CHAPTER I
I. INTRODUCTION

1.1 INTRODUCTION

This is a simple Java Player is made for java dealing with THE REAL-TIME MULTIMEDIA PRESENTATION & EFFECT PROCESSING. It handles time-based media which changes with respect to time. The user can also know THE METADATA INFORMATION while playing a time based media whether it is audio file (.mp3) or video file (.MPG or .MPEG) on the information pane. In this player the MENU BAR is reduced to much extend, resulting the graphical user interface only for the user. The user can switch to the views by tabs providing. It also saves the song list as .M3U file what is a standard and oldest format to be read by all players. There is also a facility provided to the user to create the colour theme for the player.

1.2 NECESSITY

The project is a desktop application created in java. In order to use it the user should meet the following requirements

- Java virtual machine (JVM) should be installed on the system.
- This application is designed for JAVA 6 version, so the user should at least have JAVA 6.0 runtime environment on the machine.
- The system should support at least 512KB cache memory for running this application

1.3 OBJECTIVES

The main objective of this application to provide convenience to choose the time based media in his accords. This will provide uniform information in the graphical view to the user instead to traversing through menus and menu items.

1.3.1 Practicality:

The system is stable and can be operated with average intelligence.

1.3.2 Efficiency:

There should be balance among various factors like accuracy, comprehensiveness on one hand and response timeliness on the other hand.

1.3.3 Cost:

It is desirable to aim for the system with the minimum cost subject to the condition that it must satisfy the entire requirement.

1.3.4 Flexibility:

This application is modifiable according to the changing needs of the user and time. It can be modified if further modification in JMF is done by the SUN MICROSYSTEM.

1.4 THEME

This project is developed in JAVA platform under java media framework. File handling is used for storing song playlist and reading from too. Since .M3U file for playlist has a

universal acceptance in all sort of media player so this application provides the facility to read and write files to be stored on the hard disk to be used later or in other players.

It is very inconvenient for the user to go and search through menus and menu items, so the basic approach in creating this application is to reduce the menu items as much possible.

1.4.1 Tab driven approach:

The basic idea behind this application is to make it as user friendly as possible. The menu items are removed and are changed to tab driven packages. The user interacts with different properties, functions and facilities by parsing through tabs reducing the time of searching for the user.

1.4.2 Easy retrieval of data:

Operations like creation, updating, removing all are done either using buttons or by recognition of mouse clicks. The user can easily view or retrieve data.

1.4.3 User friendly:

This application is completely user friendly. It is designed taking user with less computer knowledge into consideration. The user who may have very less computer knowledge can use it easily.

1.5 ORGANIZATION

NIIT is a leading Global Talent Development Corporation, building a skilled manpower pool for global industry requirements. The company which was set up in 1981, to help the nascent IT industry overcome its human resource challenges, has today grown to rank among the world's leading talent development organisations offering learning solutions to Individuals, Enterprises and Institutions across 40 countries. NIIT's training solution in IT, Business Process Outsourcing, Banking, Finance and Insurance, Executive Management Education and Communication and Professional Life Skills, touch five million learners every year.

NIIT's expertise in learning content development, training delivery and education process management make us the most preferred training partner, worldwide.
Education, and Communication and Professional Life Skills, touch five million learners every year. NIIT's expertise in learning content development, training delivery and education process management make us the most preferred training partner, worldwide.

Research-based Innovation, a key driver at NIIT, has enabled us to develop programmes and curricula that use cutting-edge instructional design methodologies and training delivery. NIIT's Individual Learning Solutions include industry-endorsed IT training programmes like GNIIT, Integrated programmes for Engineers (NIIT Edgeineers) and Infrastructure Management programmes (NIIT GlobalNet+).

NIIT Imperia, Centre for Advanced Learning, brings Executive Management Education Programmes from premier B-schools in India, to the doorsteps of working professionals.

NIIT Institute of Finance Banking & Insurance (IFBI), set up by NIIT with equity participation from ICICI Bank, offers programmes for individuals and corporates in the Banking, Financial Services and Insurance segments.

NIIT Uniqua, Centre for Process Excellence, addresses the increasing demand for skilled workers in the business and technology services industry by providing training programmes in relevant areas. This initiative is a part of the NIIT Institute of Process Excellence, a NIIT-Genpact joint venture.

NIIT's School Learning Solutions offer turnkey IT integration programmes for schools and has provided computer-based learning to nearly 7.8 million students in over 9,500 Government and private schools. NIIT eGuru, a comprehensive range of learning solutions for schools is powering NIIT's portfolio for the K-12 segment.

In order to address the vast population of underprivileged, school-aged children, NIIT launched the Hole-in-the-Wall education initiative, which has been recognised and acclaimed globally. Our achievements in the area of Minimally Invasive Education earned us the coveted **Digital Opportunity Award, conferred by the World Information Technology Services Alliance (WITSA) in 2008.**

NIIT's Corporate Learning Solutions business offers integrated learning solutions, including strategic consulting, learning design, content development, delivery, technology, assessment and learning management to Fortune 500 companies, Universities, Technology companies, Training corporations and Publishing houses. Element K, the spearhead of our corporate learning solutions, provides a tailored combination of catalogue learning products, technology, and services to customers and partners. The offerings include: vLab®: hands-on labs, instructor-led courseware, comprehensive e-reference libraries, technical journals, and KnowledgeHub™ , a hosted learning management platform. NIIT, together with Element K, is now the first and the best choice for comprehensive learning solutions, worldwide.

1.5.1 Achievements & Milestones

1981
- Incorporated on Dec 2, 1981.

1982
- Sets up Education Centres in Bombay, Chennai
- Introduced Multimedia technology in education

1983
- Education Centre set up in Bangalore
- Corporate Training programs introduced

1984
- IT Consultancy services started

1985
- Head Office integrated at New Delhi

1986
- Software Product Distribution started under 'Insoft' brand

1987	• Education Centre set up in Calcutta, Hyderabad • Conceived the Franchising model of Education
1988	• The birth of an NIIT-ian -- a branding for NIIT alumni
1989	• Education Centre opens in Pune
1990	• Created the Computer drome to provide "unlimited" computer time to students

CHAPTER II

LITERATURE SURVEY RELATED WITH TRAINING

2.1 Preliminary investigation:

The purpose of the preliminary investigation is to evaluate project requests. It is not a design study nor does it include the collection of details to describe the business system in all respects.

Analysts working on the preliminary investigations should accomplish the following objectives:

- Clarify and understand the project request.
- Determine the size of the project.
- Assess costs and benefits of alternative approaches.
- Determine the technical and operational feasibility of alternative approaches.
- Report the findings to management, with recommendations outlining the acceptance or rejection of the proposal.

2.2 System Analysis:

System analysis is the process of studying the business processors and procedures, generally referred to as business systems, to see how they can operate and whether improvement is needed.

This may involve examining data movement and storage, machines and technology used in the system, programs that control the machines, people providing inputs, doing the processing and receiving the outputs.

System analysis is conducted with the following objectives in mind:

- Identify the customers need
- Evaluate the system concept for feasibility
- Perform economic and technical analysis
- Allocate functions to hardware, software, people, database and other system elements
- Establish cost and schedule constraints
- Create a system definition that forms the foundation for all subsequent engineering

2.3 Feasibility study:

Prior to stating whether the system is feasible or not, we believe that we should emphasize on what is implied by the word **Feasible**. Feasibility is the measure of how beneficial or practical the system is which is to be developed for the organization. It is a preliminary survey for the systems investigation. It aims to provide information to facilitate a later in-depth investigation.

The report produced at the end of the feasibility study contains suggestions and reasoned arguments to help management decide whether to commit further resources to the proposed project.

Within the scheduled duration we were assigned to study both the positive and negative aspects of the current manual system, in which we have come up with a

number of drawbacks that prevent the progress of the clinic if it is continued to function manually.

If and when the objectives of the system are met and the new system is approved, then the more specific details in the proposal should be considered and approved.

The decision to implement any new project or program must be based on a thorough analysis of the current operation. In addition the impact of implementation of the proposed project/program on the future operation of a system must be evaluated. Such an analysis is critical in making a final decision on whether to progress and how that progression should occur. A feasibility study provides the process for this analysis.

The feasibility study is conducted to assist the decision-makers in making the decision that will be in the best interest of the proposed operation. The extensive research, conducted in a non-biased manner, will provide data upon which to base a decision.

After deciding on the area to study and gathering the appropriate documents, the developer take next step to identify who will be reading the feasibility study. Feasibility studies can provide developers with the comfort and knowledge that comes with thorough research and objectivity. In short, besides providing data and recommendations, a good feasibility study can give owners confidence in their project.

2.3.1 Types:

There are many feasibility measures using which we can know if the system is feasible or not. Some of methods are:

2.3.1.1 Technical Feasibility
2.3.1.2 Operational Feasibility
2.3.1.3 Economical Feasibility

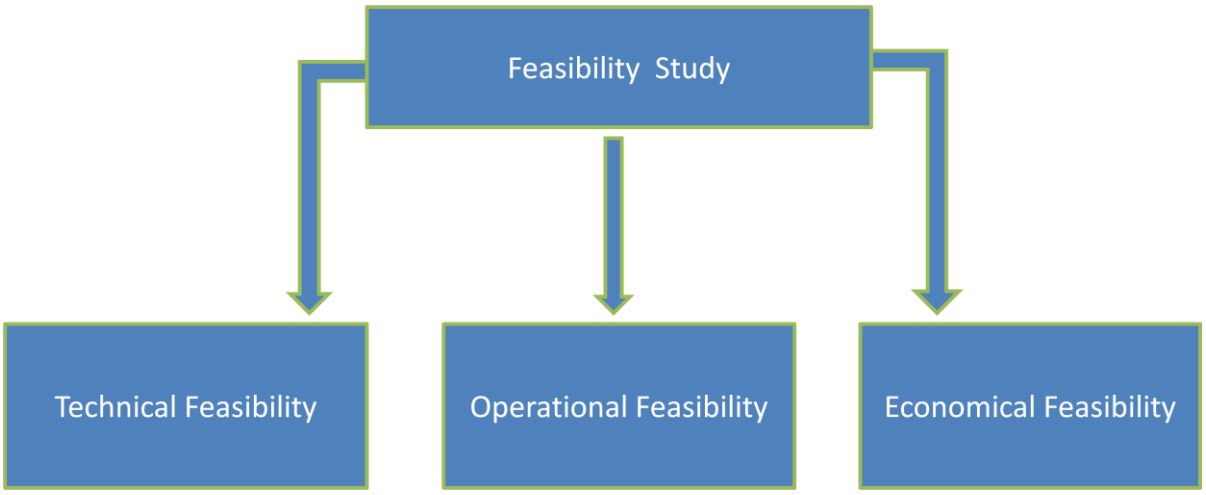


Fig 2.1 Types of feasibility study

2.3.1.1 Technical Feasibility:

Based on the outline design of system requirements in terms of inputs, outputs, files, procedures and staff, the technical issues raised during technical feasibility include:

- Does the necessary technology exist to do what is proposed?
- Does the proposed equipment have the technical capacity to hold the data required to use in the new system?
- Adequate responses provided by the proposed system?
- Is the system flexible enough to facilitate expansion?
- Is there any technical guarantee of accuracy, reliability, ease of access and data security?

Technical analysis begins with an assessment of the technical viability of the proposed system.

- What technologies are required to accomplished system function and performance?
- What new materials, methods, algorithms or processes are required and what is their development risk?
- How will these obtained from technical analysis form the basis for another go/no-go decision on the test system? If the technical risk is severe, if models indicate that the desired function cannot be achieved, if the pieces just won't fit together smoothly-it's back to the drawing board.

2.3.1.2 Operational Feasibility:

A proposed system is beneficial only if it can be turned into an information system that will meet the operational requirements of an organization. A system often fails if it does not fit within existing operations and if users resist the change.
Important issues a systems developer must look into are:

- Will the new system be used if implemented in an organization?
- Are there major barriers to implementation or is proposed system accepted without destructive resistance?

2.3.1.3 Economical Feasibility:

In making recommendations a study of the economics of the proposed system should be made. The proposed system must be justifiable in terms of cost and benefit, to ensure that the investment in a new/changed system provide a reasonable return.
Cost-benefit analysis of information is complicated by the fact that many of the systems cost elements are poorly defined and that benefit can often be highly qualitative and subjective in nature.
In our proposed system various costs are evaluated. Even though finding out the costs of the proposed project is difficult we and assume and estimate the costs and benefits as follows.
According to the computerized system we propose, the costs can be broken down to two categories.

- Costs associated with the development of the system.
- Costs associated with operating the system.

Among the most important information contained in feasibility study is Cost Benefit Analysis and assessment of the economic justification for a computer based system

project. Cost Benefit Analysis delineates costs for the project development and weighs them against tangible and intangible benefits of a system. Cost Benefits Analysis is complicated by the criteria that vary with the characteristics of the system to be developed, the relative size of the project and the expected return on investment desired as part of company's strategic plan. In addition, many benefits derived from a computer-based system are intangible (e.g. better design quality through iterative optimization, increased customer satisfaction through programmable control etc.)As this is an in-house project for the company, to be used for its own convenience and also it is not that big a project. So neither it requires a huge amount of money nor any costly tools or infrastructure need to be set up for it.

CHAPTER III

TRAINING WORK

3.1 Idea Statement

The idea came in to existence after reading the JMF's inbuilt codec and effect compatibility. Due to limited support in the standard API a java based mp3 player has to employ some external libraries. JMF is a mature project and it is employed in many existing java applications. Thus it seemed a logical choice to use it.

3.2 Why MP3?

The MP3 format is a very good target for this research because it is currently one of the most popular music encodings. Potential users of karaoke software are most likely to pick the mp3 format above any other audio encoding on the market. Because the end goal of this project is to create a usable piece of software, catering to the tastes and needs of the end users seems to be a good idea.

Furthermore, mp3 is an open standard which means that it is well documented and accessible. Thus the uncovering the inner workings of this format does not pose any legal threats to the researchers. On the other hand, choosing a proprietary, closed format such as Windows Media Audio (WMA) could put the researchers in legal jeopardy.

Doubtlessly any stegnographic research will rely heavily on exploiting certain properties of the data format chosen as the information carrier. This project is no different. However the actual research process, data examination and implementation steps could be replicated for other media to create analogous solutions.

3.2.1 MP3 Encoding:

MP3 is a lossy data format which aims to preserve the sound quality while minimizing storage space. The encoding process takes into account the properties of human auditory system. For example, humans cannot hear frequencies below 20Hz and above 20kHz. Furthermore human ear is often unable to distinguish between two or more notes with specific frequencies when they are played together. Thus mp3 file can safely discard any sounds with frequencies out of the audible scale, and needs to store only a single copy out of a group of similar sounding notes. This is of course not a trivial process. Mp3 encoders employ a complex psychoacoustic modeling to perceptually optimize the data.

The encoding process is very complex and involves both perceptual optimization, as well as more conventional data compression methods. Fig 3.1 shows a conceptual model of an mp3 algorithm:

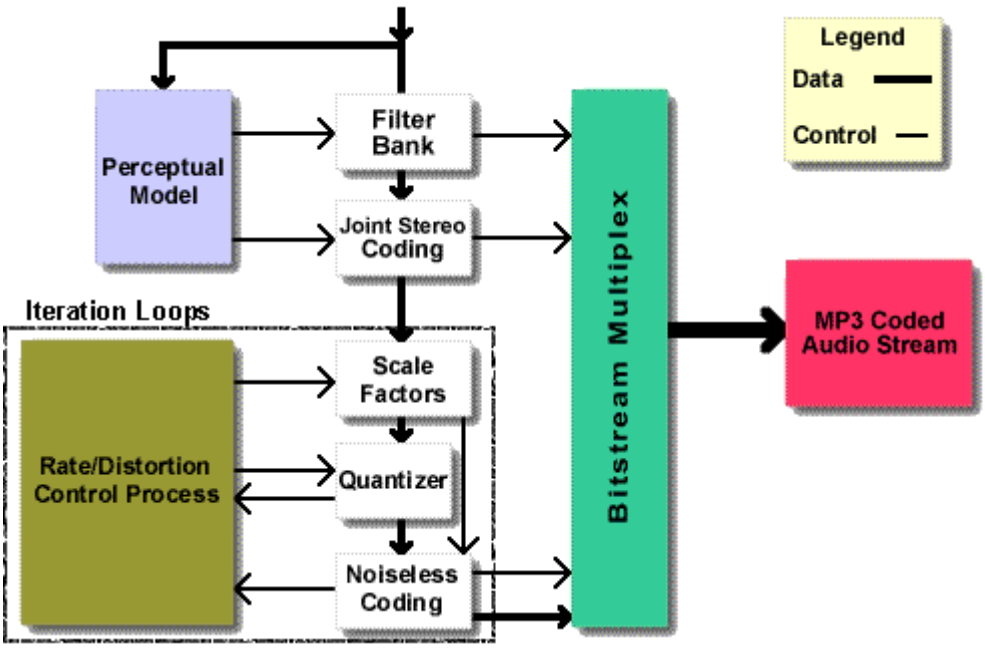


Fig 3.1 Conceptual model of MP3 algorithm

Explaining the actual encoding procedure is out of the scope of this paper. However there are several tasks performed by that the encoder that may be of some interest to a stegonography researcher.

The mp3 encoder breaks the audio data into small fragments called frames. Each frame represents a fraction of a second. The size of the frame depends on the audio resolution or bit rate. The most convenient (algorithmically) to do this is to assume a constant bit rate throughout the recording thus forcing same size onto all frames. However music is not structured this way. Very often very dynamic sequences including vocals and many instruments playing at the same time are interweaved with very simple melodic tracks. Therefore using a constant bit rate (CBR) is not always economical. MP3 specification allows the data to be stored in a variable bit rate format (VBR) which means that the audio frames are not the same size.

Each frame is perceptually analyzed using the psychoacoustic model. The frequencies that are not audible are discarded, or allocated minimal number of bits.

The exact inner workings of this procedure are complex and beyond the scope of this paper.

Once perceptual optimization is done, the data is compressed using Huffman coding. This is a lossless algorithm so the audio information is preserved, while decreasing storage space. This is an important fact for a stegonographer.

Because of the nature of the compression algorithm, Huffman coded data cannot be easily modified. Huffman coded data is stored using variable length bit strings that are matched against a lookup table. The most frequently used characters are encoded with the shortest possible strings, while the rare ones are coded with longer strings. Thus it is possible that certain values have two or three bit codes. Inverting a single bit therefore can completely change a value of the coded data.

Furthermore the data cannot be easily divided into bytes, words and etc. So the least significant bit of a given byte may actually be the most significant bit of a Huffman coded character. Therefore least significant bit substitution cannot be easily done on Huffman coded data.

Compressed audio data is then reassembled. Each frame is pre-pended with a header which stores information about the bit rate, sample rate and other meta-data.

3.2.2 MP3 File Structure:

Fig 3.2 below shows a conceptual model of an mp3 file:

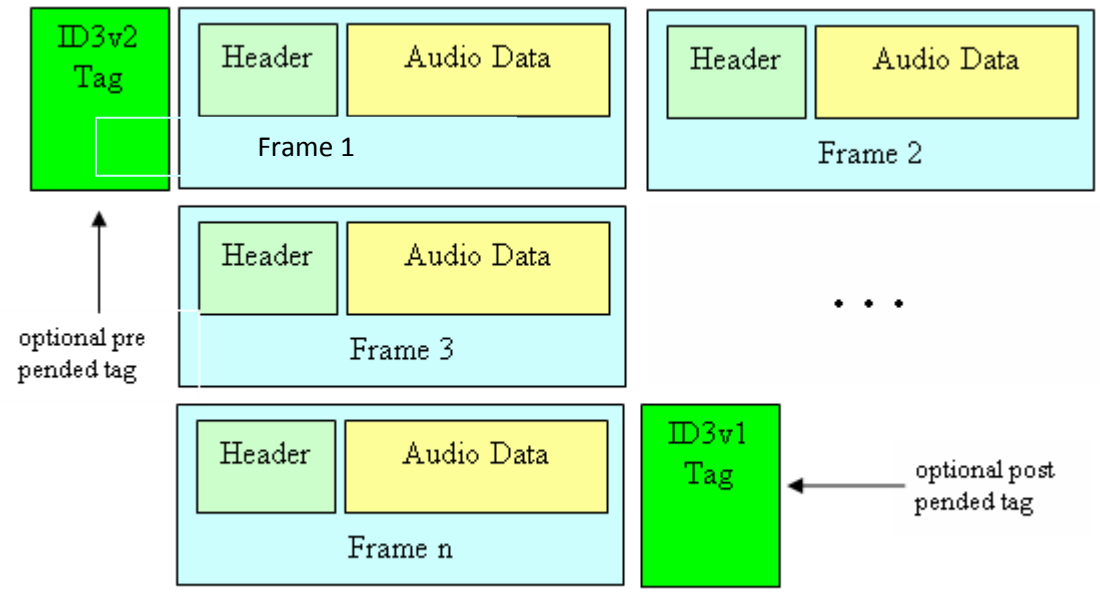


Fig 3.2 MP3 file structure

Due to their extendibility the ID3v2 tags would be an interesting target for embedding information, however they are not guaranteed to be present in every mp3 file. Thus the best approach is to embed the data into the data frames. Before discussing stegonographic methodology however, it would be best to take a closer look at the data frame.

3.3 Project Plan:

3.3.1 Team Structure:

The term team is used here to mean a group of people with similar skills and sharing a common activity. A project team is staffed by system analysts, programmers, prime users, hardware/software suppliers and even sub contractor. The staff may be retained for the duration of the project.

The project team is expected to tap the skills of its members can address the issues that face the project and suggest alternative solutions to keep the project moving the completion.

3.4 System Development Life cycle:

Systems Development Life Cycle (SDLC) is any logical process used by a systems analyst to develop an information system, including requirements, validation, training, and user (stakeholder) ownership. The systems development lifecycle (SDLC) is a type of methodology used to describe the process for building information systems, intended to develop information systems in a very deliberate, structured and methodical way, reiterating each stage of the life cycle.

3.4.1 Systems Development phases:

Systems Development Life Cycle (SDLC) adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation.

There are several Systems Development Life Cycle Models in existence. The oldest model, that was originally regarded as "the Systems Development Life Cycle" is the waterfall model: a sequence of stages in which the output of each stage becomes the input for the next. These stages generally follow the same basic steps but many different waterfall methodologies give the steps different names and the number of steps seems to vary between 4 and 7. There is no definitively correct Systems Development Life Cycle model, but the steps can be characterized and divided in several steps.

According to Elliott (2004), "the traditional life cycle approaches to systems development have been increasingly replaced with alternative approaches and

frameworks, which attempted to overcome some of the inherent deficiencies of the traditional SDLC"

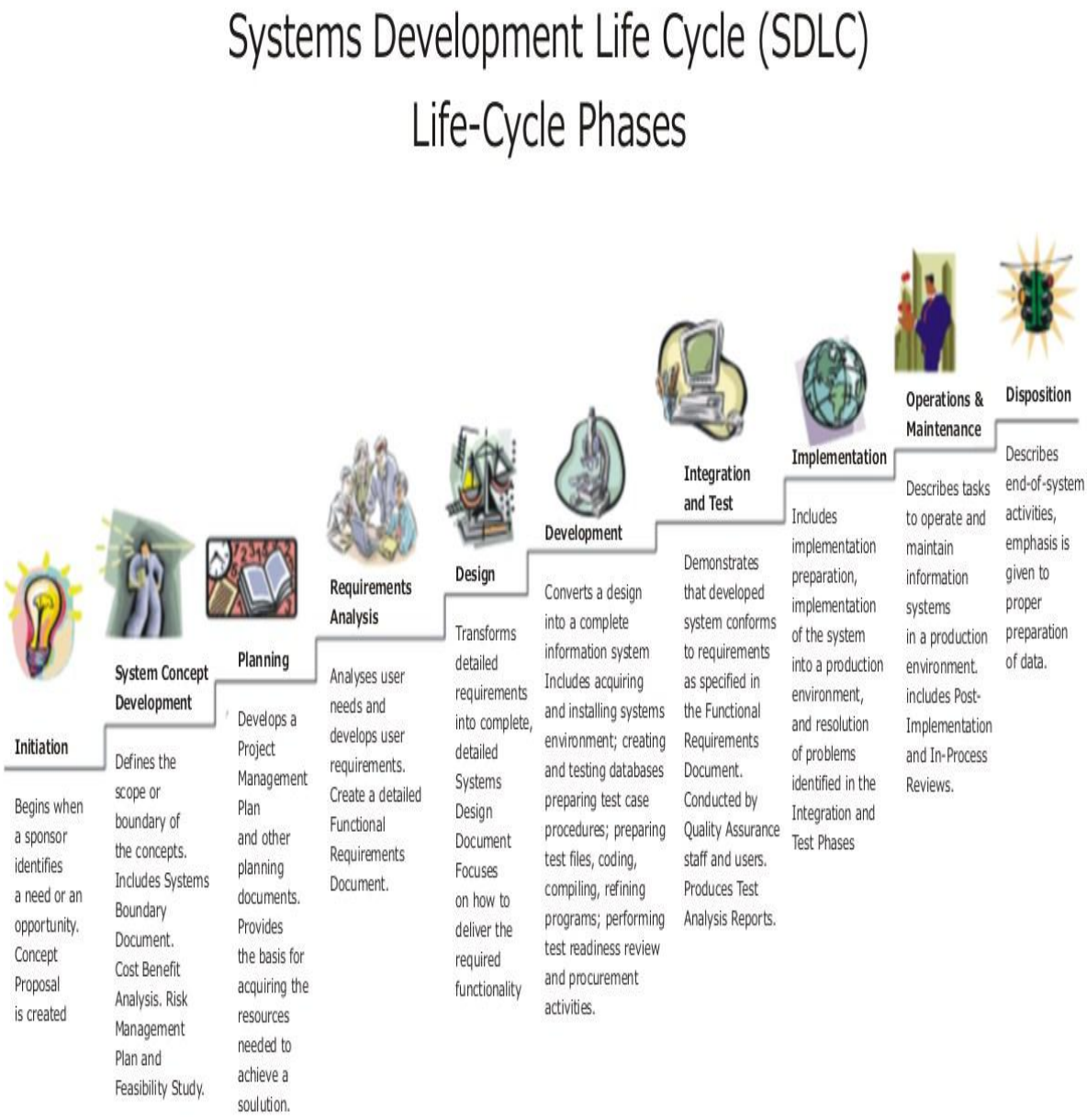


Fig 3.3 Different phases of SDLC

3.4.1.1 Initiation/Planning

To generate a high-level view of the intended project and determine the goals of the project. The feasibility study is sometimes used to present the project to upper management in an attempt to gain funding. Projects are typically evaluated in three areas of feasibility: economical, operational or organizational, and technical. Furthermore, it is also used as a reference to keep the project on track and to evaluate the progress of the MIS team. The MIS is also a complement of those phases. This phase is also called the analysis phase.

3.4.1.2 Investigation phase:

The investigation phase is also known as the fact-finding stage or the analysis of the current system. This is a detailed study conducted with the purpose of wanting to fully understand the existing system and to identify the basic information requirements. Various techniques may be used in fact-finding and all fact obtained must be recorded.

A thorough investigation was done in every effected aspect when determining whether the purposed system is feasible enough to be implemented.

3.4.1.2.1 Investigation:

As it was essential for us to find out more about the present system, we used the following methods to gather the information: -

Observation: - Necessary to see the way the system works first hand.

Document sampling: - These are all the documents that are used in the system. They are necessary to check all the data that enters and leaves the system.

Questionnaires: - These were conducted to get views of the other employees who are currently employed in the system.

3.4.1.2.2 Analysis of Investigation:

The goal of systems analysis is to determine where the problem is in an attempt to fix the system. This step involves breaking down the system in different pieces and drawing diagrams to analyze the situation, analyzing project goals, breaking need to be created and attempting to engage users so that definite requirements can be defined. Requirement Gathering sometimes require individual/team from client as well as service provider side to get a detailed and accurate requirements. Analysis gathers the requirements for the system. This stage includes a detailed study of the business needs of the organization. Options for changing the business process may be considered. Design focuses on high level design like, what programs are needed and how are they going to interact, low-level design (how the individual programs are going to work), interface design (what are the interfaces going to look like) and data

design (what data will be required). During these phases, the software's overall structure is defined. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

3.4.1.2.3 Constraints and Limitations:

The constraints and limitation within a system are the drawbacks that occur during the implementation of the system. These limitations and constraints can crop up in almost every system; the most important fact is to find a way to overcome these problems.

Software design is the first of three technical activities – design, code generation, and test that are required to build and verify the software. Each activity transforms information in manner that ultimately results in validated computer software.

The design task produces a data design, an architectural design, an interface design and component design.

The design of an information system produces the details that clearly describe how a system will meet the requirements identified during system analysis. The system design process is not a step by step adherence of clear procedures and guidelines. When I started working on system design, I face different types of problems; many of these are due to constraints imposed by the user or limitations of hardware and software available. Sometimes it was quite difficult to enumerate that complexity of the problems and solutions thereof since the variety of likely problems is so great and no solutions are exactly similar however the following consideration I kept in mind during design phased.

Analysis is a detailed study of the various operations performed by the system and the relationships within and outside the system .In this stage the organization management studies the feasibility report and suggests modifications if any . Then the system analyst develops final specifications of the information system .After knowing the constraints on available resources and modified requirements specified by the organization , the analyst do the detailed analysis and define boundaries of the system. During analysis , data are collected on the available files , decision points and transactions handled by the present system . The tools used are analysis are DFDs, questionnaires etc.

3.4.1.3 System design

System design is the process of developing specifications for a candidate system that meet the criteria established in the system analysis. Major step in system design is the preparation of the input forms and the output reports in a form applicable to the user.
The main objective of the system design is to make the system user friendly. System

design involves various stages as:

- Data Entry
- Data Correction
- Data Deletion
- Processing
- Sorting and Indexing
- Report Generation

System design is the creative act of invention, developing new inputs, a database, offline files, procedures and output for processing business to meet an organization objective. System design builds information gathered during the system analysis.

3.4.1.3.1 Characteristics of well defined System:

In design an efficient and effective system is of great importance to consider the human factor and equipment that these will require to use. System analyst must evaluate the capabilities and limitations of the personal and corresponding factors of the equipment itself.

The characteristics associated with effective system operations are:

- Accessibility
- Decision Making Ability
- Economy
- Flexibility
- Reliability
- Simplicity

Success is a new system pivots on its acceptance or non-acceptance by the organization.

3.4.1.3.2 Two layers of Projects

- Stored Procedures
- Designing

3.4.1.3.2.1 Stored Procedure

I. Create player
II. Stop Player
III. Pause Player
IV. Destroy Player
V. Set Volume
VI. Play Linked List
VII. Video Panel
VIII. Full Screen
IX. Audio Property
X. File Manager
XI. Save Playlist Window
XII. Settings

I. Create Player

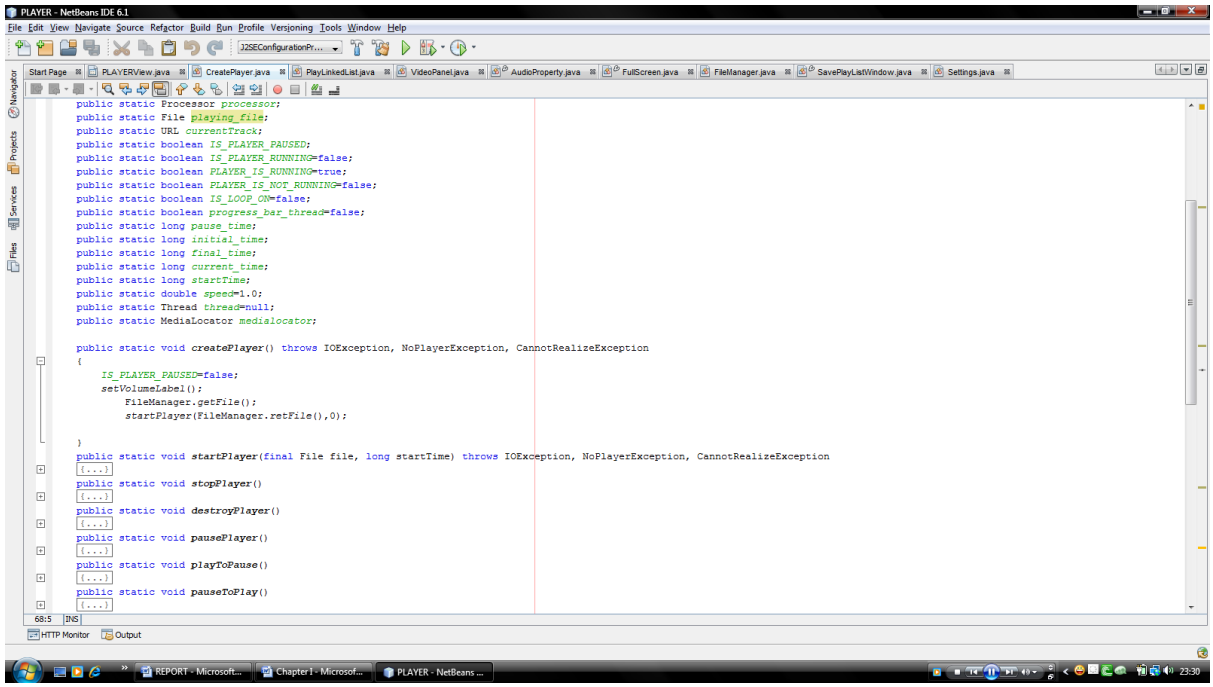


Fig. 3.4 Create Player Class

II. Stop Player

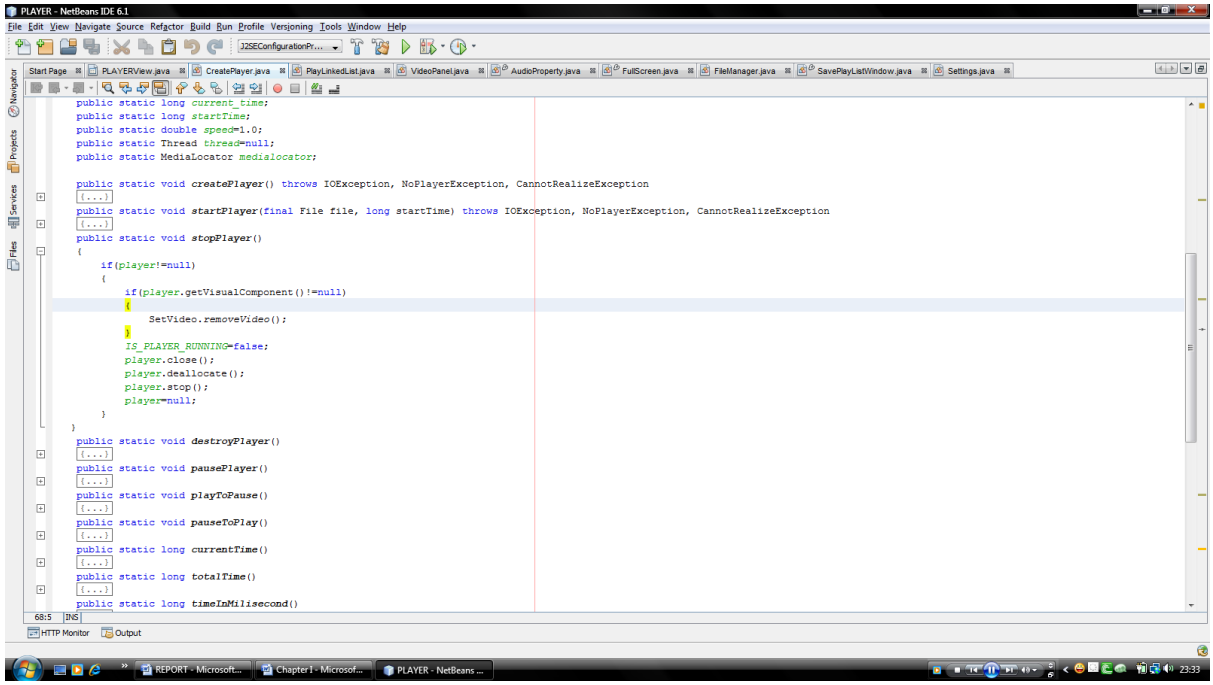


Fig.3.5 Stop Player Procedure

III. Pause Player

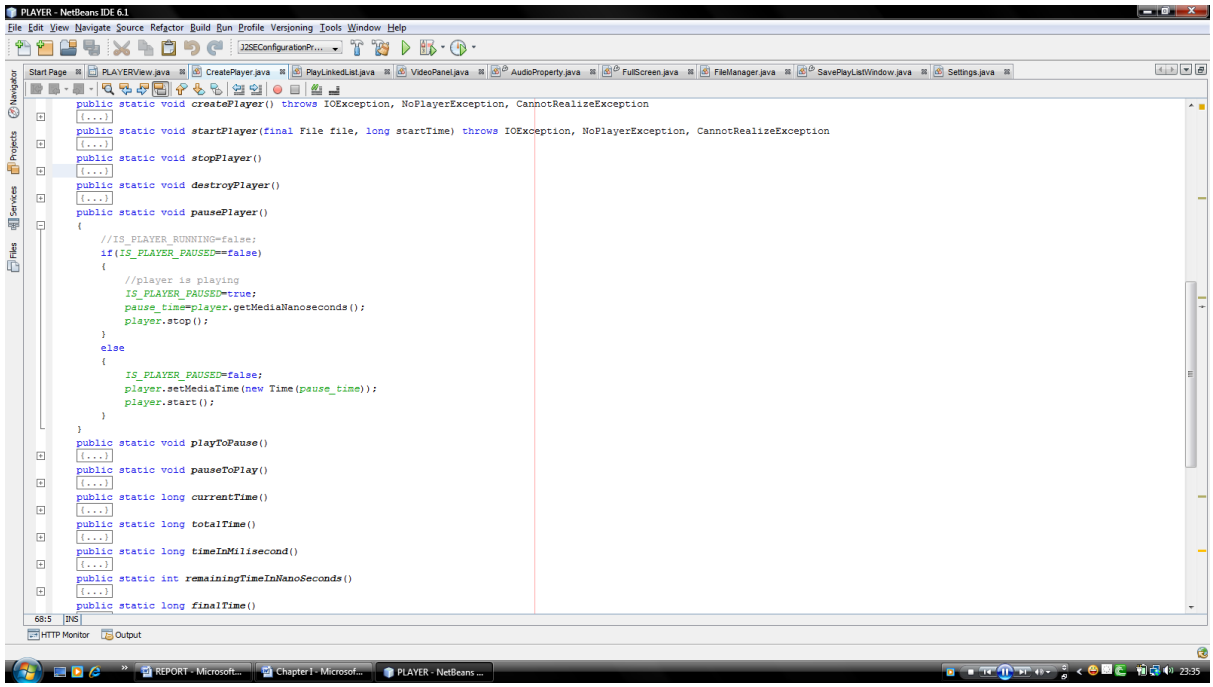


Fig. 3.6 Pause Player Procedure

IV. Destroy Player

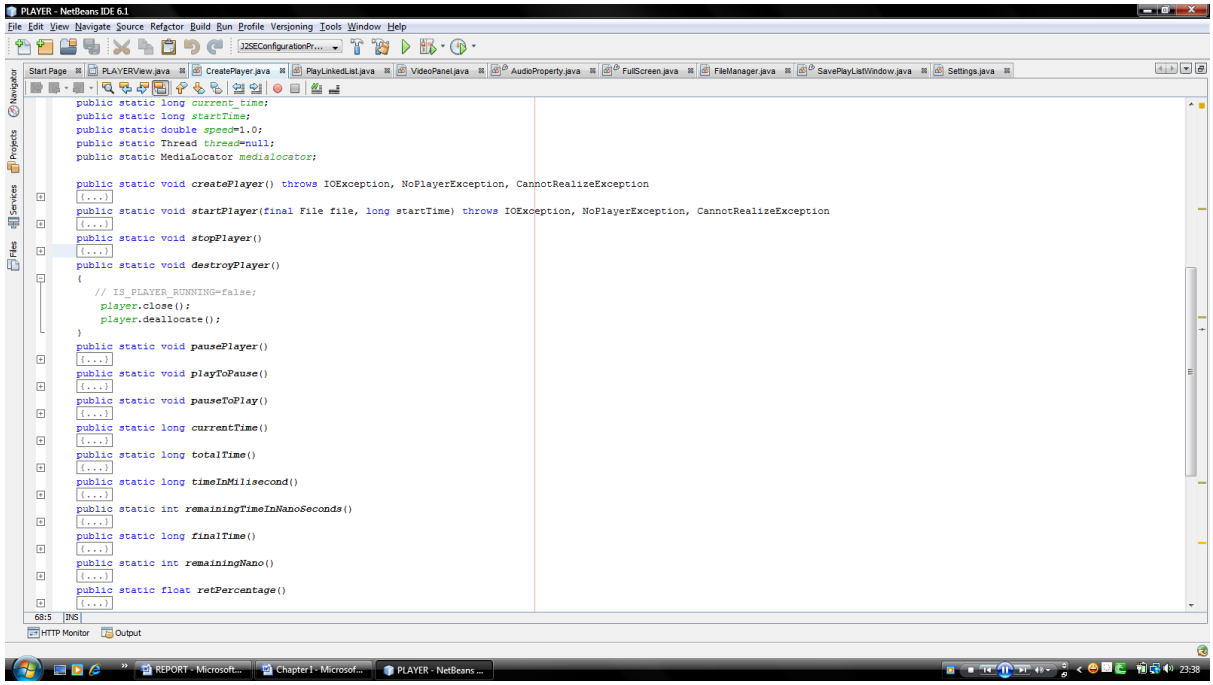


Fig. 3.7 Destroy Player Procedure

V. Set Volume

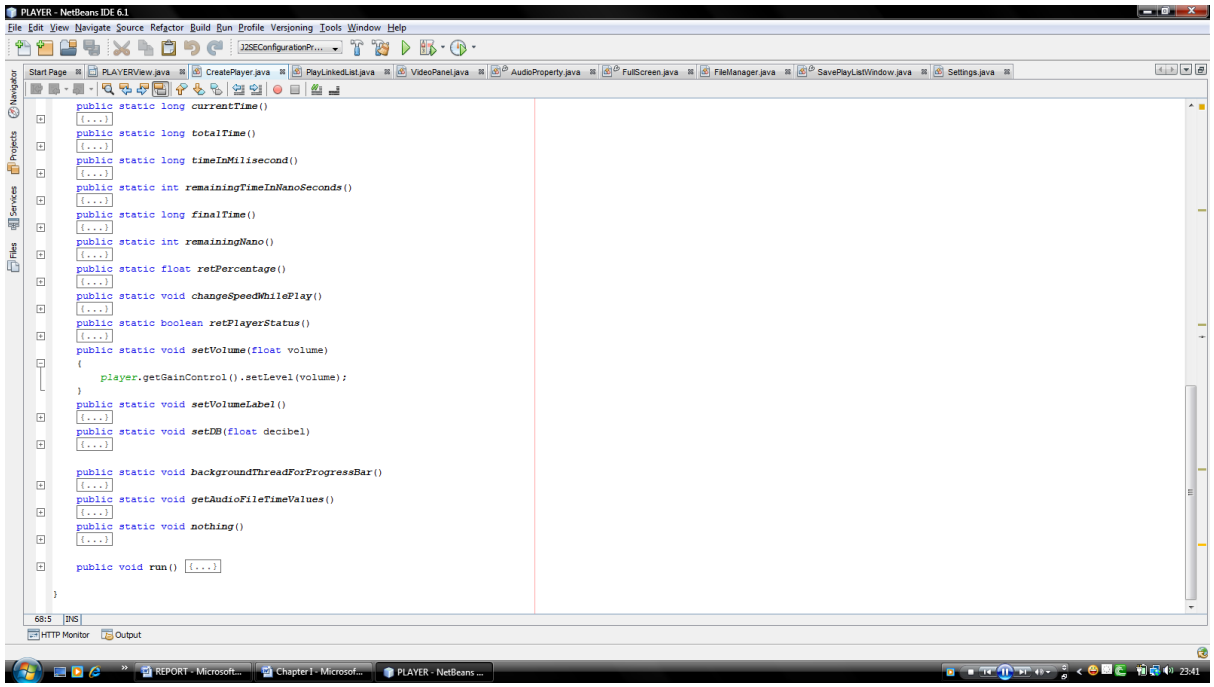


Fig 3.8 Set Volume Procedure

VI. Play Linked List

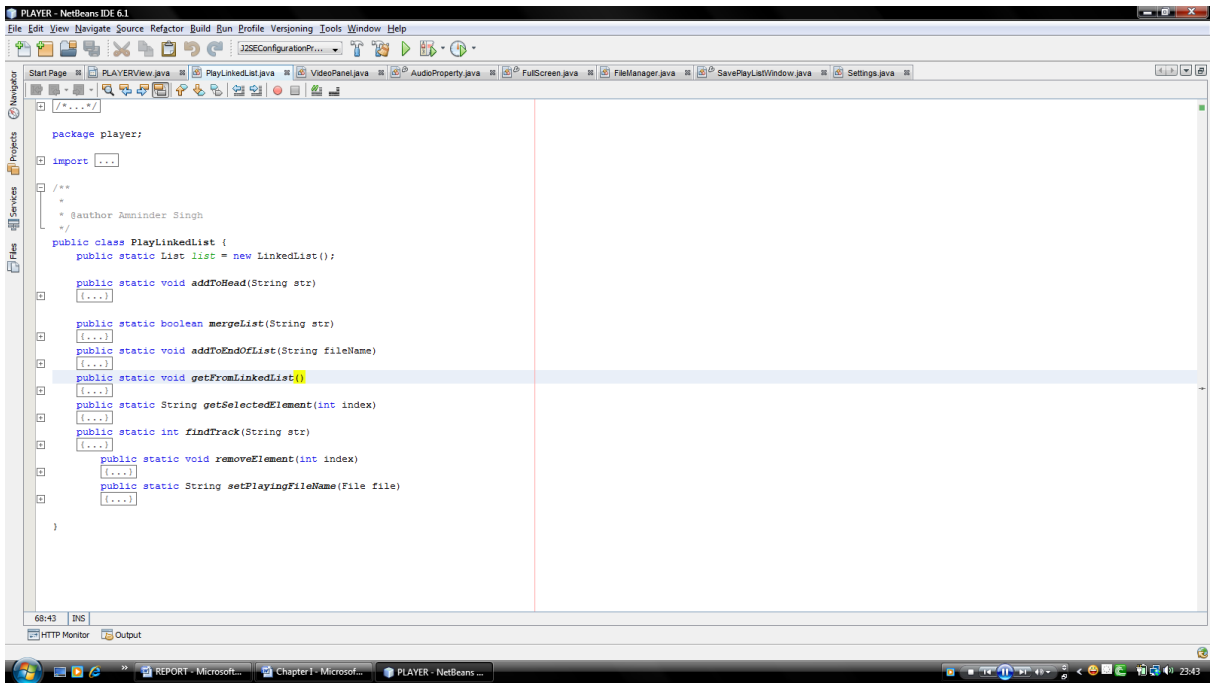


Fig. 3.9 Play Linked List Procedure

VII. Video Panel

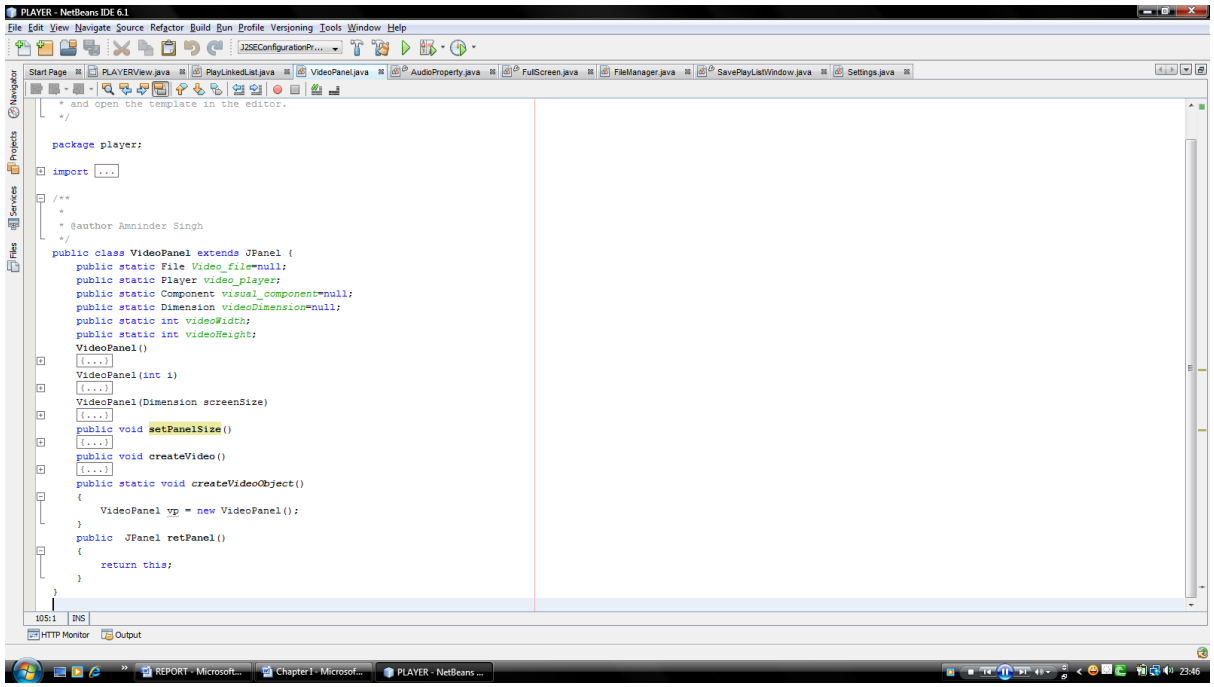


Fig. 3.10 Video Panel Class & Procedure

VIII. Full Screen

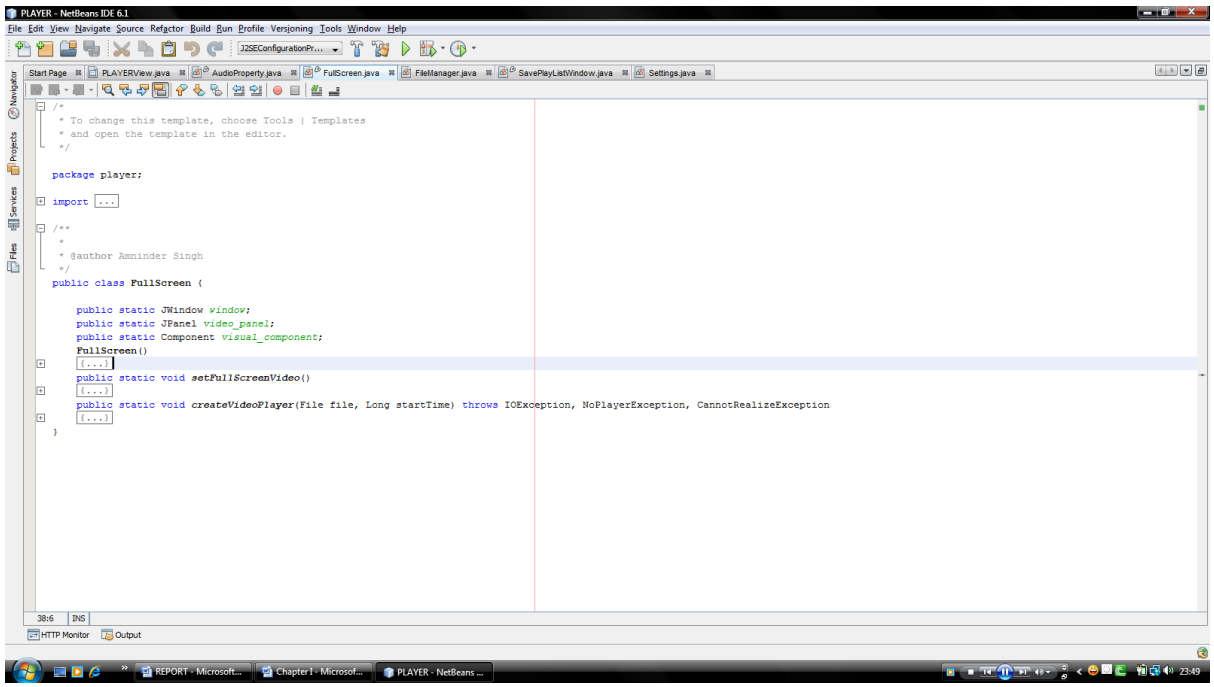


Fig. 3.11 Full Screen Class

IX. Audio Property

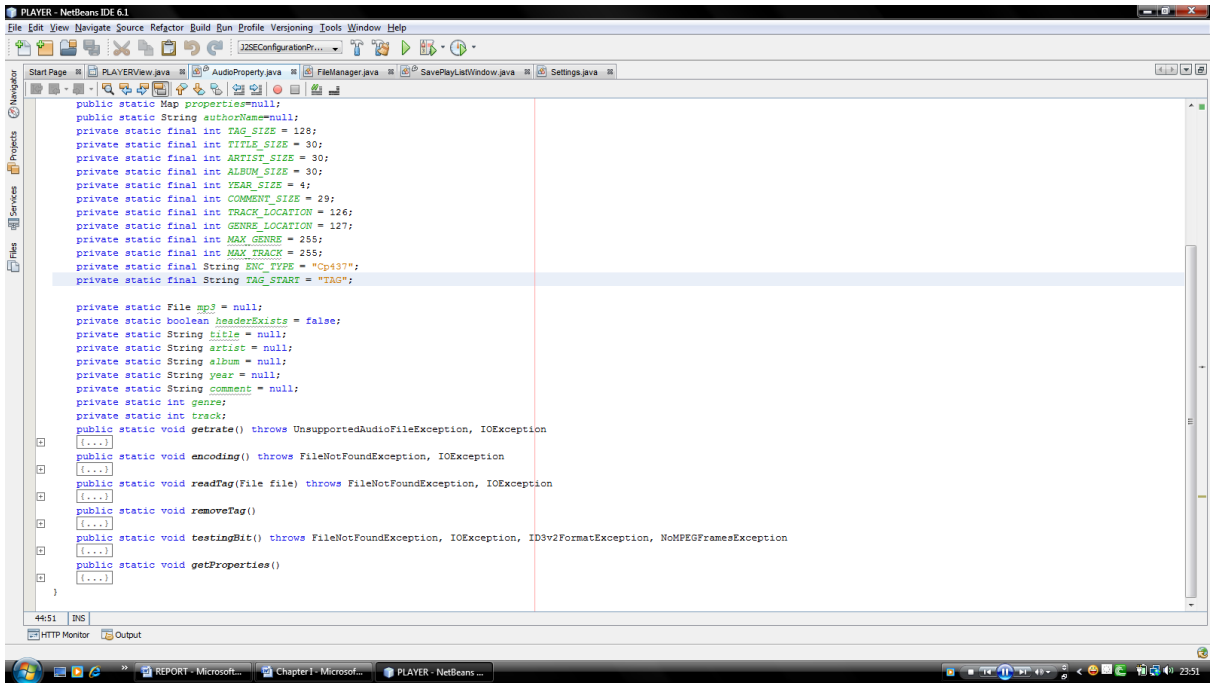


Fig. 3.12 Audio Property class of Media File

X. File Manager

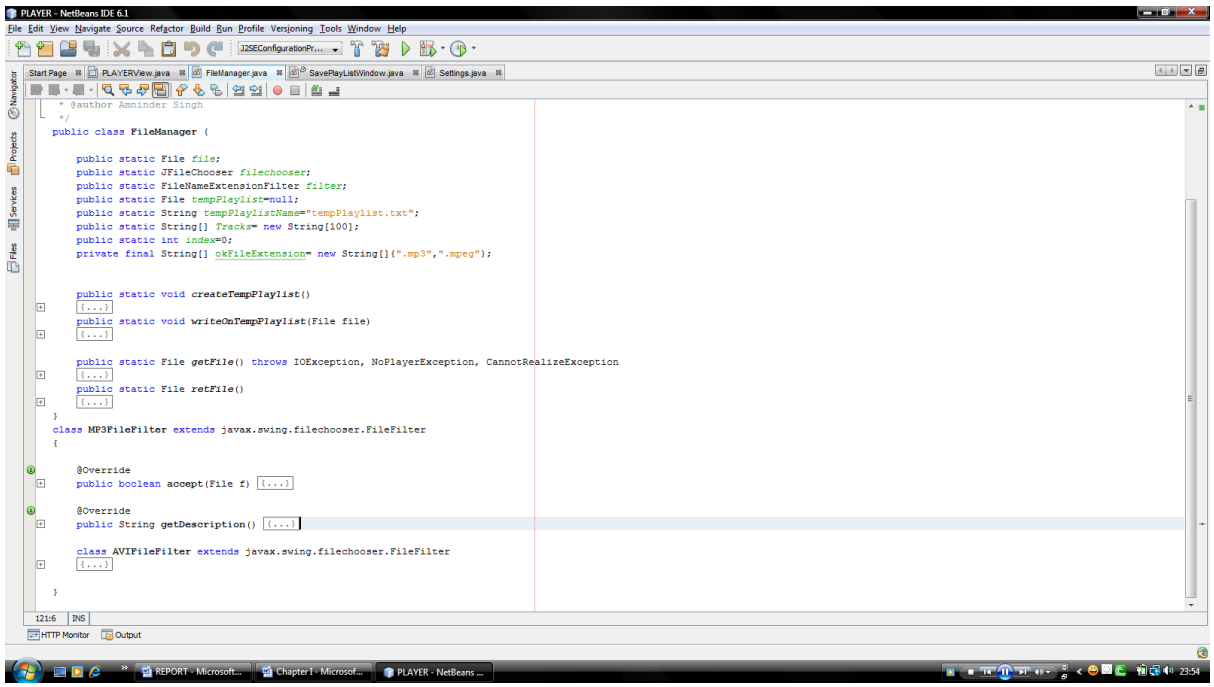


Fig. 3.13 File Manager Class

XI. Save Play List Window

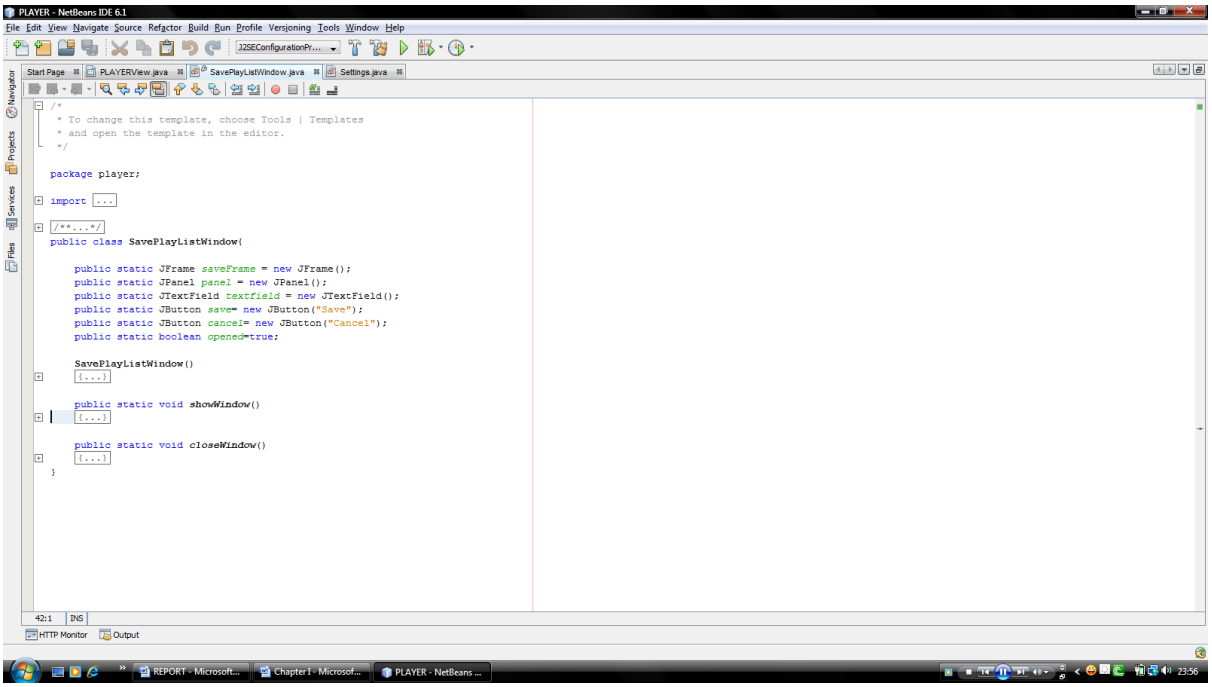


Fig. 3.14 Save Play List Window Class

XII. Settings

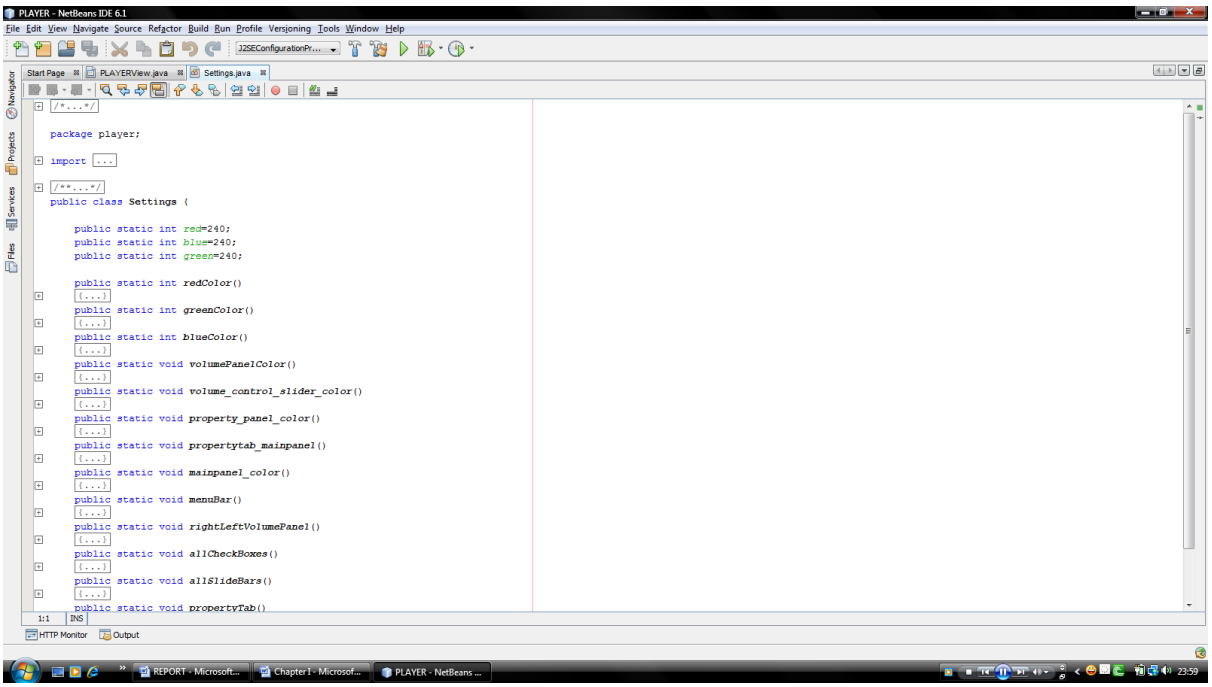


Fig. 3.15 Settings Class (color themes)

3.4.1.3.2.2 Designing Layer

Designing layer consists of the main design panel of the application which have further panels:

Load Songs Tab:

- Load songs panel consists of a File Chooser window which shows the files to be selected.
- This window has File Filter applied on.
- File Filter only shows either .MP3, .MPG, .MPEG, FOLDERS files to be selected.
- MOUSE.CLICK event listener is applied on the JFileChooser to selected either by double clicking on the file or the user may can selected the file by clicking on **Load Playlist** button on it.
- On selection, the file is loaded on the list pane and the path is stored in linked list.

Status Tab:

- This pane included two components, Video Panel and JProgressBar component.
- Video is loaded on video panel only when the media file includes Visual component. Otherwise this panel remains blank
- The Progress Bar tells the percentage of the media content elapsed and it also shows the time elapsed.

Color Theme Tab:

- This tab panel has three sliders for three colors RED, BLUE and GREEN.
- RED slider controls the red color values of all the panels and components.
- BLUE slider controls the blue color values of all the panels and components.
- GREEN slider controls the green color values of all the panels and components
- The respective values are displayed on the label attached on the labels provided.

Property Tab:

- Property tab embeds the Meta Data values of the stream playing.
- The elapsing time is displayed by the label provided at the top of the panel
- Album information like Artist , Album, Track, Genre and year information of the stream is provided at the labels embedded at the west direction labels.
- Audio information like Version, Layer, Channel, Copyrighted value, CRCED and Emphasis value of the stream is provided at the labels embedded at the east direction labels
- Buttons **Full Screen** is provided to switch to the full screen view if available.

- **Write** button is provided to write the song lists in the playlist file.
- **Read** button is provided to read from the playlist file.

Control Panel:

- Control panel controls the functioning of the playing stream.
- It has four buttons for playing, stop, next track and previous track.
- On west direction, album information is embedded which is displayed when the stream is playing.
- Playback speed of stream playing is controlled by the checkbox **2X ON**. On checkbox selected the speed changes to two times the normal speed and when unselected the speed changes to normal.
- Loop checkbox controls the loop playback of the playlist. On checkbox selected, the loop turns on and when checkbox unselected, loop turns off.
- The play list is saved on clicking button **Save PlayList**
- The visibility of the playlist panel is controlled by button **PlayList ON** button.
- Volume of the stream is controlled by the vertical slider provided.

PlayList Panel:

- Song list is stored on the Play List panel.
- List Pane is added on the play List panel.
- Mouse click event is listened on the list items, right click removed the song from the list

3.5 Hardware & Software Requirements

3.5.1 JAVA System Requirements:

To ensure the adequate performance, java should meet following requirements.

3.5.1.1 Operating System Requirements:

Java supports the following framework:

Scenario	Operating System
Client	Windows XP Windows Millennium Edition Windows Media Centre

3.5.1.2 Hardware Requirement:

Scenario	Processor	RAM	Required	Recommended
Client	Pentium	500MHz	128MB	64MB

3.6 Data Flow Diagram

In our DFD, we give names to data flows, processes, and data stores. Although the names are descriptive of the data, they do not give details. So the following the DFD, our interest is to build some structured place to keep details of the contents of data flow, processes, and data store. A data dictionary is a structured repository of data about data. It is a set of rigorous definition of all DFD data element and data structure

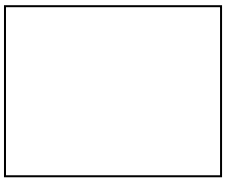
3.6.1 DFD Symbols:

In the DFD, there are four symbols,

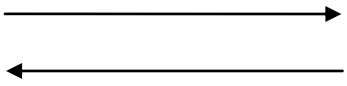
- **A Square** defines a source (originator) or destination of system data.
- **An Arrow** identifies data flow- data in motion .It is pipeline through which information flows.
- **A circle** or a **bubble** (or a oval bubble) represents a process that transforms incoming data flow(s) into outgoing data flow(s)
- **An open rectangle** is a data store-data at rest, or temporary repository of data.
- The DFD was first developed by "Larry Constatine" as a way of expressing system requirements in a graphical form. A DFD, also referred to as a bubble chart has a purpose of clarifying system requirements and identifying major transformations that will become the program in this system design.

3.6.1.1 DFD Symbol description:

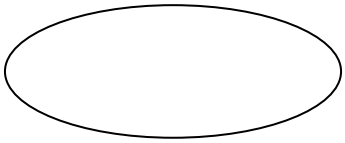
- A square defines a source of destination or system data.



- An arrow line identifies the data flow or data in motion. It is a pipeline through which information flows.

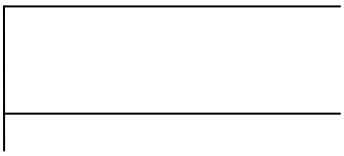


- A circle or bubble represents a process transform incoming data flow in to outgoing data flow.



- A horizontal line represents data stored or data at rest or a temporary rest repository of data.

- An open rectangle refers to the database storage



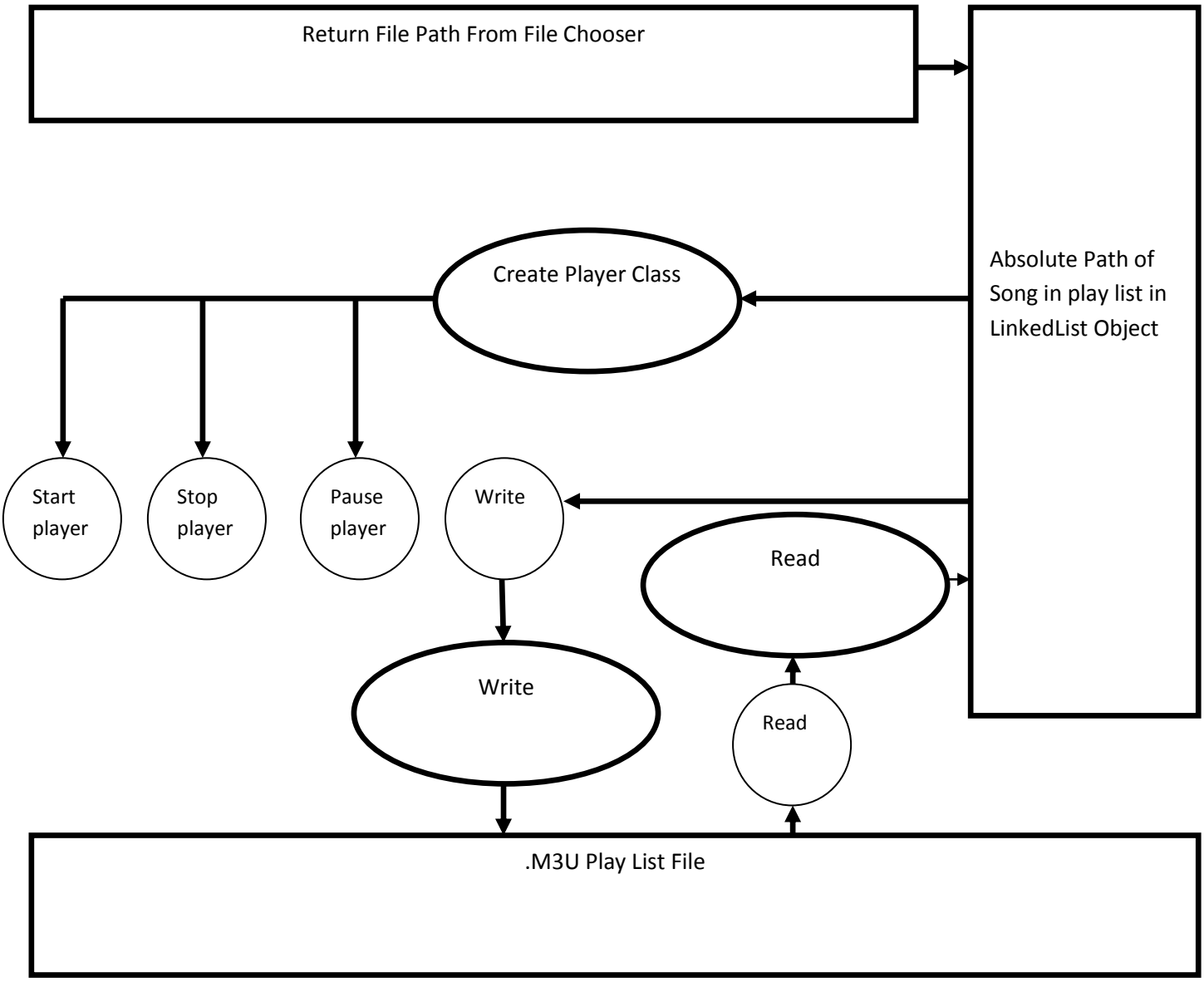


Fig. 3.16 DFD of basic class structure of Player

3.7 Overview of JAVA

3.7.1 Introduction to Java

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words.[10] Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation

The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of their Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

THE FIVE PRIMARY GOALS OF JAVA LANGUAGE ARE:-

1. It is "simple, object oriented, and familiar".
2. It is "robust and secure".
3. It is "architecture neutral and portable".
4. It executes with "high performance".
5. It is "interpreted, threaded, and dynamic"

3.7.2 JAVA PLATFORM:-

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode instructions are analogous to machine code, but are intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

Standardized libraries provide a generic way to access host-specific features such as graphics, threading and networking.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would, and Java suffered a reputation for poor performance. This gap has been narrowed by a number of optimization techniques introduced in the more recent JVM implementations.

3.7.3 Performance:

Programs written in Java have a reputation for being slower and requiring more memory than those written in some other languages. However, Java programs' execution speed improved significantly with the introduction of Just-in-time compilation in 1997/1998 for Java 1.1, the addition of language features supporting better code analysis,[clarification needed] and optimizations in the Java Virtual Machine itself, such as HotSpot becoming the default for Sun's JVM in 2000.

JUST-IN-TIME COMPILER (JIT COMPILER):-

In computing, just-in-time compilation (JIT), also known as dynamic translation, is a technique for improving the runtime performance of a computer program.

JIT builds upon two earlier ideas in run-time environments:

bytecode compilation and dynamic compilation. It converts code at runtime prior to executing it natively, for example bytecode into native machine code. The performance improvement over interpreters originates from caching the results of translating blocks of code, and not simply reevaluating each line or operand each time it is met. It also has advantages over statically compiling the code at development time, as it can recompile the code if this is found to be advantageous, and may be able to enforce security guarantees. Thus JIT can combine some of the advantages of interpretation and static (ahead-of-time) compilation.

** Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution*

3.7.4 CONDITION STATEMENTS

- **THE If Else STATEMENT:-**
  The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed

only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes(){
    if (isMoving){  // the "if" clause: bicycle must be moving
        currentSpeed--; // the "then" clause: decrease current speed
    }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.
In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes(){
    if (isMoving) currentSpeed--; // same as above, but without braces
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

- **THE If-Then-Else STATEMENT:**

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes(){
    if (isMoving) {
        currentSpeed--;
    } else {
        System.out.println("The bicycle has already stopped!");
    }
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
   public static void main(String[] args) {

      int testscore = 76;
```

```
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is:

```
    Grade = C
```

3.7.5 THE Switch STATEMENT

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types (discussed in Classes and Inheritance) and a few special classes that "wrap" certain primitive types: Character, Byte, Short, and Integer (discussed in Simple Data Objects).

The following program, SwitchDemo, declares an int named month whose value represents a month out of the year. The program displays the name of the month, based on the value of month, using the switch statement.

```
class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
```

```
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month.");break;
        }
    }
}
```

In this case, "August" is printed to standard output.

The body of a switch statement is known as a switch block. Any statement immediately contained by the switch block may be labeled with one or more case or default labels. The switch statement evaluates its expression and executes the appropriate case.

Of course, you could also implement the same thing with if-then-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
```

Deciding whether to use if-then-else statements or a switch statement is sometimes a judgment call. You can decide which one to use based on readability and other factors. An if-then-else statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer or enumerated value.
Another point of interest is the break statement after each case. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, case statements fall through; that is, without an explicit break, control will flow sequentially through subsequent case statements. The following program, SwitchDemo2, illustrates why it might be useful to have case statements fall through:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
```

```
            case 12:
               numDays = 31;
               break;
            case 4:
            case 6:
            case 9:
            case 11:
               numDays = 30;
               break;
            case 2:
               if ( ((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0) )
                  numDays = 29;
               else
                  numDays = 28;
               break;
            default:
               System.out.println("Invalid month.");
               break;
         }
         System.out.println("Number of Days = " + numDays);
      }
   }
```

This is the output from the program.

```
   Number of Days = 29
```

3.7.6 THE Do-While STATEMENT

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The while statement evaluates expression, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
```

```
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
   // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

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

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count <= 11);
    }
}
```

3.7.7 THE for STATEMENT

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the for statement, keep in mind that:
The initialization expression initializes the loop; it's executed once, as the loop

begins.
When the termination expression evaluates to false, the loop terminates.
The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.
The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

The output of this program is:
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10

3.7.8 DEFINING METHODS IN JAVA

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines, double length, double grossTons) {
        //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.
More generally, method declarations have six components, in order:

Modifiers—such as public, private, and others you will learn about later.
The return type—the data type of the value returned by the method, or void if the method does not return a value.
The method name—the rules for field names apply to method names as well, but the convention is a little different.
The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters,

you must use empty parentheses.
An exception list—to be discussed later.
The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

3.7.8.1 NAME THE METHOD

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized.

- **OVERLOADING METHODS**

The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
        ...
        public void draw(String s) {
                ...
        }
        public void draw(int i) {
                ...
        }
        public void draw(double f) {
                ...
        }
        public void draw(int i, double f) {
                ...
        }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are

distinct and unique methods because they require different argument types.
You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

- **PROVIDING CONSTRUCTURES FOR THE CLASS**

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.
Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
        gear = 1;
        cadence = 10;
        speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must

verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

- **PASSING INFORMATION TO A METHOD**

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(double loanAmt,
                             double rate,
                             double futureValue,
                             int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest), -numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                    - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Creating Nested Class

The Java programming language allows you to define a class within another class. Such a class is called a nested class and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private.)

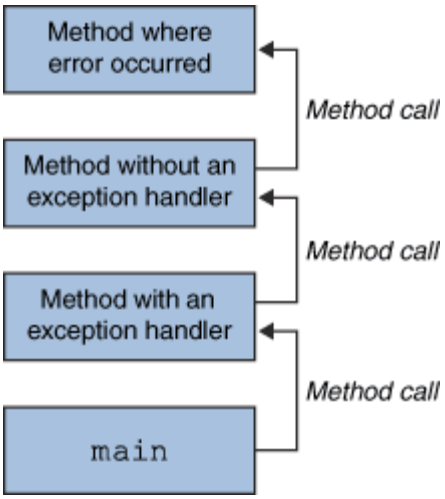
Exception in Java

What is Exception?

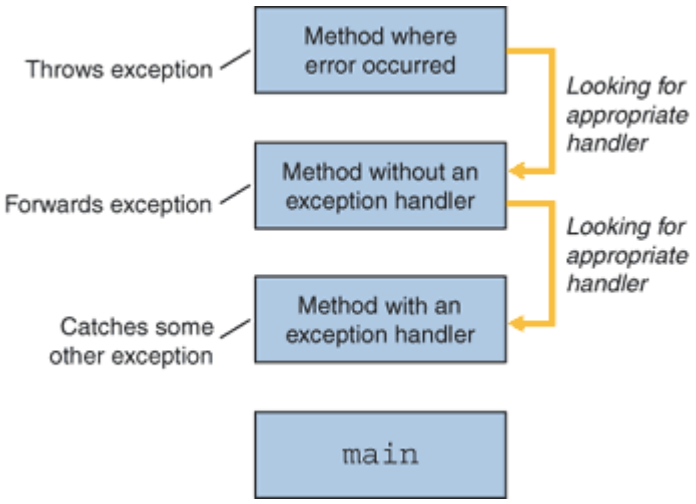
When an error occurs within a method, the method creates an object and hands it off

to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the next figure).



The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



3.7.9 THE CATCH OR SPECIFY THE REQUIREMENT

Valid Java programming language code must honour the Catch or Specify Requirement. This means that code that might throw certain exceptions must be enclosed by either of the following:
A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.
A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.
Code that fails to honor the Catch or Specify Requirement will not compile.
Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

- **The three kinds of exceptions:**

The first kind of exception is the checked exception. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name. Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

The second kind of exception is the error. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file

for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit. Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the runtime exception. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

3.7.9.1 How to throw the exception

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.
As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the Throwable class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

The ‘throw’ statement

All methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement.
throw someThrowableObject;

Let's look at the throw statement in context. The following pop method is taken from

a class that implements a common stack object. The method removes the top element from the stack and returns the object.

```
public Object pop() {
   Object obj;

   if (size == 0) {
      throw new EmptyStackException();
   }
   obj = objectAt(size - 1);
   setObjectAt(size - 1, null);
   size--;
   return obj;
}
```

The pop method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new EmptyStackException object (a member of java.util) and throws it. The Creating Exception Classes section in this chapter explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the java.lang.Throwable class.

Exception class

Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions as opposed to Errors.

The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur. For example, IllegalAccessException signals that a particular method could not be found, and NegativeArraySizeException indicates that a program attempted to create an array with a negative size.

One Exception subclass, RuntimeException, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is NullPointerException, which occurs when a method tries to access a member of an object through a null reference. The section Unchecked Exceptions — The Controversy discusses why most applications shouldn't throw runtime exceptions or subclass RuntimeException.

There's one more thing worth saying about buttons, which applies to any component that generates an action event. Java lets us specify an "action command" string for buttons (and other components, like menu items, that can generate action events). The action command is less interesting than it sounds. It is just a String that serves to identify the component that sent the event. By default, the action command of a JButton is the same as its label; it is included in action events, so you can use it to figure out which button an event came from.

To get the action command from an action event, call the event's getActionCommand( ) method. The following code checks whether the user pressed the Yes button:

```
public void actionPerformed(ActionEvent e){
   if (e.getActionCommand(  ).equals("Yes") {
      //the user pressed "Yes"; do something
      ...
   }
}
```

You can change the action command by calling the button's setActionCommand( ) method. The following code changes button myButton's action command to "confirm":

```
myButton.setActionCommand("confirm");
```

It's a good idea to get used to setting action commands explicitly; this helps to prevent your code from breaking when you or some other developer "internationalizes" it, or otherwise changes the button's label. If you rely on the button's label, your code will stop working as soon as that label changes; a French user might see the label Oui rather than Yes. By setting the action command, you eliminate one source of bugs; for example, the button myButton in the previous example will always generate the action command confirm, regardless of what its label says.

Swing buttons can have an image in addition to a label. The JButton class includes constructors that accept an Icon object, which knows how to draw itself. You can create buttons with captions, images, or both. A handy class called ImageIcon takes care of loading an image for you and can be used to easily add an image to a button. The following example shows how this works:

```
//file: PictureButton.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PictureButton extends JFrame {

  public PictureButton(  ) {
    super("PictureButton v1.0");
    setSize(200, 200);
    setLocation(200, 200);

    Icon icon = new ImageIcon("rhino.gif");
    JButton button = new JButton(icon);
    button.addActionListener(new ActionListener(  ) {
      public void actionPerformed(ActionEvent ae) {
        System.out.println("Urp!");
      }
    });

    Container content = getContentPane(  );
    content.setLayout(new FlowLayout(  ));
```

```
    content.add(button);
  }

  public static void main(String[] args) {
    JFrame f = new PictureButton(  );
    f.addWindowListener(new WindowAdapter(  ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setVisible(true);
  }
}
```

The example creates an ImageIcon from the rhino.gif file. Then a JButton is created from the ImageIcon. The whole thing is displayed in a JFrame. This example also shows the idiom of using an anonymous inner class as an ActionListener.

There's even less to be said about JLabel components. They're just text strings or images housed in a component. There aren't any special events associated with labels; about all you can do is specify the text's alignment, which controls the position of the text within the label's display area. As with buttons, JLabels can be created with Icons if you want to create a picture label. The following code creates some labels with different options:

```
// default alignment (CENTER)
JLabel label1 = new JLabel("Lions");

// left aligned
JLabel label2 = new JLabel("Tigers", SwingConstants.LEFT);

//label with no text, default alignment
JLabel label3 = new JLabel(  );

// create image icon
Icon icon = new ImageIcon("rhino.gif");

// create image label
JLabel label4 = new JLabel(icon);

// assigning text to label3
label3.setText("and Bears");

// set alignment
label3.setHorizontalAlignment(SwingConstants.RIGHT);
```

The alignment constants are defined in the SwingConstants interface.

Now we've built several labels, using a variety of constructors and several of the class's methods. To display the labels, just add them to a container by calling the container's add( ) method.

The other characteristics you might like to set on labels, such as changing their font or color, are accomplished using the methods of the Component class, JLabel's distant ancestor. For example, you can call setFont( ) and setColor( ) on a label, as with any other component.

Given that labels are so simple, why do we need them at all? Why not just draw a text string directly on the container object? Remember that a JLabel is a JComponent. That's important; it means that labels have the normal complement of methods for setting fonts and colors that we mentioned earlier, as well as the ability to be managed sensibly by a layout manager. Therefore, they're much more flexible than a text string drawn at an absolute location within a container.

Speaking of layouts--if you use the setText( ) method to change the text of your label, the label's preferred size may change. But the label's container will automatically lay out its components when this happens, so you don't have to worry about it.

Swing can interpret HTML-formatted text in JLabel and JButton labels. The following example shows how to create a button with HTML-formatted text:

```
JButton button = new JButton(
  "<html>"
  + "S<font size=-1>MALL<font size=+0> "
  + "C<font size=-1>APITALS");
```

3.7.10 AWT COMPONENT

3.7.10.1 CheckBoxes and Radio Buttons:

A checkbox is a labeled toggle switch. Each time the user clicks it, its state toggles between checked and unchecked. Swing implements the checkbox as a special kind of button. Radio buttons are similar to checkboxes, but they are usually arranged in groups. Click on one radio button in the group, and the others automatically turn off. They are named for the preset buttons on old car radios.

Checkboxes and radio buttons are represented by instances of JCheckBox and JRadioButton, respectively. Radio buttons can be tethered together using an instance of another class called ButtonGroup. By now you're probably well into the swing of things (no pun intended) and could easily master these classes on your own. We'll use an example to illustrate a different way of dealing with the state of components and to show off a few more things about containers.

A JCheckBox sends ItemEvents when it's pushed. Since a checkbox is a kind of button, it also fires ActionEvents when it becomes checked. For something like a checkbox, we might want to be lazy and check on the state of the buttons only at some later time, such as when the user commits an action. It's like filling out a form; you can change your choices until you submit the form.

The following application, DriveThrough, lets us check off selections on a fast food menu, as shown in Figure 14-1. DriveThrough prints the results when we press the Place Order button. Therefore, we can ignore all the events generated by our checkboxes and radio buttons and listen only for the action events generated by the regular button.

```
//file: DriveThrough.java
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class DriveThrough {
  public static void main(String[] args) {
    JFrame f = new JFrame("Lister v1.0");
    f.setSize(300, 150);
    f.setLocation(200, 200);
    f.addWindowListener(new WindowAdapter( ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });

    JPanel entreePanel = new JPanel( );
    final ButtonGroup entreeGroup = new ButtonGroup( );
    JRadioButton radioButton;
    entreePanel.add(radioButton = new JRadioButton("Beef"));
    radioButton.setActionCommand("Beef");
    entreeGroup.add(radioButton);
    entreePanel.add(radioButton = new JRadioButton("Chicken"));
    radioButton.setActionCommand("Chicken");
    entreeGroup.add(radioButton);
    entreePanel.add(radioButton = new JRadioButton("Veggie", true));
    radioButton.setActionCommand("Veggie");
    entreeGroup.add(radioButton);

    final JPanel condimentsPanel = new JPanel( );
    condimentsPanel.add(new JCheckBox("Ketchup"));
    condimentsPanel.add(new JCheckBox("Mustard"));
    condimentsPanel.add(new JCheckBox("Pickles"));

    JPanel orderPanel = new JPanel( );
    JButton orderButton = new JButton("Place Order");
    orderPanel.add(orderButton);

    Container content = f.getContentPane( );
    content.setLayout(new GridLayout(3, 1));
    content.add(entreePanel);
    content.add(condimentsPanel);
    content.add(orderPanel);

    orderButton.addActionListener(new ActionListener( ) {
```

```
    public void actionPerformed(ActionEvent ae) {
      String entree =
        entreeGroup.getSelection().getActionCommand( );
      System.out.println(entree + " sandwich");
      Component[] components = condimentsPanel.getComponents( );
      for (int i = 0; i < components.length; i++) {
        JCheckBox cb = (JCheckBox)components[i];
        if (cb.isSelected( ))
          System.out.println("With " + cb.getText( ));
      }
    }
  });

  f.setVisible(true);
 }
}
```

Fig 3.17 Radio Button and CheckBox

DriveThrough lays out three panels. The radio buttons in the entreePanel are tied together through a ButtonGroup object. We add( ) the buttons to a ButtonGroup to make them mutually exclusive. The ButtonGroup object is an odd animal. One expects it to be a container or a component, but it isn't; it's simply a helper object that allows only one RadioButton to be selected at a time.

In this example, the button group forces you to choose a beef, chicken, or veggie entree, but not more than one. The condiment choices, which are JCheckBoxes, aren't in a button group, so you can request any combination of ketchup, mustard, and pickles on your sandwich.

When the Place Order button is pushed, we receive an ActionEvent in the actionPerformed( ) method of our inner ActionListener. At this point, we gather the information in the radio buttons and checkboxes and print it. actionPerformed( ) simply reads the state of the various buttons. We could have saved references to the buttons in a number of ways; this example demonstrates two. First, we find out which entree was selected. To do so, we call the ButtonGroup's getSelection( ) method. This returns a ButtonModel, upon which we immediately call getActionCommand( ). This returns the action command as we set it when we created the radio buttons. The action commands for the buttons are the entrée names, which is exactly what we need.

To find out which condiments were selected, we use a more complicated procedure. The problem is that condiments aren't mutually exclusive, so we don't have the

convenience of a ButtonGroup. Instead, we ask the condiments JPanel for a list of its components. The getComponents( ) method returns an array of references to the container's child components. We'll use this to loop over the components and print the results. We cast each element of the array back to JCheckBox and call its isSelected( ) method to see if the checkbox is on or off. If we were dealing with different types of components in the array, we could determine each component's type with the instanceof operator.

3.7.10.2 List and Combo Boxes:

JLists and JComboBoxes are a step up on the evolutionary chain from JButtons and JLabels. Lists let the user choose from a group of alternatives. They can be configured to force the user to choose a single selection or to allow multiple choices. Usually, only a small group of choices are displayed at a time; a scrollbar lets the user move to the choices that aren't visible. The user can select an item by clicking on it. He or she can expand the selection to a range of items by holding down Shift and clicking on another item. To make discontinuous selections, the user can hold down the Control key instead of the Shift key.

A combo box is a cross-breed between a text field and a list. It displays a single line of text (possibly with an image) and a downward pointing arrow at one side. If you click on the arrow, the combo box opens up and displays a list of choices. You can select a single choice by clicking on it. After a selection is made, the combo box closes up; the list disappears and the new selection is shown in the text field.

Like every other component in Swing, lists and combo boxes have data models that are distinct from visual components. The list also has a selection model that controls how selections may be made on the list data.

Lists and combo boxes are similar because they have similar data models. Each is simply an array of acceptable choices. This similarity is reflected in Swing, of course: the type of a JComboBox's data model is a subclass of the type used for a JList's data model. The next example demonstrates this relationship.

The following example creates a window with a combo box, a list, and a button. The combo box and the list use the same data model. When you press the button, the program writes out the current set of selected items in the list. Figure 14-2 shows the example; the code itself follows.

Fig 3.18 JList

```
/file: Lister.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Lister {
  public static void main(String[] args) {
    JFrame f = new JFrame("Lister v1.0");
    f.setSize(200, 200);
    f.setLocation(200, 200);
    f.addWindowListener(new WindowAdapter(  ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });

    // create a combo box
    String [] items = { "uno", "due", "tre", "quattro", "cinque",
                        "sei", "sette", "otto", "nove", "deici",
                        "undici", "dodici" };
    JComboBox comboBox = new JComboBox(items);
    comboBox.setEditable(true);

    // create a list with the same data model
    final JList list = new JList(comboBox.getModel(  ));

    // create a button; when it's pressed, print out
    // the selection in the list
    JButton button = new JButton("Per favore");
    button.addActionListener(new ActionListener(  ) {
      public void actionPerformed(ActionEvent ae) {
        Object[] selection = list.getSelectedValues(  );
        System.out.println("-----");
        for (int i = 0; i < selection.length; i++)
          System.out.println(selection[i]);
      }
    });

    // put the controls the content pane
    Container c = f.getContentPane(  );
    JPanel comboPanel = new JPanel(  );
    comboPanel.add(comboBox);
    c.add(comboPanel, BorderLayout.NORTH);
    c.add(new JScrollPane(list), BorderLayout.CENTER);
    c.add(button, BorderLayout.SOUTH);

    f.setVisible(true);
  }
}
```

The combo box is created from an array of strings. This is a convenience--behind the scenes, the JComboBox constructor creates a data model from the strings you supply

and sets the JComboBox to use that data model. The list is created using the data model of the combo box. This works because JList expects to use a ListModel for its data model, and the ComboBoxModel used by the JComboBox is a subclass of ListModel.

The button's action event handler simply prints out the selected items in the list, which are retrieved with a call to getSelectedValues( ). This method actually returns an object array, not a string array. List and combo box items, like many other things in Swing, are not limited to text. You can use images, or drawings, or some combination of text and images.

You might expect that selecting one item in the combo box would select the same item in the list. In Swing components, selection is controlled by a selection model. The combo box and the list have distinct selection models; after all, you can select only one item from the combo box, while it's possible to select multiple items from the list. Thus, while the two components share a data model, they have separate selection models.

We've made the combo box editable. By default, it would not be editable: the user could choose only one of the items in the drop-down list. With an editable combo box, the user can type in a selection, as if it were a text field. Non-editable combo boxes are useful if you just want to offer a limited set of choices; editable combo boxes are handy when you want to accept any input but offer some common choices.

There's a great class tucked away in the last example that deserves some recognition. It's JScrollPane. In Lister, you'll notice we created one when we added the List to the main window.

JScrollPane simply wraps itself around another Component and provides scrollbars as necessary. The scrollbars show up if the contained Component's preferred size (as returned by getPreferredSize( )) is greater than the size of the JScrollPane itself. In the previous example, the scrollbars show up whenever the size of the List exceeds the available space.

You can use JScrollPane to wrap any Component, including components with drawings or images or complex user interface panels

3.7.11 Borders:

Any Swing component can have a decorative border. JComponent includes a method called setBorder( ); all you have to do is call setBorder( ), passing it an appropriate implementation of the Border interface.

Swing provides many useful Border implementations in the javax.swing.border package. You could create an instance of one of these classes and pass it to a component's setBorder( ) method, but there's an even simpler technique.

The BorderFactory class can create any kind of border for you using static "factory" methods. Creating and setting a component's border, then, is simple:

```
JLabel labelTwo = new JLabel("I have an etched border.");
labelTwo.setBorder(BorderFactory.createEtchedBorder(  ));
```

Every component has a setBorder( ) method, from simple labels and buttons right up to the fancy text and table components.

BorderFactory is convenient, but it does not offer every option of every border type. For example, if you want to create a raised EtchedBorder instead of the default lowered border, you'll need to use EtchedBorder's constructor rather than a method in BorderFactory, like this:

```
JLabel labelTwo = new JLabel("I have a raised etched border.");
labelTwo.setBorder( new EtchedBorder(EtchedBorder.RAISED) );
```

The Border implementation classes are listed and briefly described here:

- BevelBorder
  This border draws raised or lowered beveled edges, giving an illusion of depth.

- SoftBevelBorder
  This border is similar to BevelBorder, but thinner.

- EmptyBorder
  Doesn't do any drawing, but does take up space. You can use it to give a component a little breathing room in a crowded user interface.

- EtchedBorder
  A lowered etched border gives the appearance of a rectangle that has been chiseled into a piece of stone. A raised etched border looks like it is standing out from the surface of the screen.

- LineBorder
  Draws a simple rectangle around a component. You can specify the color and width of the line in LineBorder's constructor.

- MatteBorder
  A souped-up version of LineBorder. You can create a MatteBorder with a certain color and specify the size of the border on the left, top, right, and bottom of the component. MatteBorder also allows you to pass in an Icon that will be used to draw the border. This could be an image (ImageIcon) or any other implementation of the Icon interface.

- TitledBorder
  A regular border with a title. TitledBorder doesn't actually draw a border; it just draws a title in conjunction with another border object. You can specify the locations of the title, its justification, and its font. This border type is particularly useful for grouping different sets of controls in a complicated interface.

- CompoundBorder
  A border that contains two other borders. This is especially handy if you want to enclose a component in an EmptyBorder and then put something decorative around it, like an EtchedBorder or a MatteBorder.

  The following example shows off some different border types. It's only a sampler, though; many more border types are available. Furthermore, the example only encloses labels with borders. You can put a border around any component in Swing. The example is shown in Figure 14-3; the source code follows.

Fig 3.19 Borders

```
//file: Borders.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Borders {
  public static void main(String[] args) {
    // create a JFrame to hold everything
    JFrame f = new JFrame("Borders");
    f.addWindowListener(new WindowAdapter(  ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setSize(300, 300);
    f.setLocation(200, 200);
```

```
    // Create labels with borders.
    int center = SwingConstants.CENTER;
    JLabel labelOne = new JLabel("raised BevelBorder", center);
    labelOne.setBorder(
       BorderFactory.createBevelBorder(BevelBorder.RAISED));
    JLabel labelTwo = new JLabel("EtchedBorder", center);
    labelTwo.setBorder(BorderFactory.createEtchedBorder( ));
    JLabel labelThree = new JLabel("MatteBorder", center);
    labelThree.setBorder(
       BorderFactory.createMatteBorder(10, 10, 10, 10, Color.pink));
    JLabel labelFour = new JLabel("TitledBorder", center);
    Border etch = BorderFactory.createEtchedBorder( );
    labelFour.setBorder(
       BorderFactory.createTitledBorder(etch, "Title"));
    JLabel labelFive = new JLabel("TitledBorder", center);
    Border low = BorderFactory.createLoweredBevelBorder( );
    labelFive.setBorder(
       BorderFactory.createTitledBorder(low, "Title",
       TitledBorder.RIGHT, TitledBorder.BOTTOM));
    JLabel labelSix = new JLabel("CompoundBorder", center);
    Border one = BorderFactory.createEtchedBorder( );
    Border two =
       BorderFactory.createMatteBorder(4, 4, 4, 4, Color.blue);
    labelSix.setBorder(BorderFactory.createCompoundBorder(one, two));

    // add components to the content pane
    Container c = f.getContentPane( );
    c.setLayout(new GridLayout(3, 2));









    c.add(labelOne);
    c.add(labelTwo);
    c.add(labelThree);
    c.add(labelFour);
    c.add(labelFive);
    c.add(labelSix);

    f.setVisible(true);
  }
}
```

3.7.12Menus:

A JMenu is a standard pull-down menu with a fixed name. Menus can hold other

menus as submenu items, enabling you to implement complex menu structures. In Swing, menus are first-class components, just like everything else. You can place them wherever a component would go. Another class, JMenuBar, holds menus in a horizontal bar. Menu bars are real components, too, so you can place them wherever you want in a container: top, bottom, or middle. But in the middle of a container, it usually makes more sense to use a JComboBox rather than some kind of menu.

Menu items may have associated images and shortcut keys; there are even menu items that look like checkboxes and radio buttons. Menu items are really a kind of button. Like buttons, menu items fire action events when they are selected. You can respond to menu items by registering action listeners with them.

There are two ways to use the keyboard with menus. The first is called mnemonics. A mnemonic is one character in the menu name. If you hold down the Alt key and type a menu's mnemonic, the menu will drop down, just as if you had clicked on it with the mouse. Menu items may also have mnemonics. Once a menu is dropped down, you can select individual items in the same way.

Menu items may also have accelerators. An accelerator is a key combination that selects the menu item, whether or not the menu that contains it is showing. A common example is the accelerator Ctrl-C, which is frequently used as a shortcut for the Copy item in the Edit menu.

The following example demonstrates several different features of menus. It creates a menu bar with three different menus. The first, Utensils, contains several menu items, a submenu, a separator, and a Quit item that includes both a mnemonic and an accelerator. The second menu, Spices, contains menu items that look and act like checkboxes. Finally, the Cheese menu demonstrates how radio button menu items can be used.

This application is shown in Figure 14-4 with one of its menus dropped down. Choosing Quit from the menu (or pressing Ctrl-Q) removes the window. Give it a try.

```
//file: DinnerMenu.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DinnerMenu extends JFrame {

  public DinnerMenu(  ) {
    super("DinnerMenu v1.0");
    setSize(200, 200);
    setLocation(200, 200);

    // create the Utensils menu
    JMenu utensils = new JMenu("Utensils");
    utensils.setMnemonic(KeyEvent.VK_U);
    utensils.add(new JMenuItem("Fork"));
```

```
  utensils.add(new JMenuItem("Knife"));
  utensils.add(new JMenuItem("Spoon"));
  JMenu hybrid = new JMenu("Hybrid");
  hybrid.add(new JMenuItem("Spork"));
  hybrid.add(new JMenuItem("Spife"));
  hybrid.add(new JMenuItem("Knork"));
  utensils.add(hybrid);
  utensils.addSeparator( );

  // do some fancy stuff with the Quit item
  JMenuItem quitItem = new JMenuItem("Quit");
  quitItem.setMnemonic(KeyEvent.VK_Q);
  quitItem.setAccelerator(
     KeyStroke.getKeyStroke(KeyEvent.VK_Q, Event.CTRL_MASK));
  quitItem.addActionListener(new ActionListener( ) {
   public void actionPerformed(ActionEvent e) { System.exit(0); }
  });
  utensils.add(quitItem);

  // create the Spices menu
  JMenu spices = new JMenu("Spices");
  spices.setMnemonic(KeyEvent.VK_S);
  spices.add(new JCheckBoxMenuItem("Thyme"));
  spices.add(new JCheckBoxMenuItem("Rosemary"));
  spices.add(new JCheckBoxMenuItem("Oregano", true));
  spices.add(new JCheckBoxMenuItem("Fennel"));

  // create the Cheese menu
  JMenu cheese = new JMenu("Cheese");
  cheese.setMnemonic(KeyEvent.VK_C);
  ButtonGroup group = new ButtonGroup( );
  JRadioButtonMenuItem rbmi;
  rbmi = new JRadioButtonMenuItem("Regular", true);
  group.add(rbmi);
  cheese.add(rbmi);
  rbmi = new JRadioButtonMenuItem("Extra");
  group.add(rbmi);
  cheese.add(rbmi);
  rbmi = new JRadioButtonMenuItem("Blue");
  group.add(rbmi);
  cheese.add(rbmi);

  // create a menu bar and use it in this JFrame
  JMenuBar menuBar = new JMenuBar( );
  menuBar.add(utensils);
  menuBar.add(spices);
  menuBar.add(cheese);
  setJMenuBar(menuBar);
}
public static void main(String[] args) {
```

```
    JFrame f = new DinnerMenu(  );
    f.addWindowListener(new WindowAdapter(  ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setVisible(true);
  }
}
```

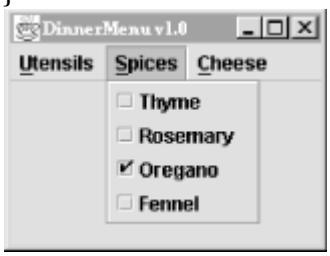


Fig 3.20 Menu and MenuItems

JTabbed Pane Class:

If you've ever dealt with the System control panel in Windows, you already know what a JTabbedPane is. It's a container with labeled tabs. When you click on a tab, a new set of controls is shown in the body of the JTabbedPane. In Swing, JTabbedPane is simply a specialized container.

Each tab has a name. To add a tab to the JTabbedPane, simply call addTab( ). You'll need to specify the name of the tab as well as a component that supplies the tab's contents. Typically, it's a container holding other components.

Even though the JTabbedPane only shows one set of components at a time, be aware that all the components on all the pages are in memory at one time. If you have components that hog processor time or memory, try to put them into some "sleep" state when they are not showing.

The following example shows how to create a JTabbedPane. It adds standard Swing components to a first tab, named Controls. The second tab is filled with an instance of ImageComponent, which was presented earlier in this chapter.

```
//file: TabbedPaneFrame.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class TabbedPaneFrame {
  public static void main(String[] args) {
    // create a JFrame to hold everything
    JFrame f = new JFrame("TabbedPaneFrame");
    f.addWindowListener(new WindowAdapter(  ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
```

```
    f.setSize(200, 200);
    f.setLocation(200, 200);

    JTabbedPane tabby = new JTabbedPane(  );

    // create a controls pane
    JPanel controls = new JPanel(  );
    controls.add(new JLabel("Service:"));
    JList list = new JList(
       new String[] { "Web server", "FTP server" });
    list.setBorder(BorderFactory.createEtchedBorder(  ));
    controls.add(list);
    controls.add(new JButton("Start"));

    // create an image pane
    String filename = "Piazza di Spagna.jpg";
    Image image = Toolkit.getDefaultToolkit(  ).getImage(filename);
    JComponent picture = new JScrollPane(new ImageComponent(image));

    tabby.addTab("Controls", controls);
    tabby.addTab("Picture", picture);

    f.getContentPane(  ).add(tabby);
    f.setVisible(true);
  }
}
```

If you've ever dealt with the System control panel in Windows, you already know what a JTabbedPane is. It's a container with labeled tabs. When you click on a tab, a new set of controls is shown in the body of the JTabbedPane. In Swing, JTabbedPane is simply a specialized container.

Each tab has a name. To add a tab to the JTabbedPane, simply call addTab( ). You'll need to specify the name of the tab as well as a component that supplies the tab's contents. Typically, it's a container holding other components.

Even though the JTabbedPane only shows one set of components at a time, be aware that all the components on all the pages are in memory at one time. If you have components that hog processor time or memory, try to put them into some "sleep" state when they are not showing.

The following example shows how to create a JTabbedPane. It adds standard Swing components to a first tab, named Controls. The second tab is filled with an instance of ImageComponent, which was presented earlier in this chapter.

```
//file: TabbedPaneFrame.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
```

```
public class TabbedPaneFrame {
  public static void main(String[] args) {
    // create a JFrame to hold everything
    JFrame f = new JFrame("TabbedPaneFrame");
    f.addWindowListener(new WindowAdapter( ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setSize(200, 200);
    f.setLocation(200, 200);

    JTabbedPane tabby = new JTabbedPane( );

    // create a controls pane
    JPanel controls = new JPanel( );
    controls.add(new JLabel("Service:"));
    JList list = new JList(
        new String[] { "Web server", "FTP server" });
    list.setBorder(BorderFactory.createEtchedBorder( ));
    controls.add(list);
    controls.add(new JButton("Start"));

    // create an image pane
    String filename = "Piazza di Spagna.jpg";
    Image image = Toolkit.getDefaultToolkit( ).getImage(filename);
    JComponent picture = new JScrollPane(new ImageComponent(image));
    tabby.addTab("Controls", controls);
    tabby.addTab("Picture", picture);
    f.getContentPane( ).add(tabby);
    f.setVisible(true);
  }
}
```

Fig 3.21 Tab Panes

3.7.13 Event Handling in Java

In computer programming, an event handler is an asynchronous callback subroutine that handles inputs received in a program. Each event is a piece of application-level information from the underlying framework, typically the GUI toolkit. GUI events include key presses, mouse movement, action selections, and timers expiring. On a lower level, events can represent availability of new data for reading a file or

network stream. Event handlers are a central concept in event-driven programming.

The events are created by the framework based on interpreting lower-level inputs, which may be lower-level events themselves. For example, mouse movements and clicks are interpreted as menu selections. The events initially originate from actions on the operating system level, such as interrupts generated by hardware devices, software interrupt instructions, or state changes in polling. On this level, interrupt handlers and signal handlers correspond to event handlers.

Created events are first processed by an event dispatcher within the framework. It typically manages the associations between events and event handlers, and may queue event handlers or events for later processing. Event dispatchers may call event handlers directly, or wait for events to be dequeued with information about the handler to be executed.

An event handler requires a single piece of information: a reference to the instance of the Event class containing information about the event that just occurred.

The value returned from the handleEvent() method is important. It indicates to the Java run-time system whether or not the event has been completely handled within the event handler. A true value indicates that the event has been handled and propagation should stop. A false value indicates that the event has been ignored, could not be handled, or has been handled incompletely and should continue up the tree.

How to write an ActionListener:

Action listeners are probably the easiest — and most common — event handlers to implement. You implement an action listener to define what should be done when an user performs certain operation.

An action event occurs, whenever an action is performed by the user. Examples: When the user clicks a button, chooses a menu item, presses Enter in a text field. The result is that an actionPerformed message is sent to all action listeners that are registered on the relevant component.

To write an Action Listener, follow the steps given below:

Declare an event handler class and specify that the class either implements an ActionListener interface or extends a class that implements an ActionListener interface. For example:

```
public class MyClass implements ActionListener {
```

Register an instance of the event handler class as a listener on one or more components. For example:

```
someComponent.addActionListener(instanceOfMyClass);
```

Include code that implements the methods in listener interface. For example:

```
public void actionPerformed(ActionEvent e) {
   ...//code that reacts to the action...
}
```

In general, to detect when the user clicks an onscreen button (or does the keyboard equivalent), a program must have an object that implements the ActionListener interface. The program must register this object as an action listener on the button (the event source), using the addActionListener method. When the user clicks the onscreen button, the button fires an action event. This results in the invocation of the action listener's actionPerformed method (the only method in the ActionListener interface). The single argument to the method is an ActionEvent object that gives information about the event and its source.
Let us write a simple program which displays how many number of times a button is clicked by the user. First, here is the code that sets up the TextField , button and numClicks variable:

```
public class AL extends Frame implements WindowListener,ActionListener {
TextField text = new TextField(20);
Button b;
private int numClicks = 0;
```

In the above example, the event handler class is AL which implements ActionListener.

We would like to handle the button-click event, so we add an action listener to the button b as below:

```
b = new Button("Click me");
b.addActionListener(this);
```

In the above code, Button b is a component upon which an instance of event handler class AL is registered.
Now, we want to display the text as to how many number of times a user clicked button. We can do this by writing the code as below:

```
public void actionPerformed(ActionEvent e) {
        numClicks++;
        text.setText("Button Clicked " + numClicks + " times");
```

Now, when the user clicks the Button b, the button fires an action event which invokes the action listener's actionPerformed method. Each time the user presses the button, numClicks variable is appended and the message is displayed in the text field.
Here is the complete program(AL.java):

```
import java.awt.*;
import java.awt.event.*;
public class AL extends Frame implements WindowListener,ActionListener {
        TextField text = new TextField(20);
        Button b;
        private int numClicks = 0;
        public static void main(String[] args) {
                AL myWindow = new AL("My first window");
                myWindow.setSize(350,100);
                myWindow.setVisible(true);
        }
        public AL(String title) {

                super(title);
```

```
                setLayout(new FlowLayout());
                addWindowListener(this);
                b = new Button("Click me");
                add(b);
                add(text);
                b.addActionListener(this);
        }
        public void actionPerformed(ActionEvent e) {
                numClicks++;
                text.setText("Button Clicked " + numClicks + " times");
        }
        public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
        }
        public void windowOpened(WindowEvent e) {}
        public void windowActivated(WindowEvent e) {}
        public void windowIconified(WindowEvent e) {}
        public void windowDeiconified(WindowEvent e) {}
        public void windowDeactivated(WindowEvent e) {}
        public void windowClosed(WindowEvent e) {}
}
```

3.7.13.1 The MouseListener Interface

This type of mouse listener is for events which typically don't happen very often -- a mouse button is pressed, released, or the mouse enters or leaves the area of the component with a listener. Here are the actions that a MouseListener catches. press one of the mouse buttons is pressed.
release one of the mouse buttons is released.
click a mouse button was pressed and released without moving the mouse. This is perhaps the most commonly used.
enter mouse cursor enters the component. Often used to change cursor.
exit mouse cursor exits the component. Often used to restore cursor. To listen for these events you will use addMouseListener.
MouseListener Interface
To implement a MouseListener interface, you must define the following methods. You can copy these definitions into your program and only make a meaningful body for those methods that are of interest.

```
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
```

This method is called	*When the user does this action*
mouseClicked	A *click* is the result of a press and a release. This is probably the most common method to write.
mousePressed	A mouse button is pressed (any of three possible mouse buttons)
mouseReleased	A mouse button is released.
mouseEntered	The mouse cursor enters a component. You might write this to change the cursor.
mouseExited	The mouse cursor leaves a component. You might write this to restore the cursor.

To Get the Mouse Coordinates
All coordinates are relative to the upper left corner of the component with the mouse listener. Use the following MouseEvent methods to get x and y coordinates of where the mouse event occurred.
int getX() // returns the x coordinate of the event.
int getY() // returns the y coordinate of the event.
To Check for Double Clicks
Use the following MouseEvent method to get a count of the number of clicks.
int getClickCount() // number of mouse clicks

3.7.14 JAVA MEDIA FRAMEWORK

3.7.14.1 Understanding JMF

Java™ Media Framework (JMF) provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF is designed to support most standard media content types, such as AIFF, AU, AVI, GSM, MIDI, MPEG, QuickTime, RMF, and WAV.

By exploiting the advantages of the Java platform, JMF delivers the promise of "Write Once, Run Anywhere™" to developers who want to use media such as audio and video in their Java programs. JMF provides a common cross-platform Java API for accessing underlying media frameworks. JMF implementations can leverage the capabilities of the underlying operating system, while developers can easily create portable Java programs that feature time-based media by writing to the JMF API.

With JMF, you can easily create applets and applications that present, capture, manipulate, and store time-based media. The framework enables advanced developers and technology providers to perform custom processing of raw media data and seamlessly extend JMF to support additional content types and formats, optimize handling of supported formats, and create new presentation mechanisms.

3.7.14.2 High Level Architecture

Devices such as tape decks and VCRs provide a familiar model for recording, processing, and presenting time-based media. When you play a movie using a VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads

and interprets the data on the tape and sends appropriate signals to your television and speakers.

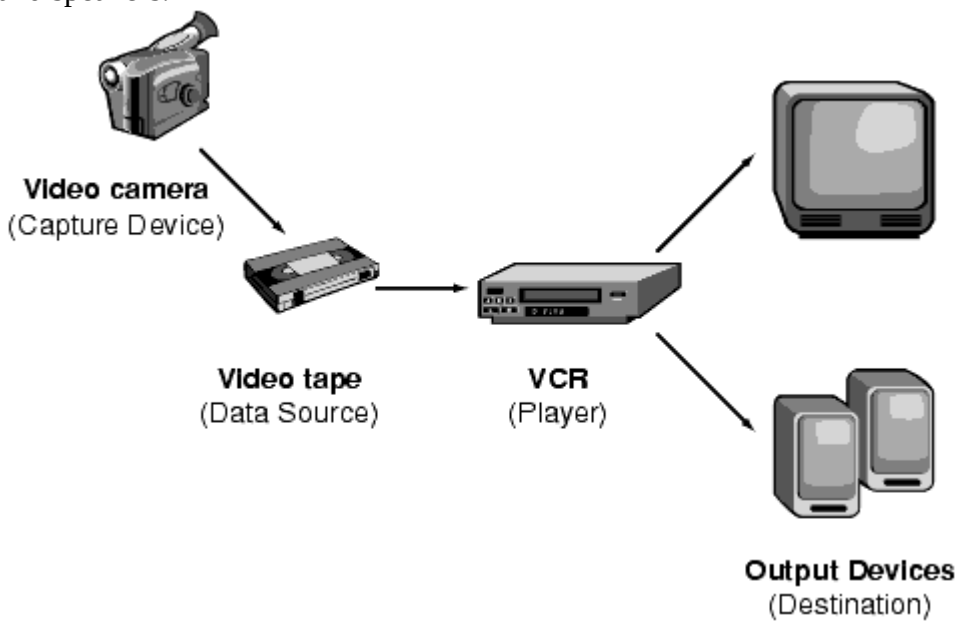


Figure 3.22: Recording, processing, and presenting time-based media.

JMF uses this same basic model. A *data source* encapsulates the media stream much like a video tape and a *player* provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.
Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media. JMF also provides a lower-level API that supports the seamless integration of custom processing components and extensions. This layering provides Java developers with an easy-to-use API for incorporating time-based media into Java programs while maintaining the flexibility and extensibility required to support advanced media applications and future media technologies.

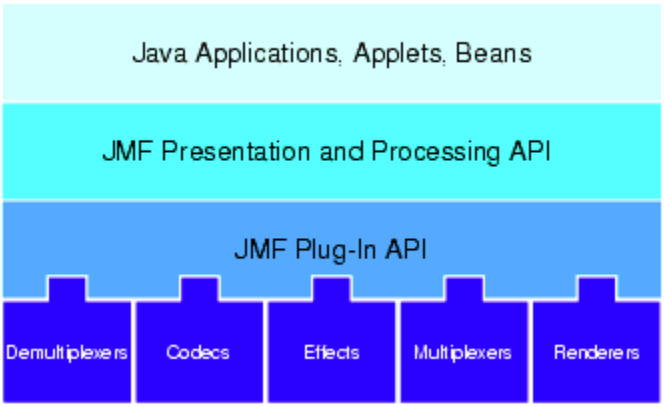


Fig 3.23: High-level JMF achitecture.

3.7.14.3 Time Model

JMF keeps time to nanosecond precision. A particular point in time is typically represented by a `Time` object, though some classes also support the specification of time in nanoseconds.

Classes that support the JMF time model implement `Clock` to keep track of time for a particular media stream. The `Clock` interface defines the basic timing and synchronization operations that are needed to control the presentation of media data.

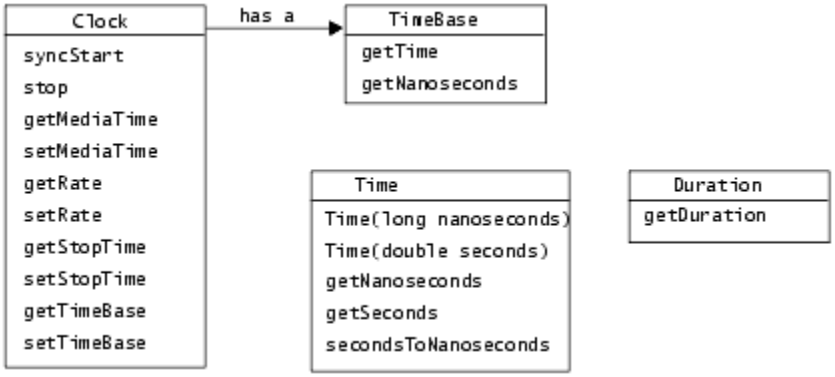


Figure 3.24: JMF time model.

A `Clock` uses a `TimeBase` to keep track of the passage of time while a media stream is being presented. A `TimeBase` provides a constantly ticking time source, much like a crystal oscillator in a watch. The only information that a `TimeBase` provides is its current time, which is referred to as the *time-base time*. The time-base time cannot be stopped or reset. Time-base time is often based on the system clock.

A `Clock` object's *media time* represents the current position within a media stream--the beginning of the stream is media time zero, the end of the stream is the maximum media time for the stream. The *duration* of the media stream is the elapsed time from start to finish--the length of time that it takes to present the media stream. (Media objects implement the `Duration` interface if they can report a media stream's duration.)

To keep track of the current media time, a `Clock` uses:

- The time-base start-time--the time that its `TimeBase` reports when the presentation begins.
- The media start-time--the position in the media stream where presentation begins.
- The playback rate--how fast the `Clock` is running in relation to its `TimeBase`. The *rate* is a scale factor that is applied to the `TimeBase`. For example, a rate of 1.0 represents the normal playback rate for the media stream, while a rate of 2.0 indicates that the presentation will run at twice the normal rate. A negative rate indicates that the `Clock` is running in the opposite direction from its `TimeBase`--for example, a negative rate might be used to play a media stream backward.

When presentation begins, the media time is mapped to the time-base time and the advancement of the time-base time is used to measure the passage of time. During presentation, the current media time is calculated using the following formula:

MediaTime = MediaStartTime + Rate(TimeBaseTime - TimeBaseStartTime)

When the presentation stops, the media time stops, but the time-base time continues to advance. If the presentation is restarted, the media time is remapped to the current time-base time.

3.7.14.4 Manager

The JMF API consists mainly of interfaces that define the behavior and interaction of objects used to capture, process, and present time-based media. Implementations of these interfaces operate within the structure of the framework. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes.
JMF uses four managers:
`Manager`--handles the construction of `Players`, `Processors`, `DataSources`, and `DataSinks`. This level of indirection allows new implementations to be integrated seamlessly with JMF. From the client perspective, these objects are always created the same way whether the requested object is constructed from a default implementation or a custom one.
`PackageManager`--maintains a registry of packages that contain JMF classes, such as custom `Players`, `Processors`, `DataSources`, and `DataSinks`.
`CaptureDeviceManager`--maintains a registry of available capture devices.
`PlugInManager`--maintains a registry of available JMF plug-in processing components, such as `Multiplexers`, `Demultiplexers`, `Codecs`, `Effects`, and `Renderers`.
To write programs based on JMF, you'll need to use the `Manager create` methods to construct the `Players`, `Processors`, `DataSources`, and `DataSinks` for your application. If you're capturing media data from an input device, you'll use the `CaptureDeviceManager` to find out what devices are available and access information about them. If you're interested in controlling what processing is performed on the data, you might also query the `PlugInManager` to determine what plug-ins have been registered.
If you extend JMF functionality by implementing a new plug-in, you can register it with the `PlugInManager` to make it available to `Processors` that support the plug-in API. To use a custom `Player`, `Processor`, `DataSource`, or `DataSink` with `JMF`, you register your unique package prefix with the `PackageManager`.

3.7.14.5 JMF EVENT MODEL

JMF uses a structured event reporting mechanism to keep JMF-based programs informed of the current state of the media system and enable JMF-based programs to

respond to media-driven error conditions, such as out-of data and resource unavailable conditions. Whenever a JMF object needs to report on the current conditions, it posts a MediaEvent. MediaEvent is subclassed to identify many particular types of events. These objects follow the established Java Beans patterns for events.
For each type of JMF object that can post MediaEvents, JMF defines a corresponding listener interface. To receive notification when a MediaEvent is posted, you implement the appropriate listener interface and register your listener class with the object that posts the event by calling its add*Listener* method.
Controller objects (such as Players and Processors) and certain Control objects such as GainControl post media events.

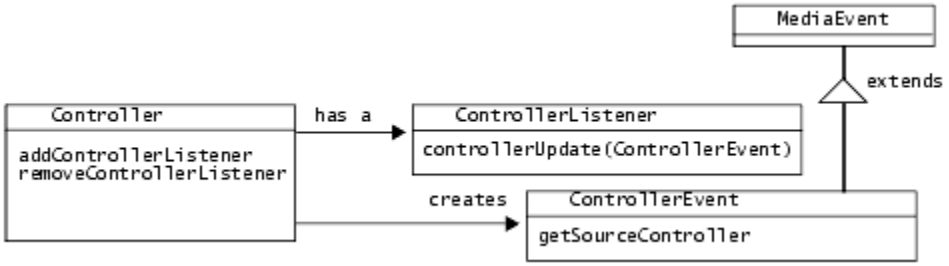


Figure 3.25: JMF event model.

3.7.14.6 User Interface Components

A Control can provide access to a user interface Component that exposes its control behavior to the end user. To get the default user interface component for a particular Control, you call getControlComponent. This method returns an AWT Component that you can add to your applet's presentation space or application window.
A Controller might also provide access to user interface Components. For example, a Player provides access to both a visual component and a control panel component--to retrieve these components, you call the Player methods getVisualComponent and getControlPanelComponent.
If you don't want to use the default control components provided by a particular implementation, you can implement your own and use the event listener mechanism to determine when they need to be updated. For example, you might implement your own GUI components that support user interaction with a Player. Actions on your GUI components would trigger calls to the appropriate Player methods, such as start and stop. By registering your custom GUI components as ControllerListeners for the Player, you can also update your GUI in response to changes in the Player object's state.

3.7.14.7 Players

A Player processes an input stream of media data and renders it at a precise time. A DataSource is used to deliver the input media-stream to the Player.The rendering destination depends on the type of media being presented.

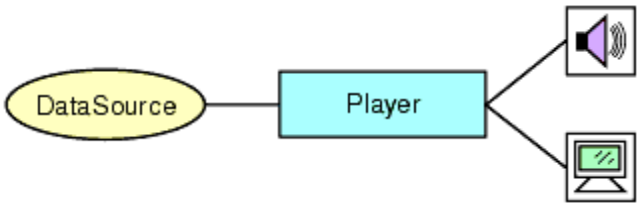


Figure 3.26: JMF player model.

A Player does not provide any control over the processing that it performs or how it renders the media data.
Player supports standardized user control and relaxes some of the operational restrictions imposed by Clock and Controller.

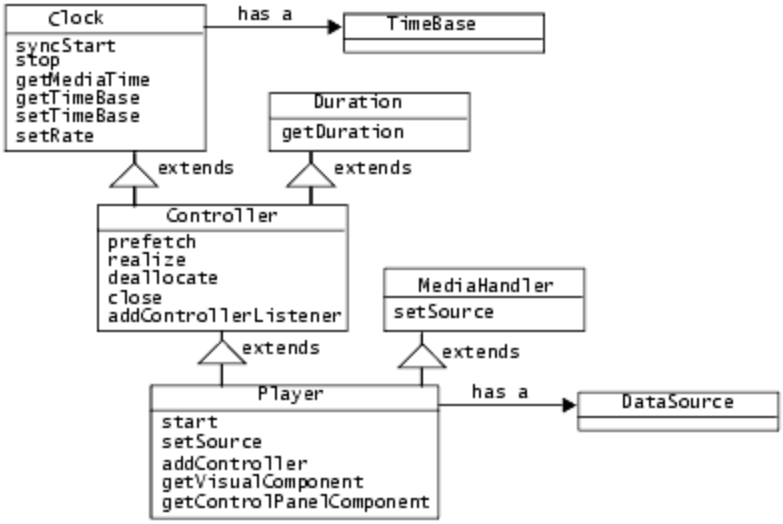


Figure 3.27: JMF players.

3.7.14.7.1 Player States

A Player can be in one of six states. The Clock interface defines the two primary states: *Stopped* and *Started*. To facilitate resource management, Controller breaks the *Stopped* state down into five standby states: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, and *Prefetched*.

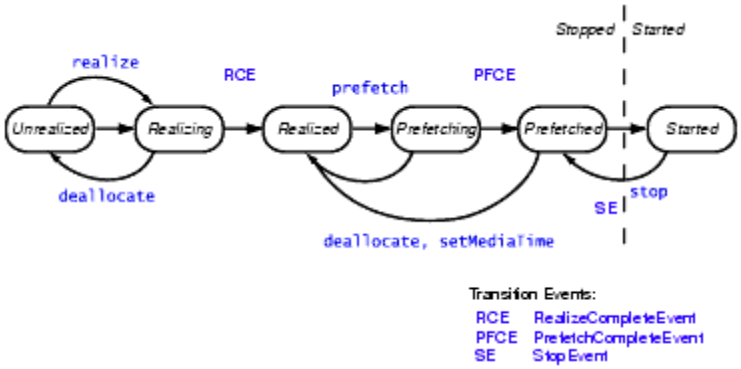


Figure 3.28: Player states.

In normal operation, a Player steps through each state until it reaches the *Started* state:

A Player in the *Unrealized* state has been instantiated, but does not yet know anything about its media. When a media Player is first created, it is *Unrealized*.

When realize is called, a Player moves from the *Unrealized* state into the *Realizing* state. A *Realizing* Player is in the process of determining its resource requirements. During realization, a Player acquires the resources that it only needs to acquire once. These might include rendering resources other than exclusive-use resources. (Exclusive-use resources are limited resources such as particular hardware devices that can only be used by one Player at a time; such resources are acquired during *Prefetching*.) A *Realizing* Player often downloads assets over the network.

When a Player finishes *Realizing*, it moves into the *Realized* state. A *Realized* Player knows what resources it needs and information about the type of media it is to present. Because a *Realized* Player knows how to render its data, it can provide visual components and controls. Its connections to other objects in the system are in place, but it does not own any resources that would prevent another Player from starting.

When prefetch is called, a Player moves from the *Realized* state into the *Prefetching* state. A *Prefetching* Player is preparing to present its media. During this phase, the Player preloads its media data, obtains exclusive-use resources, and does whatever else it needs to do to prepare itself to play. *Prefetching* might have to recur if a Player object's media presentation is repositioned, or if a change in the Player object's rate requires that additional buffers be acquired or alternate processing take place.

When a Player finishes *Prefetching*, it moves into the *Prefetched* state. A *Prefetched* Player is ready to be started.

Calling start puts a Player into the *Started* state. A *Started* Player object's time-base time and media time are mapped and its clock is running, though the Player might be waiting for a particular time to begin presenting its media data.

A Player posts TransitionEvents as it moves from one state to another. The ControllerListener interface provides a way for your program to determine what state a Player is in and to respond appropriately. For example, when your program calls an asynchronous method on a Player or Processor, it needs to listen for the appropriate event to determine when the operation is complete.

Using this event reporting mechanism, you can manage a Player object's start latency by controlling when it begins *Realizing* and *Prefetching*. It also enables you to determine whether or not the Player is in an appropriate state before calling

methods on the Player.

3.7.14.7.2 Presentation Controls

In addition to the standard presentation controls defined by `Controller`, a `Player` or `Processor` might also provide a way to adjust the playback volume. If so, you can retrieve its `GainControl` by calling `getGainControl`. A `GainControl` object posts a `GainChangeEvent` whenever the gain is modified. By implementing the `GainChangeListener` interface, you can respond to gain changes. For example, you might want to update a custom gain control `Component`.
Additional custom `Control` types might be supported by a particular `Player` or `Processor` implementation to provide other control behaviors and expose custom user interface components. You access these controls through the `getControls` method.
For example, the `CachingControl` interface extends `Control` to provide a mechanism for displaying a download progress bar. If a `Player` can report its download progress, it implements this interface. To find out if a `Player` supports `CachingControl`, you can call `getControl(CachingControl) or` use `getControls` to get a list of all the supported `Controls`.

Chapter IV

EVALUATION OF TRAINING

4.1 OUTPUT OF PLAYER

4.1.1 Load Song

This is the panel where user loads song to the playlist. Following files are selectable in this player:

- .mp3
- .mpg
- .mpeg

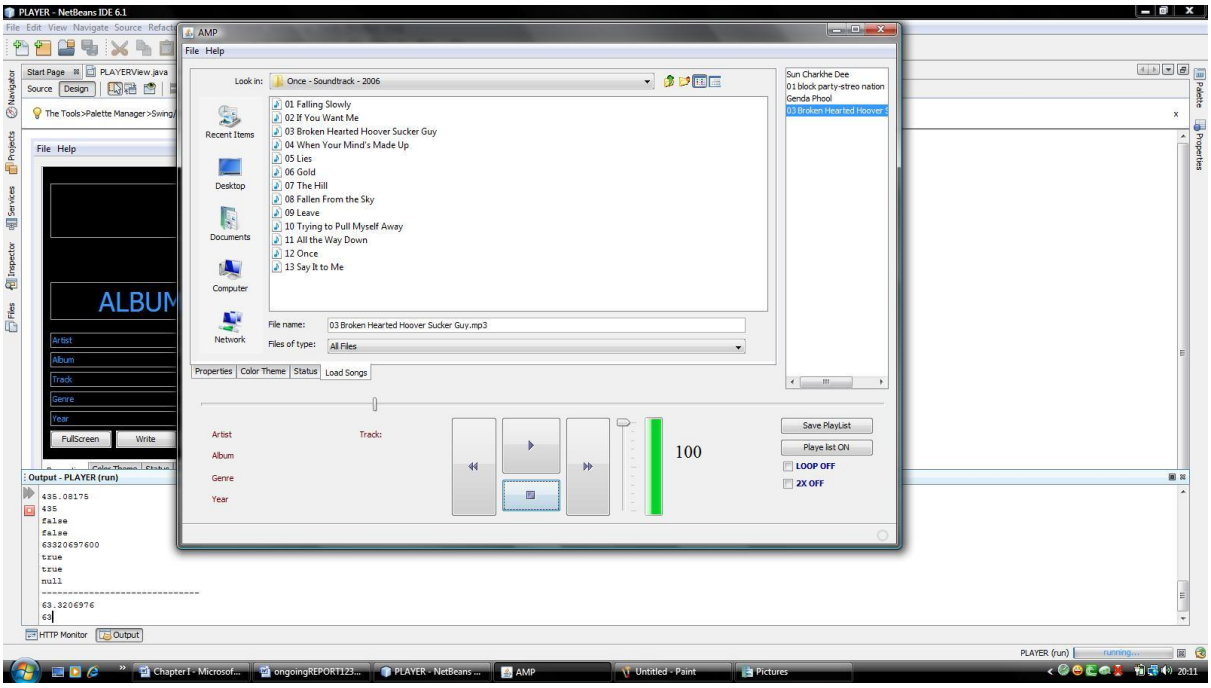


Fig. 4.1 Load Song

4.1.2 Song Properties

This panel displays the song file properties. This panel is divided into two parts LEFT and RIGHT. Left part contains the album properties which displays ARTIST, ALBUM, TRACK, GENRE and YEAR. To the right panel is the SOUND PROPERTIES which displays properties such as VERSION, MPEG LAYER, CHANNEL, COPYRIGHT, CRC and EMPHASIS values.

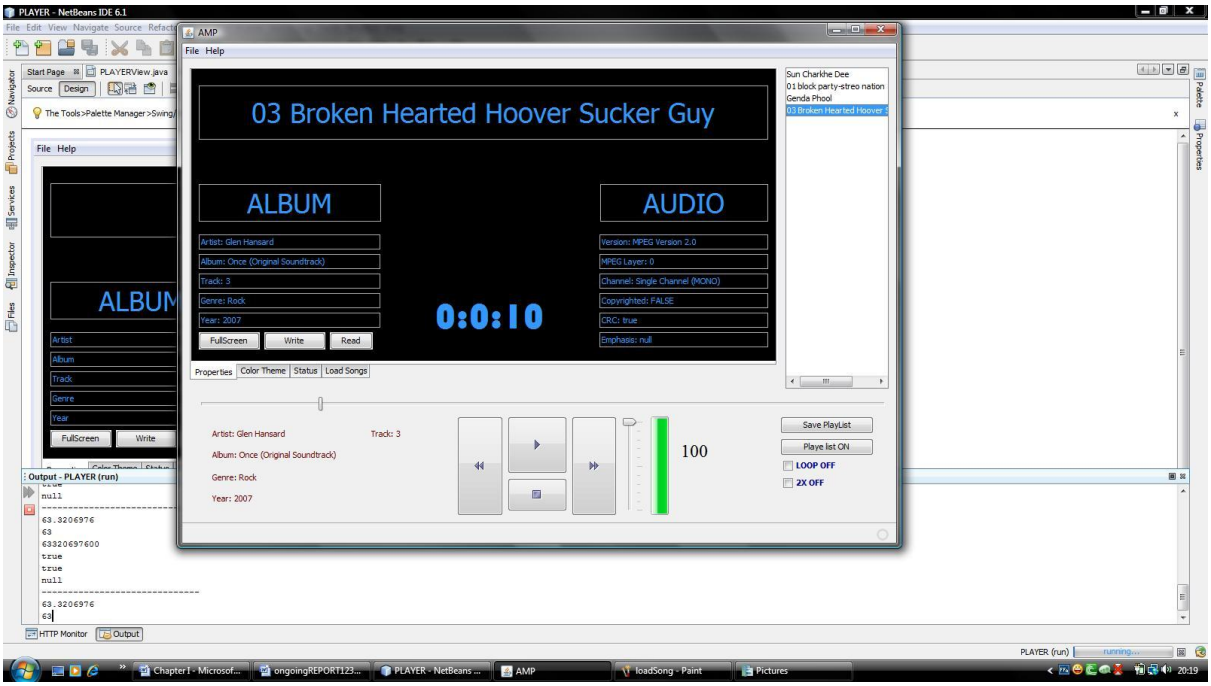


Fig. 4.2 Sound Properties

4.1.3 Status and Video Panel

This is the panel where video is loaded and status progress bar of the loaded Media File. If the media stream doesn't contain any video component; this panel remains blank.

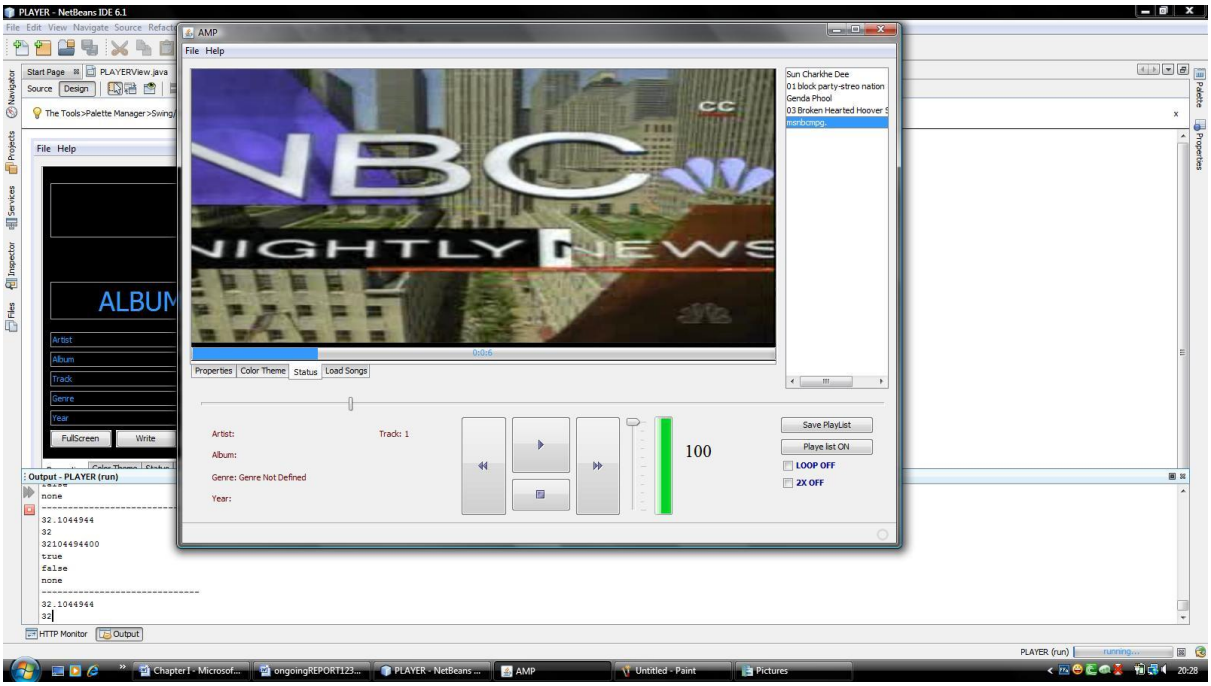


Fig.4.3 Video Panel

4.1.4 Save Playlist

Save playlist window opens on the player control panel and below the playlist panel. It includes one text field where the name of the playlist is typed. Two names are provided which controls saving and cancelation of the file.

Fig 4.4 Saving Song Playlist

4.1.5 Color Theme Selection

Here user creates color themes for the panel. Colors are generated by three color mixers RED Slider, GREEN Slider and BLUE Slider.

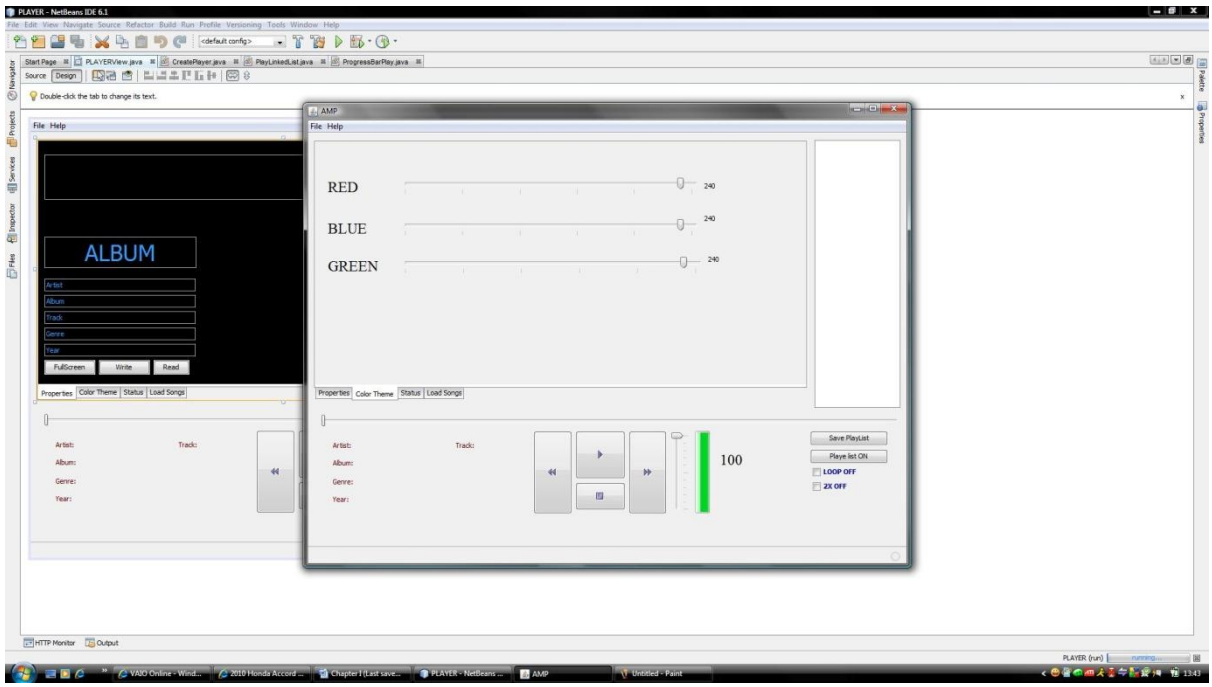


Fig. 4.5 Color Chooser

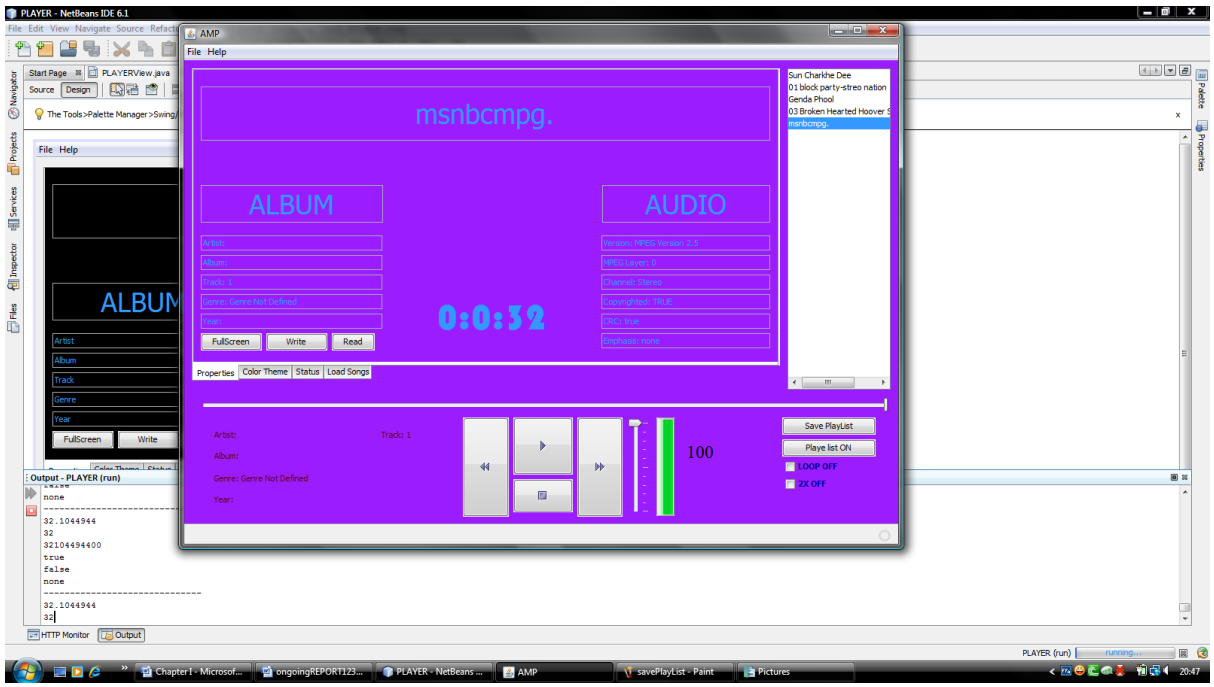


Fig 4.6 Choosing Color Theme

4.1.6 The About Box

This is the about box of the player displaying the version of the software and update website address where this player can be updated in future.

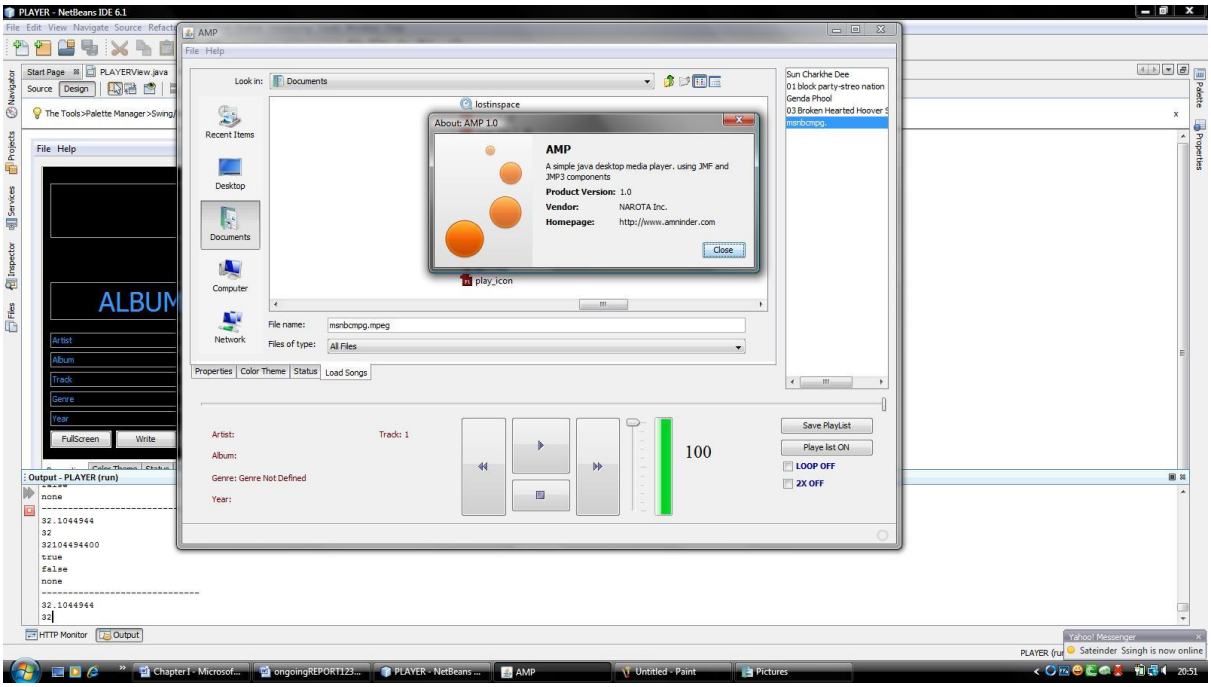


Fig 4.7 About Box

4.1.7 Full Screen

When user double clicks the video panel (if the media stream contains video component) the video panel gets transferred to full screen window.

4.8 Full Screen Window

Chapter V

Conclusion & Future Scope of the Training

5.1 Conclusion & Verifiable

This application is working properly under before mentioned system conditions. The application can be installed easily on all hardware. All it needs is proper buffer memory provided by JVM, proper JAVA2 environment. The playlist file it creates can also be read in other players.

This player uses JAVA MP3 API which enables the .mp3 support since this player is designed taking under consideration of KARAOKE users.

The setup installer is created in the Third Party software ADVANCED INSTALLER. Due to license agreement reasons it the setup is created under trial conditions which limits the buffer memory usage by player. So there is possibilities to in certain system where it might give problem in performance and work.

Due to the costlier license agreement of third party SPI's for EFFECTS, PLUGIN's, this player is unable to embed the GRAPHICAL EQUALIZERs and other GRAPHICAL EQUILIZERS in video panels in case video panel is not available.

This player is designed for reading MATA DATA under ID3v1 TAG of the media stream.

On playing a media stream through this player, FFCODEC is acquired which includes the DOLBY NR sound encoding for the output. The effect can be disabled from the respective panel.

5.2 Future Aspects

Since this player is designed under JMF environment so the most possible probability is upgrading for the support of the following :

- Able for playing popular streams such as .AVI, .WAV, .WMA
- Able to fetch META DATA under ID3v2 TAG of the media stream
- Able to record and transmit video streams by knowing video capture devices such as microphones or connected cameras.
- The JAVA ME version can also be created

APPENDIX

User : user is the client side user of the player.

System: System is the client side computer where the player is being used or installed.

JAVA 2: It refers to the SUN MICROSYSTEM JAVA version 2.

JRE 1.4: It refers to the JAVA RUNTIME ENVIRONMENT version 1.4

JRE 1.6: It refers to the JAVA RUNTIME ENVIRONMENT version 1.6

JMF: It refers to JAVA MEDIA FRAME WORK developed by IBM & JAVA

MP3 API: It refers to JMF's mp3 application programming interface. It is the add-on which supports mp3 files.

NETBEANS 6.1: Netbeans is the IDE for java programming. This IDE is used for developing this application.

ID3v1: id3lib is an open-source, cross-platform software development library for reading, writing, and manipulating ID3v1 tags.

META DATA: Meta data is the information about the media stream which is to be played in the player. It includes ARTIST, ALBUM, YEAR, GENRE and other information regarding sound like, MPEG version, CHANNEL, EMPHASIS, CRC etc.

References

Books:

- The Complete Reference JAVA 6 (Fifth Edition) by HERBERT SCHILDT
- The JAVA(TM) Developers Almanac 1.4 (Fourth Edition) by PATRIC CHAN

Websites:

- http://www.exampledepot.com
- http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/documentatido.html
- http://java.sun.com/docs/books/tutorial/uiswing/components/progress.html
- http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/apidocs/javax/media/control/MpegAudioControl.html
- http://java.sun.com/javase/technologies/desktop/media/jmf/mp3/download.html