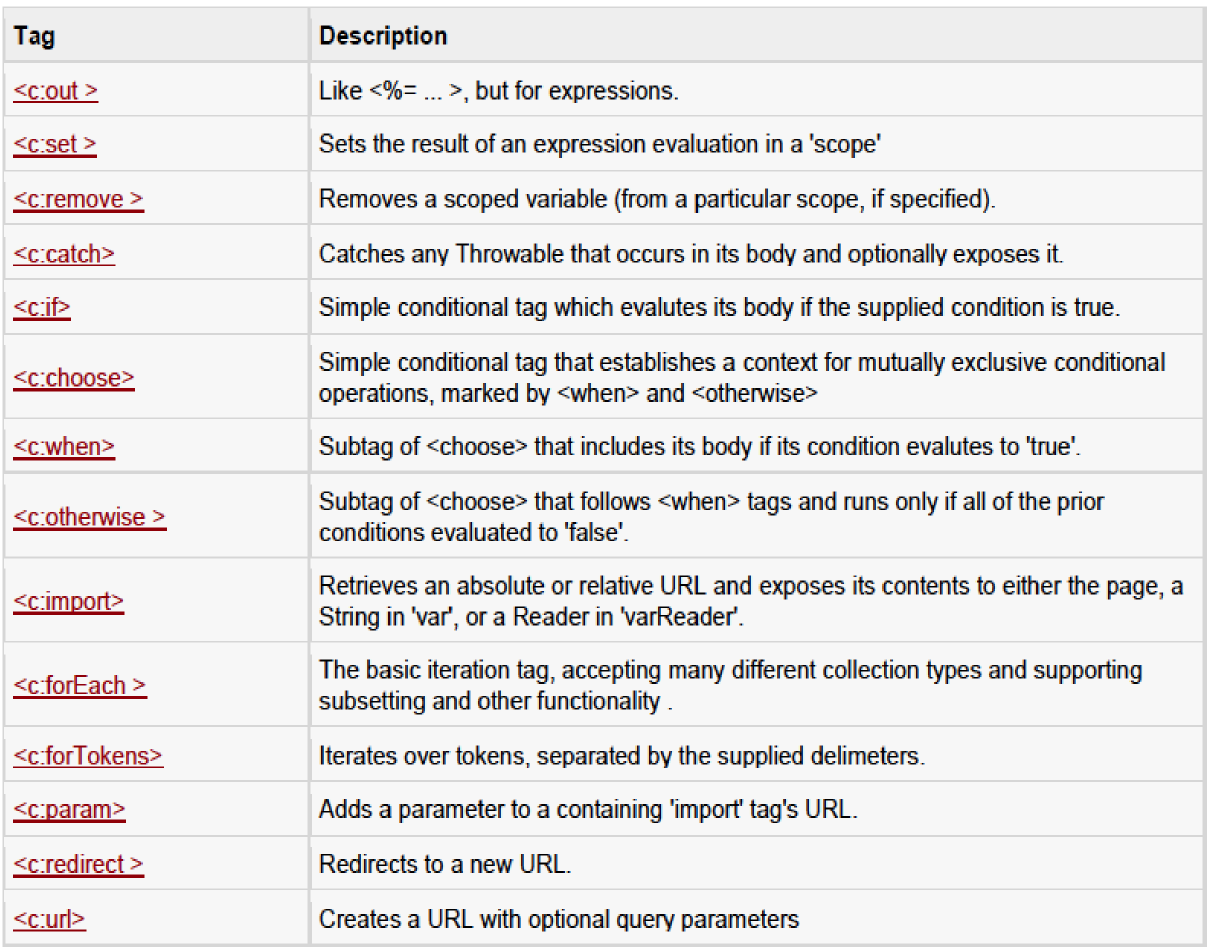
The JSTL tags can be classified, according to their functions, into following JSTL tag library groups that can be used when creating a JSP page:

- Core Tags
- Formatting tags
- SQL tags
- XML tags
- JSTL Functions

10.8.7.1 Core Tags

The core group of tags is the most frequently used JSTL tags. Following is the syntax to include JSTL Core library in our JSP:

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

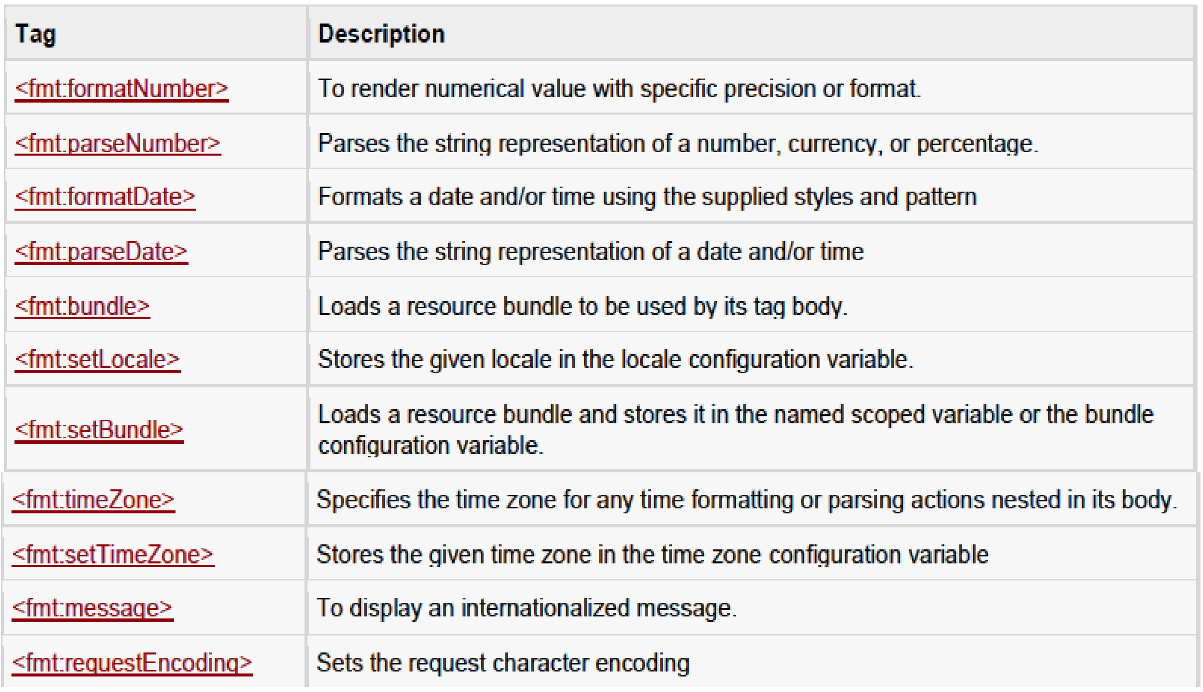


Tag	Description
<c:out >	Like <%= ... >, but for expressions.
<c:set >	Sets the result of an expression evaluation in a 'scope'
<c:remove >	Removes a scoped variable (from a particular scope, if specified).
<c:catch>	Catches any Throwable that occurs in its body and optionally exposes it.
<c:if>	Simple conditional tag which evalutes its body if the supplied condition is true.
<c:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<c:when>	Subtag of <choose> that includes its body if its condition evalutes to 'true'.
<c:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'.
<c:import>	Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
<c:forEach >	The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
<c:forTokens>	Iterates over tokens, separated by the supplied delimeters.
<c:param>	Adds a parameter to a containing 'import' tag's URL.
<c:redirect >	Redirects to a new URL.
<c:url>	Creates a URL with optional query parameters

10.8.7.2 Formatting tags

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites. Following is the syntax to include formatting library in our JSP:

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>



Tag	Description
<fmt:formatNumber>	To render numerical value with specific precision or format.
<fmt:parseNumber>	Parses the string representation of a number, currency, or percentage.
<fmt:formatDate>	Formats a date and/or time using the supplied styles and pattern
<fmt:parseDate>	Parses the string representation of a date and/or time
<fmt:bundle>	Loads a resource bundle to be used by its tag body.
<fmt:setLocale>	Stores the given locale in the locale configuration variable.
<fmt:setBundle>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
<fmt:timeZone>	Specifies the time zone for any time formatting or parsing actions nested in its body.
<fmt:setTimeZone>	Stores the given time zone in the time zone configuration variable
<fmt:message>	To display an internationalized message.
<fmt:requestEncoding>	Sets the request character encoding

10.8.7.3 SQL tags

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server. Following is the syntax to include JSTL SQL library in our JSP:

<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

Tag	Description
<sql:setDataSource>	Creates a simple DataSource suitable only for prototyping
<sql:query>	Executes the SQL query defined in its body or through the sql attribute.
<sql:update>	Executes the SQL update defined in its body or through the sql attribute.
<sql:param>	Sets a parameter in an SQL statement to the specified value.
<sql:dateParam>	Sets a parameter in an SQL statement to the specified java.util.Date value.
<sql:transaction >	Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

10.8.7.4 XML tags

The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in our JSP. The JSTL XML tag library has custom tags for interacting with XML data. This includes parsing XML, transforming XML data, and flow control based on XPath expressions.

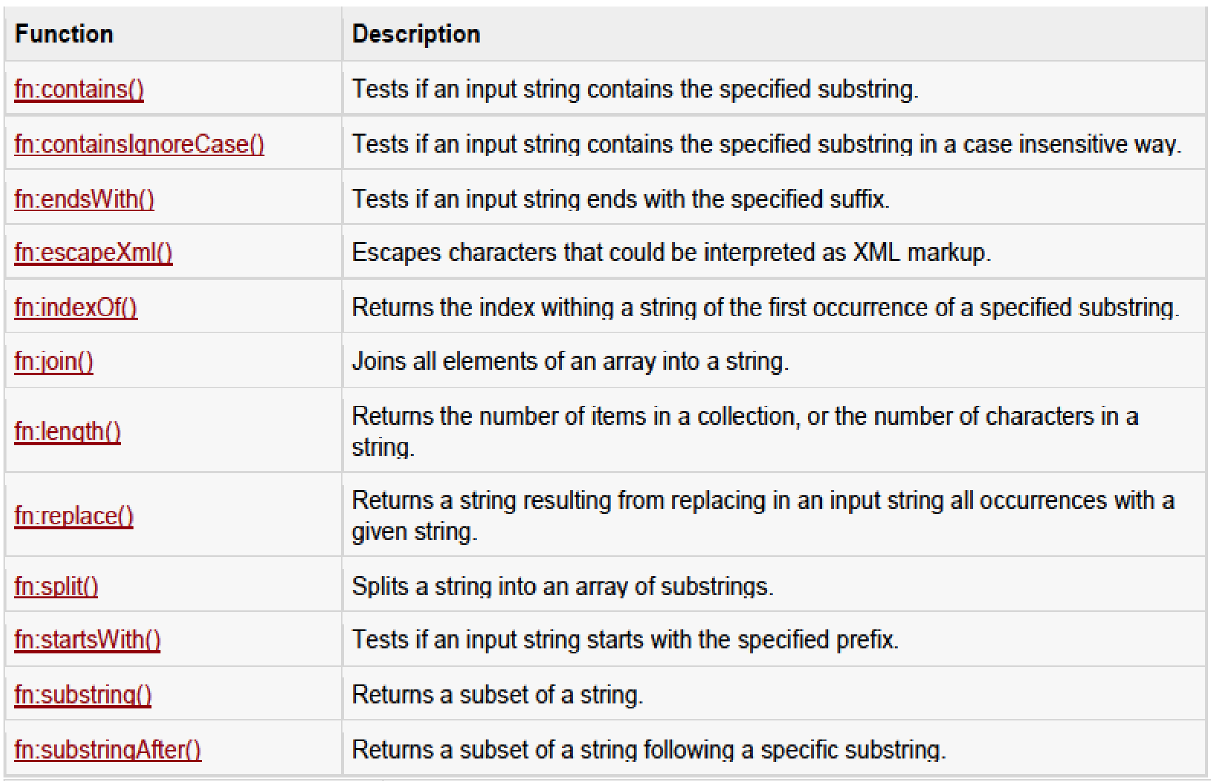
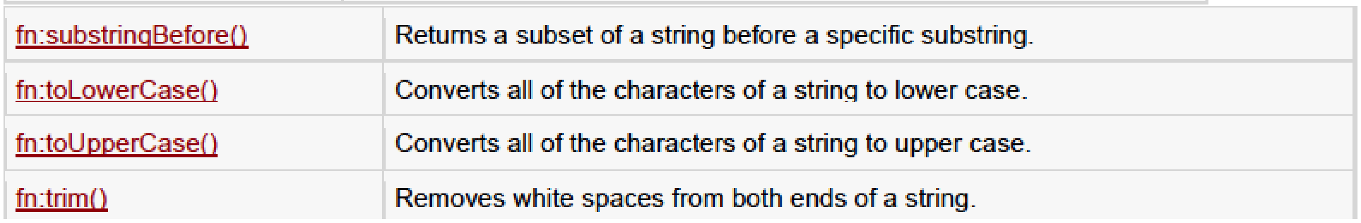
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

Tag	Description
<x:out>	Like <%= ... >, but for XPath expressions.
<x:parse>	Use to parse XML data specified either via an attribute or in the tag body.
<x:set >	Sets a variable to the value of an XPath expression.
<x:if >	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<x:forEach>	To loop over nodes in an XML document.
<x:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<x:when >	Subtag of <choose> that includes its body if its expression evalutes to 'true'
<x:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'
<x:transform >	Applies an XSL transformation on a XML document
<x:param >	Use along with the transform tag to set a parameter in the XSLT stylesheet

10.8.7.5 JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in our JSP:

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

Function	Description
fn:contains()	Tests if an input string contains the specified substring.
fn:containsIgnoreCase()	Tests if an input string contains the specified substring in a case insensitive way.
fn:endsWith()	Tests if an input string ends with the specified suffix.
fn:escapeXml()	Escapes characters that could be interpreted as XML markup.
fn:indexOf()	Returns the index withing a string of the first occurrence of a specified substring.
fn:join()	Joins all elements of an array into a string.
fn:length()	Returns the number of items in a collection, or the number of characters in a string.
fn:replace()	Returns a string resulting from replacing in an input string all occurrences with a given string.
fn:split()	Splits a string into an array of substrings.
fn:startsWith()	Tests if an input string starts with the specified prefix.
fn:substring()	Returns a subset of a string.
fn:substringAfter()	Returns a subset of a string following a specific substring.
fn:substringBefore()	Returns a subset of a string before a specific substring.
fn:toLowerCase()	Converts all of the characters of a string to lower case.
fn:toUpperCase()	Converts all of the characters of a string to upper case.
fn:trim()	Removes white spaces from both ends of a string.

11 Chapter 11: Database Handling

11.1 Introduction

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for ***storing***, ***organizing***, ***retrieving*** and ***modifying data*** for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data. Some popular relational database management systems (RDBMSs) are Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL. The JDK now comes with a pure-Java RDBMS called Java DB-Oracles's version of Apache Derby.

Java programs communicate with databases and manipulate their data using the *Java Database Connectivity (JDBC)* API. A *JDBC* driver enables Java applications to connect to a database in a particular DBMS and allows us to manipulate that database using the *JDBC API*. Most popular database management systems now provide *JDBC drivers*. There are also many third-party JDBC drivers available.

The JDBC consists of two layers. The top layer is the *JDBC API*. This API communicates with the *JDBC manager driver API*, sending it the various SQL statements. The manager should communicate with the various third-party drivers that actually connect to the database and return the information from the query or perform the action specified by the query.

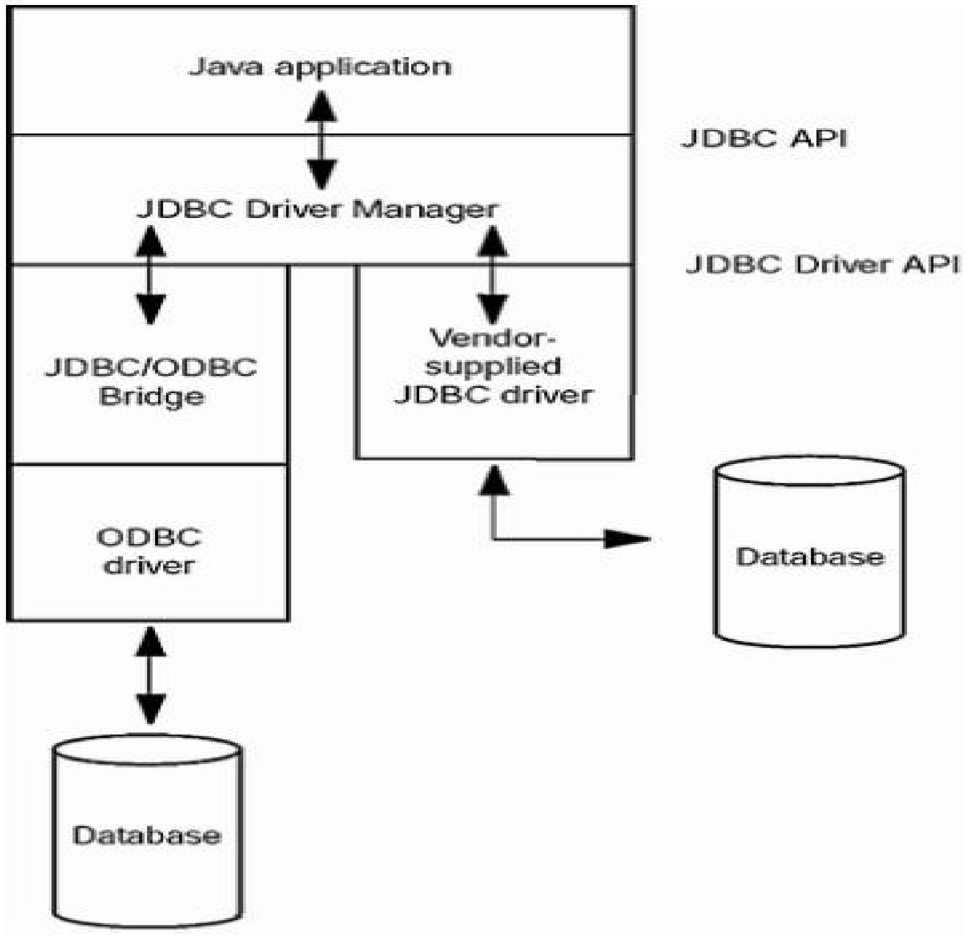


Figure 30: JDBC-to-database communication path

11.2 JDBC Driver

JDBC is: Java Database Connectivity

- is a Java API for connecting programs written in Java to the data in relational databases.
- consists of a set of classes and interfaces written in the Java programming language.
- provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.
- The standard defined by Sun Microsystems, allowing individual providers to implement and
  extend the standard with their own JDBC drivers.
- The ***Java.sql*** package contains various classes with their behaviours defined and their actual implementations are done in third-party drivers.
- Third party vendors implements the ***java.sql.Driver*** interface in their database driver.

JDBC:

- establishes a connection with a database
- sends SQL statements
- processes the results.

11.2.1 JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below.

11.2.1.1 Type 1: JDBC-ODBC Bridge Driver

In a **Type 1 driver**, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on our system a Data Source Name (DSN) that represents the target database. When Java first came out, this was a useful driver because most databases only supported ODBC access but now this ***type of driver is recommended only for experimental use or when no other alternative is available***.

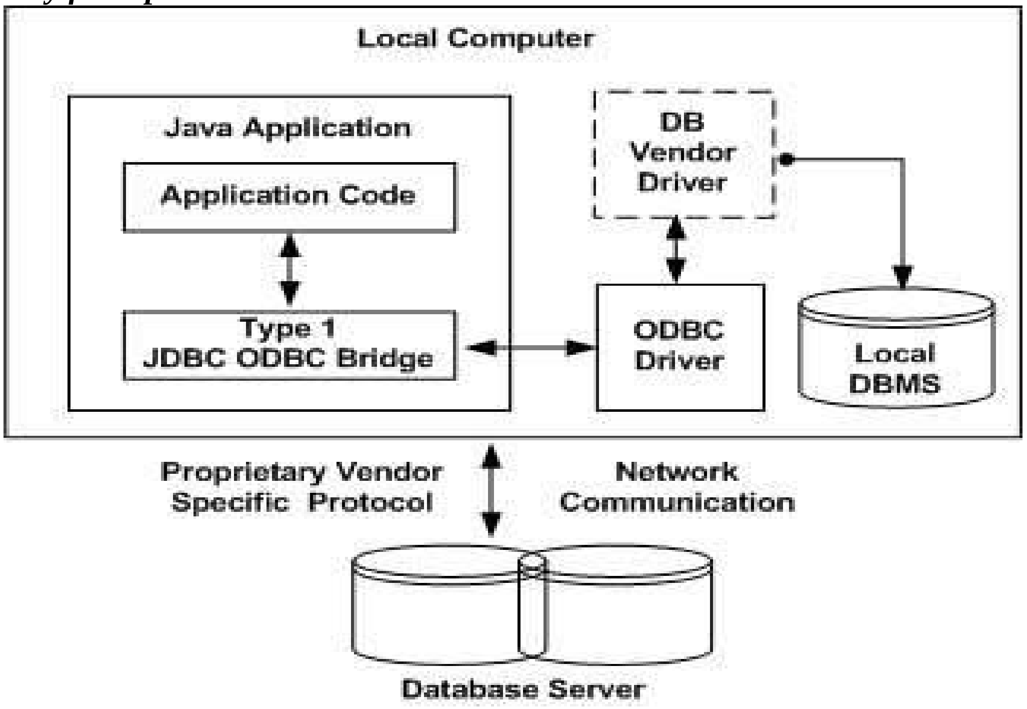


Figure 31: Type 1: JDBC-ODBC Bridge Driver

The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

11.2.1.2 Type 2: JDBC-Native API

In a **Type 2 driver**, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine. If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but we may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

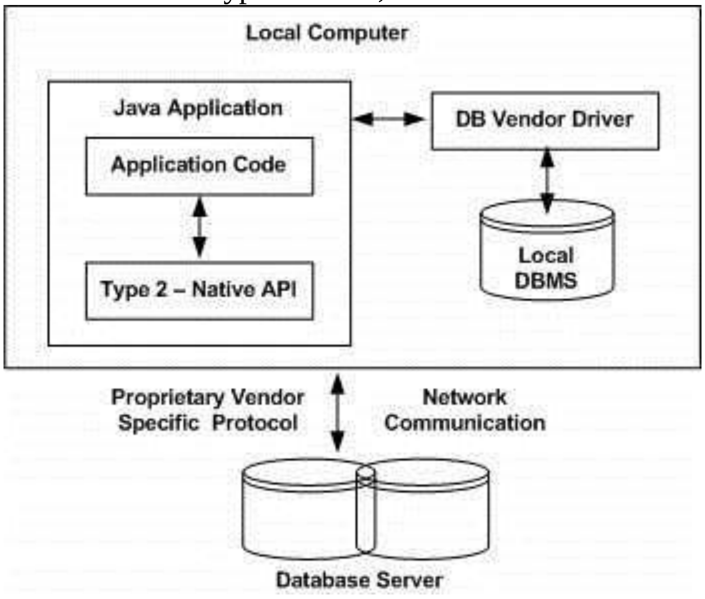


Figure 32: Type 2: JDBC-Native API

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

11.2.1.3 Type 3: JDBC-Net pure Java

In a **Type 3 driver**, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server. This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

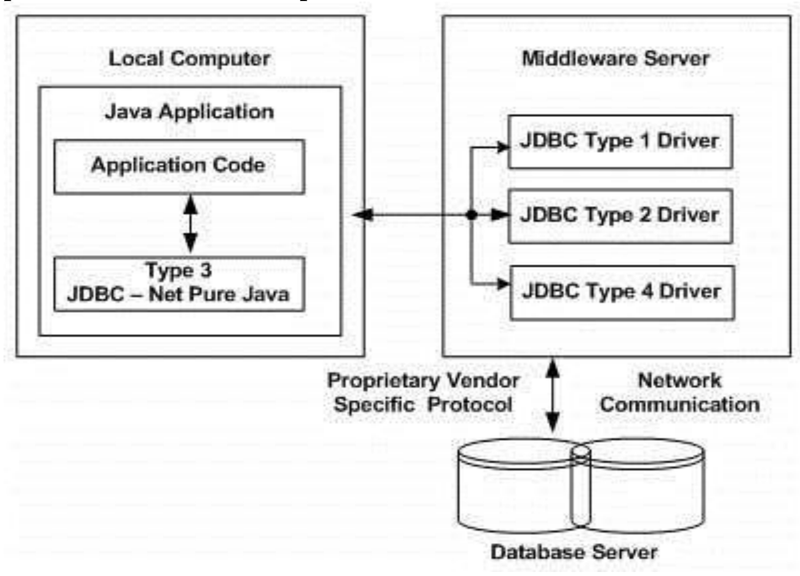


Figure 33: Type 3: JDBC-Net pure Java

We can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, we need some knowledge of the application server's configuration in order to effectively use this driver type.

11.2.1.4 Type 4: 100% Pure Java

In a **Type 4 driver**, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself. This kind of driver is extremely flexible; we don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

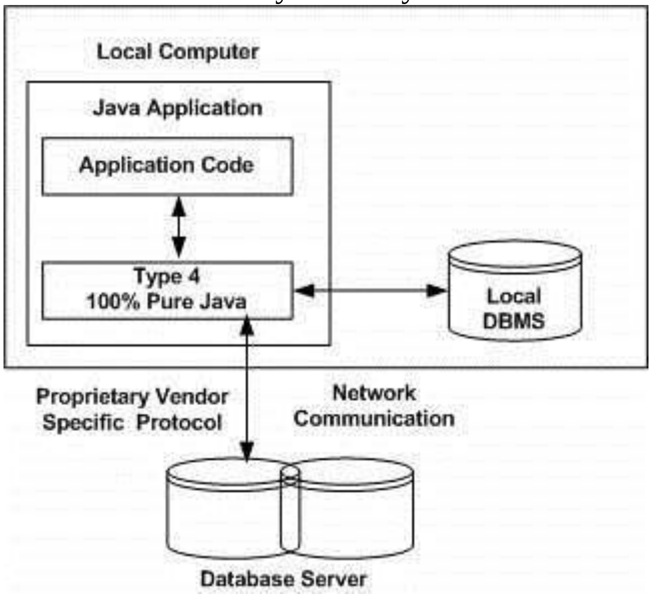


Figure 34: Type 4: 100% Pure Java

MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

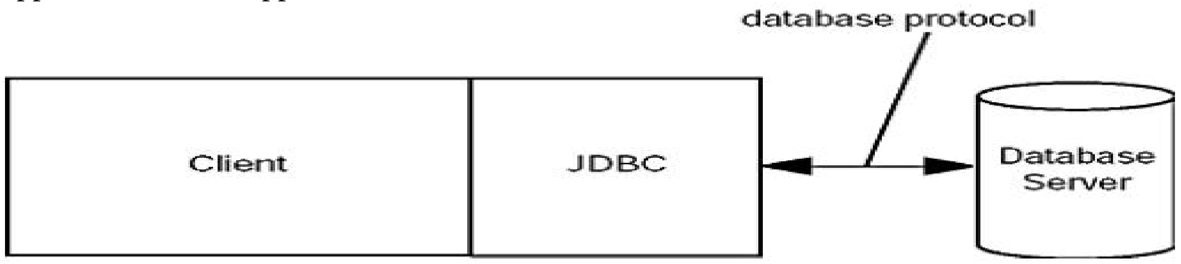
Which Driver should be Used?

- If we are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If our Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for our database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

11.3 Typical Uses of JDBC

We can use JDBC in both applications and applets. In an applet, all the normal security restrictions apply. By default, the security manager assumes that all applets written in the Java programming language are untrusted.
In particular, applets that use JDBC are only able to open a database connection to the server from which they are downloaded. That means the Web server and the database server must be the same machine, which is not a typical setup. Of course, the Web server can have a proxy service that routes database traffic to another machine. With signed applets, this restriction can be loosened.

Applications, on the other hand, have complete freedom to access remote database servers. If we implement a traditional client/server program, it probably makes more sense to use an application, not an applet, for database access.



However, the world is moving away from client/server and toward a "three-tier model" or even more advanced "n-tier models." In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates **visual presentation** (on the client) from the **business logic** (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when we use a web browser as the client), RMI (when we use an application or applet), or another mechanism. *JDBC is used to manage the communication between the middle tier and the back-end database*. Following figure shows the basic architecture.

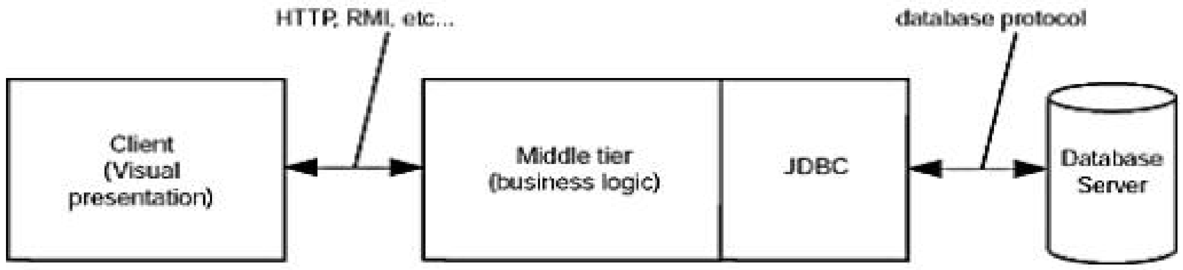


Fig: 3-tier Application

SQL is the industry-standard approach to accessing relational databases. JDBC supports SQL, enabling developers to use a wide range of database formats without knowing the specifics of the underlying database. JDBC also supports the use of database queries specific to a database format.

The JDBC class library's approach to accessing databases with SQL is comparable to existing database-development techniques, so interacting with an SQL database by using JDBC isn't much different than using traditional database tools. Java programmers who already have some database experience can hit the ground running with JDBC. The JDBC library includes classes for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating a statement using SQL
- Executing that SQL query in the database
- Viewing the resulting records

These JDBC classes are all part of the *java.sql* package.

11.4 Database Drivers

Java programs that use JDBC classes can follow the familiar programming model of issuing

SQL statements and processing the resulting data. The format of the database and the platform it was prepared on don't matter.

A driver manager makes the platform and database independence possible. The classes of the *JDBC class library are largely dependent on driver managers,* which keep track of the drivers required to access database records. We'll need a different driver for each database format that's used in a program, and sometimes we might need several drivers for versions of the same format. *Java DB includes its own driver. JDBC also includes a driver that bridges JDBC and another database-connectivity standard, ODBC.*

The JDBC-ODBC bridge allows JDBC drivers to be used as ODBC drivers by converting JDBC method calls into ODBC function calls.
Using the JDBC-ODBC bridge requires three things:

- The JDBC-ODBC bridge driver included with Java: *sun.jdbc.odbc.JdbcOdbcDriver*
- An ODBC driver
- An ODBC data source that has been associated with the driver using software such as the ODBC Data Source Administrator.

11.5 Manipulating Databases with JDBC

Steps Required to access Database using JDBC
There are following steps required to access Database using JDBC application:

1. **Import the packages:** Requires that we need to include the packages containing the JDBC classes needed for database programming. Most often, using ***import java.sql.**** will suffice.
2. **Register the JDBC driver:** Requires that we initialize a driver so we can open a communications channel with the database.
3. **Open a connection:** Requires using the ***DriverManager.getConnection()*** method to create a Connection object, which represents a physical connection with a **selected** database.
4. **Creating a Statement Object:** Create the required queries.
5. **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to perform operations in a table.
6. **Processing the Result Set**
7. **Closing the Result Set and Statement Objects**
8. **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

11.5.1 Connecting to and Querying a Database

The following section of program performs a simple query on the *books* database that retrieves the entire *Authors* table and displays the data. The program illustrates *connecting to the database, querying the database and processing the result*. The discussion that follows presents the key JDBC aspects of the program.

DisplayAuthors.java

// Displaying the contents of the authors table.

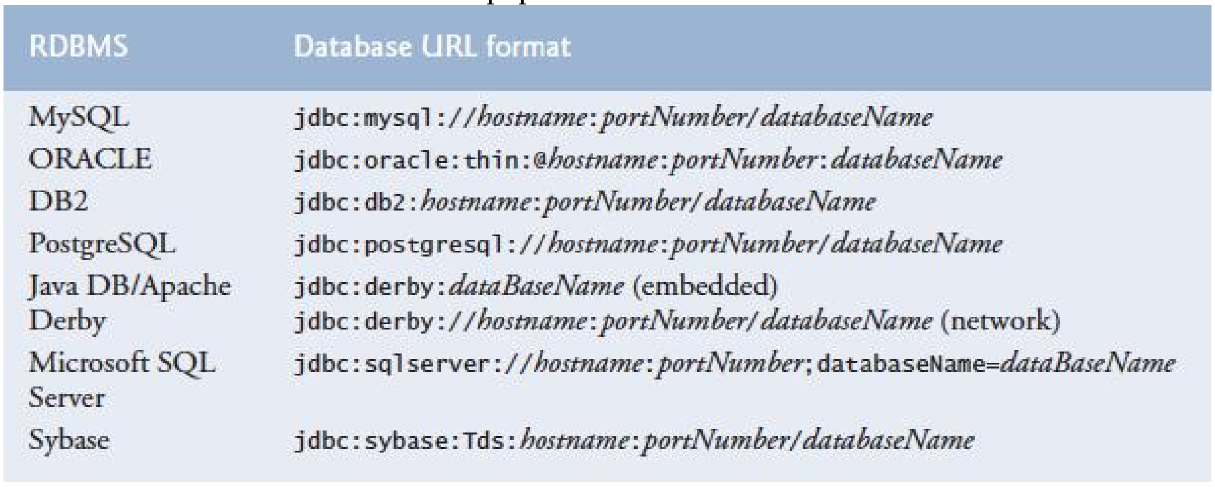
```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
```

```
public class DisplayAuthors
{
   // database URL
   static        final        String        DATABASE_URL        =
  "jdbc:mysql://localhost:3306/books";

   // launch the application
   public static void main( String args[] )
   {
      Connection connection = null; // manages connection
      Statement statement = null; // query statement
      ResultSet resultSet = null; // manages results

      // connect to database books and query database
      try
      {
         // establish connection to database
         connection = DriverManager.getConnection(
            DATABASE_URL, "root", "" );

         // create Statement for querying database
         statement = connection.createStatement();

         // query database
         resultSet = statement.executeQuery(
            "SELECT authorID, firstName, lastName FROM authors");

         // process query results
         ResultSetMetaData metaData = resultSet.getMetaData();
         int numberOfColumns = metaData.getColumnCount();
         System.out.println( "Authors Table of Books Database:\n"
  );

         for ( int i = 1; i <= numberOfColumns; i++ )
            System.out.printf( "%-8s\t", metaData.getColumnName( i
  ) );
         System.out.println();

         while ( resultSet.next() )
         {
            for ( int i = 1; i <= numberOfColumns; i++ )
               System.out.printf( "%-8s\t", resultSet.getObject( i
  ) );
            System.out.println();
         } // end while
      }  // end try
      catch ( SQLException sqlException )
      {
         sqlException.printStackTrace();
      } // end catch
      finally // ensure resultSet, statement and connection are
  closed
      {
         try
         {
            resultSet.close();
            statement.close();
            connection.close();
         } // end try
         catch ( Exception exception )
         {
            exception.printStackTrace();
         } // end catch
      } // end finally
   } // end main
} // end class DisplayAuthors
```

In this program there is an import the JDBC interfaces and classes from package ***java.sql*** is used and the declaration of a string constant for the *database URL*. This identifies the name of the database to connect to, as well as information about the *protocol* used by the JDBC driver. Method main connects to the *books* database, queries the database, displays the result of the query and closes the database connection.

11.5.2 Connecting to the Database

In an above program *connection=DriverManager.getConnection(DATABASE_URL,"root","");* create a *Connection* object (package *java.sql*) referenced by *connection*. An object that implements interface ***Connection*** manages the connection between the *Java program and the database*. Connection objects enable programs to create SQL statements that manipulate databases. The program initializes connection with the result of a call to static method ***getConnection*** of class ***DriverManager*** (package ***java.sql***), which attempts to connect to the database specified by its URL. Method *getConnection*() takes three arguments-a String that specifies the *database URL*, a String that specifies the *username* and a String that specifies the *password*. The URL locates the database (possibly on a network or in the local file system of the computer). The URL ***jdbc:mysql://localhost:3306/books*** specifies the protocol for communication (**jdbc**), the sub protocol for communication (***mysql***) and the location of the database ***(//localhost:3306/books***, where *localhost* is the host running the MySQL server , *3306* is default *mysql* running port and ***books*** is the database name). The sub protocol *mysql* indicates that the program uses a MySQL-specific sub protocol to connect to the MySQL database. If the *DriverManager* cannot connect to the database, method *getConnection* throws a *SQLException* (package *java.sql*). Following Figure lists the JDBC driver names and database URL formats of several popular RDBMSs.



RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Table 35: Popular JDBC database URL formats

11.5.3 Creating a *Statement* for Executing Queries

A *Connection* method ***createStatement*** to obtain an object that implements interface ***Statement*** (package ***java.sql***). The program uses the ***Statement*** object to submit SQL statements to the database.

Type of Statements

There are 3 different kinds of statements.

I) Statement: Used to implement simple SQL statements with no parameters.

Eg: *statement = connection.createStatement();*

II) PreparedStatement: (Extends *Statement*.) Used for precompiling SQL statements that might contain input parameters. Eg:

```
PreparedStatement pstmt = null;
String SQL = " SELECT authorID, firstName, lastName FROM authors ";
pstmt = conn.prepareStatement(SQL);
```

III) CallableStatement: (Extends *PreparedStatement*.) Used to execute stored procedures that may contain both input and output parameters.

Suppose we have a store procedure *getAuthor* in MySql to retrieve authors.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS getAuthor $$
CREATE PROCEDURE getAuthor
  (OUT authorID int, OUT FirstName VARCHAR(255), OUT LastName VARCHAR(255))
BEGIN
  SELECT authorID, firstName, lastName FROM authors;
END $$
```

In order to call those store procedure in program.

```
CallableStatement cs = null;
cs = this.conn.prepareCall("{call getAuthor()}");
ResultSet rs = cs.executeQuery();
```

Three types of parameters exist: IN, OUT, and INOUT

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. We bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. We retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. We bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

11.5.4 Executing a Query

In above program use the *Statement* object's ***executeQuery*** method to submit a query that selects all the author information from table ***Authors***. This method returns an object that implements interface ***ResultSet*** and contains the query results. The ***ResultSet*** methods enable the program to manipulate the query result.

- execute: Returns true if the first object that the query returns is a ***ResultSet*** object. Use this method if the query could return one or more *ResultSet* objects. Retrieve the *ResultSet* objects returned from the query by repeatedly calling *Statement.getResultSet*.
- executeQuery: Returns one ResultSet object.
- executeUpdate: Returns an integer representing the number of rows affected by the SQL statement. Use this method if we are using INSERT,DELETE, or UPDATE SQL statements.

11.5.5 Processing a Query's *ResultSet*

An example of processing of Query ***ResultSet*** is shown in above. First of all it needs to obtains the metadata for the ***ResultSet*** as a ***ResultSetMetaData*** (package ***java.sql***) object. The ***metadata*** describes the ***ResultSet***'s contents. Programs can use metadata programmatically to obtain information about the ***ResultSet***'s column names and types. ResultSetMetaData method ***getColumnCount*** are use to retrieve the number of columns in the ***ResultSet***. First *for loop* is used to display the column names while second *for loop* display the data in each *ResultSet* row. First, the program positions the *ResultSet* cursor (which points to the row being processed) to the first row in the *ResultSet* with method *next*. Method *next* returns *boolean* value *true* if it's able to position to the next row; otherwise, the method returns *false*.

If there are rows in the *ResultSet*, second for loop extract and display the contents of each column in the current row. When a *ResultSet* is processed, each column can be extracted as a specific Java type. In fact, *ResultSetMetaData* method ***getColumnType*** returns a constant integer from class **Types** (package *java.sql*) indicating the type of a specified column. Programs can use these values in a *switch* statement to invoke *ResultSet* methods that return the column values as appropriate Java types. If the type of a column is *Types.INTEGER*, *ResultSet* method *getInt* returns the column value as an *int*. *ResultSet* get methods typically receive as an argument either a column number (as an *int*) or a column name (as a *String*) indicating which column's value to obtain.

For simplicity, this example treats each value as an *Object*. We retrieve each column value with *ResultSet* method ***getObject*** then print the *Object's String* representation. Unlike array indices, *ResultSet* column numbers start at 1. The *finally* block closes the *ResultSet*, the *Statement* and the database *Connection*. *NullPointerException* will throw if the *ResultSet*, *Statement* or *Connection* objects were not created properly.

We can also access the data in a ***ResultSet*** object through a cursor. This cursor is not a database cursor. This cursor is a pointer that points to one row of data in the *ResultSet* object. Initially, the cursor is positioned before the first row. We call various methods defined in the *ResultSet* object to move the cursor.

```
try {
stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
int ID = resultSet.getInt("AuthorID");
          String Fname = resultSet.getString("FirstName");
          String Lname = resultSet.getString("LastName");
            //System.out.println();
             System.out.println(ID +"\t\t" + Fname +"\t\t"+ Lname);
         } // end while
      }  // end try
      catch ( SQLException sqlException )
      {
         sqlException.printStackTrace();
      }
```

Another Example:
//STEP 1. Import required packages

```
import java.sql.*;
public class JDBCExample {
      // JDBC driver name and database URL
      static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
      static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";
      // Database credentials
      static final String USER = "username";
      static final String PASS = "password";
      public static void main(String[] args) {
            Connection conn = null;
            Statement stmt = null;
            try{
                  //STEP 2: Register JDBC driver
                  Class.forName("com.mysql.jdbc.Driver");

                  //STEP 3: Open a connection
                  System.out.println("Connecting     to     a     selected
            database...");
                  conn = DriverManager.getConnection(DB_URL, USER, PASS);
                  System.out.println("Connected database successfully...");

                  System.out.println("Creating     table     in     given
            database...");

                  //STEP 4: Creating the statement
                  stmt = conn.createStatement();
                  String sql = "CREATE TABLE REGISTRATION " +
                              "(id INTEGER not NULL, " +
                              " first VARCHAR(255), " +
                              " last VARCHAR(255), " +
                              " age INTEGER, " +
                              " PRIMARY KEY ( id ))";

                  //STEP 5: Execute a query
                  stmt.executeUpdate(sql);
                  System.out.println("Created table in given database...");

                  System.out.println("Inserting     records     into     the
            table...");
                  stmt = conn.createStatement();

      String sql = "INSERT INTO Registration VALUES (100, 'Zara', 'Ali',
      18)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES (101, 'Mahnaz', 'Fatma', 25)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES (102, 'Zaid', 'Khan', 30)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES(103, 'Sumit', 'Mittal', 28)";
      stmt.executeUpdate(sql);

      System.out.println("Inserted records into the table...");
                  }catch(SQLException se){
                        //Handle errors for JDBC
                        se.printStackTrace();
                  }catch(Exception e){
                        //Handle errors for Class.forName
                        e.printStackTrace();
                  }finally{
                        //Step6: Clean up the environment
                        try{
                              if(stmt!=null)
                                    stmt.close();
                        }catch(SQLException se){
                  }// do nothing
                  try{
                        if(conn!=null)
                              conn.close();
                        }catch(SQLException se){
                              se.printStackTrace();
                        }//end finally try
                  }//end try
            System.out.println("Goodbye!");
      }//end main
}//end JDBCExample
```

11.6ODBC (Open Database Connectivity)

ODBC is

- A standard or open application programming interface (API) for accessing a database.
- SQL Access Group, chiefly Microsoft, in 1992
- By using ODBC statements in a program, we can access files in a number of different databases, including **Access, dBase, DB2, Excel**, and **Text**.
- It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases.
- ODBC handles the SQL request and converts it into a request the individual database system understands.
- we need the ODBC software, and a separate module or driver for each database to be accessed. Library that is dynamically connected to the application.
- Driver masks the heterogeneity of DBMS operating system and network protocol.
- E.g. (Sybase, Windows/NT, Novell driver)

11.6.1 JDBC vs ODBC

JDBC	ODBC
JDBC is object oriented.	ODBC is procedural oriented.
JDBC is used to provide connection between JAVA and database(oracle,sybase,DB2,ms-access).	ODBC is used to provide connection between front-end application (other than java) and back-end (database like ms-access)
JDBC is for java applications.	ODBC is for Microsoft
JDBC is used by Java programmers to connect to databases	ODBC is used between applications of different types.
JDBC allows SQL-based database access for EJB persistence and for direct manipulation from CORBA, DJB or other server objects	With a small "bridge" program, we can use the JDBC interface to access ODBC accessible databases.
JDBC is multi-threaded	ODBC is not multi-threaded
We can do everything with JDBC that we can do with ODBC, on any platform.	ODBC can't be directly used with Java because it uses a C interface.

	ODBC makes use of pointers which have been removed totally from java.
JDBC is faster	ODBC is slower
JDBC API is a natural Java Interface and is built on ODBC, and therefore JDBC retains some of the basic feature of ODBC	

Table 36: JDBC vs ODBC