

LEARNING JAVA

eBook

[**Table of Contents**](#_4d34og8)

[Chapter 01 – Introduction to Java 7](#_Toc518054797)

[What is Java? 7](#_Toc518054798)

[Why was Java Developed? 7](#_Toc518054799)

[Application Areas of Java 8](#_Toc518054800)

[History of Java 8](#_Toc518054801)

[Sun-Oracle Deal 9](#_Toc518054802)

[Java Versions 10](#_Toc518054803)

[Platform Independency in Java 10](#_Toc518054804)

[Java Platforms 12](#_Toc518054805)

[USP of Java: Java Features 13](#_Toc518054806)

[Difference between JDK, JRE and JVM 15](#_Toc518054807)

[JVM Architecture 16](#_Toc518054808)

[Installing Java on Windows 10 18](#_Toc518054809)

[Summary 26](#_Toc518054810)

[Chapter 02 – Creating First Java Program 27](#_Toc518054811)

[Understanding Text Editors to Write Programs 27](#_Toc518054812)

[Understanding Java Compiler 29](#_Toc518054813)

[Understanding Byte Code and Its Execution 30](#_Toc518054814)

[Summary 32](#_Toc518054815)

[Chapter 03 – Java Language Fundamentals 33](#_Toc518054816)

[Java Identifiers 33](#_Toc518054817)

[Java Keywords 34](#_Toc518054818)

[Java Variables 34](#_Toc518054819)

[Literals in Java 36](#_Toc518054820)

[Data Type 38](#_Toc518054821)

[Operators in Java 39](#_Toc518054822)

[Java Comments 44](#_Toc518054823)

[Control Statement 45](#_Toc518054824)

[Condition Statements 45](#_Toc518054825)

[Looping Statements 49](#_Toc518054826)

[Jump Statements 53](#_Toc518054827)

[Type Casting 57](#_Toc518054828)

[Summary 61](#_Toc518054829)

[Chapter 04 – Object Oriented Technique and Implementation 62](#_Toc518054830)

[Why Object Oriented Programming (OOP) 62](#_Toc518054831)

[OOP Concepts with Real Life Examples 62](#_Toc518054832)

[Object 62](#_Toc518054833)

[Syntax to Create an Object 63](#_Toc518054834)

[Class 63](#_Toc518054835)

[Syntax to Create a Java Class 63](#_Toc518054836)

[Declaration of Reference Variables 63](#_Toc518054837)

[Instantiation of Objects 64](#_Toc518054838)

[Constructors and Initializers 65](#_Toc518054839)

[Adding Attributes to Object 68](#_Toc518054840)

[Adding Behaviors (method) to Object 70](#_Toc518054841)

[*Static* Keyword and Its Usage 72](#_Toc518054842)

[*this* Keyword and Its Usage 72](#_Toc518054843)

[Inheritance 77](#_Toc518054844)

[Types of Inheritance 78](#_Toc518054845)

[Object class as root of Java class hierarchy 79](#_Toc518054846)

[Variable Hiding 79](#_Toc518054847)

[Method Hiding 80](#_Toc518054848)

[Method Overloading 80](#_Toc518054849)

[Method Overriding 83](#_Toc518054850)

[Super Keyword and Its Usage 84](#_Toc518054851)

[Using final to Prevent Overriding and Inheritance 86](#_Toc518054852)

[Up-casting and Down-casting 87](#_Toc518054853)

[Polymorphism 89](#_Toc518054854)

[Dynamic Binding / Dynamic Method Dispatch 90](#_Toc518054855)

[Implementing Runtime Polymorphism 91](#_Toc518054856)

[Abstraction 92](#_Toc518054857)

[abstract keyword (abstract classes and methods) 92](#_Toc518054858)

[Understanding Interfaces 95](#_Toc518054859)

[Encapsulation 98](#_Toc518054860)

[Reading Input from Keyboard 104](#_Toc518054861)

[Understanding Packages 107](#_Toc518054862)

[Access Modifiers 111](#_Toc518054863)

[Nested Classes 113](#_Toc518054864)

[Garbage Collection 116](#_Toc518054865)

[Summary 117](#_Toc518054866)

[Chapter 05: Arrays 118](#_Toc518054867)

[General Definition of Array 118](#_Toc518054868)

[Advantages of an Array 118](#_Toc518054869)

[Disadvantage of an Array 118](#_Toc518054870)

[Arrays in Java 118](#_Toc518054871)

[1-D Array 119](#_Toc518054872)

[Multidimensional Array 124](#_Toc518054873)

[Jagged Array 126](#_Toc518054874)

[Operations on Arrays 126](#_Toc518054875)

[Summary 129](#_Toc518054876)

[Chapter 06 – Command Line Arguments and Wrapper Classes 130](#_Toc518054877)

[Command Line Argument 130](#_Toc518054878)

[Wrapper Classes in Java 131](#_Toc518054879)

[Parsing of Numeric Strings 133](#_Toc518054880)

[String Representation of Primitives 134](#_Toc518054881)

[Summary 136](#_Toc518054882)

[Chapter 07 – Exception Handling 137](#_Toc518054883)

[Runtime Errors 137](#_Toc518054884)

[Types of Runtime Errors 137](#_Toc518054885)

[Types of Exception 139](#_Toc518054886)

[Exception Handler 140](#_Toc518054887)

[Java Exception Handling Keywords 140](#_Toc518054888)

[Try and Catch 140](#_Toc518054889)

[Multiple Catch Clauses 141](#_Toc518054890)

[Polymorphic Behavior of *catch()* Block 142](#_Toc518054891)

[Nested Try Statements 142](#_Toc518054892)

[Finally, throw and throws 143](#_Toc518054893)

[Creating Custom Exceptions 148](#_Toc518054894)

[Assertion 149](#_Toc518054895)

[Summary 150](#_Toc518054896)

[Chapter 08 – Working with Strings and Various Operations 151](#_Toc518054897)

[String: An Overview 151](#_Toc518054898)

[String Class 151](#_Toc518054899)

[Creating a String Object 151](#_Toc518054900)

[Operations on String 151](#_Toc518054901)

[Character Extraction in Java 157](#_Toc518054902)

[String Comparison 159](#_Toc518054903)

[StringBuffer Class and Methods 161](#_Toc518054904)

[Difference between String and StringBuffer 164](#_Toc518054905)

[String Builder Class 164](#_Toc518054906)

[Difference between StringBuffer and StringBuilder 168](#_Toc518054907)

[Summary 169](#_Toc518054908)

[Chapter 09 – Windows Programming 170](#_Toc518054909)

[Java AWT(Abstract Window Toolkit) 170](#_Toc518054910)

[Introduction to AWT Components 170](#_Toc518054911)

[Layout Manager 171](#_Toc518054912)

[Java Font and Color Classes 179](#_Toc518054913)

[Event Classes 182](#_Toc518054914)

[Event Listeners 183](#_Toc518054915)

[Java Adapter Classes 186](#_Toc518054916)

[Graphics Class 194](#_Toc518054917)

[Painting 197](#_Toc518054918)

[Problems with AWT 197](#_Toc518054919)

[Java Swing 198](#_Toc518054920)

[Controls 202](#_Toc518054921)

[Look and Feel of Swing Components 202](#_Toc518054922)

[MVC Architecture of Swing Components 218](#_Toc518054923)

[Working with Image 219](#_Toc518054924)

[Advanced Swing Components 221](#_Toc518054925)

[Summary 234](#_Toc518054926)

[Chapter 10 – Multithreaded Programming 235](#_Toc518054927)

[Multitasking 235](#_Toc518054928)

[Multiprocessing vs. Multithreading 235](#_Toc518054929)

[Main Thread (Default Java Thread) 235](#_Toc518054930)

[Creating Child Thread 236](#_Toc518054931)

[Context Switching 240](#_Toc518054932)

[Thread States 240](#_Toc518054933)

[Commonly Used Methods 242](#_Toc518054934)

[ThreadGroup in Java 243](#_Toc518054935)

[Important methods of ThreadGroup class 243](#_Toc518054936)

[Thread Synchronization: Methods and Blocks 256](#_Toc518054937)

[ReentrantLock 259](#_Toc518054938)

[Inter Thread Communication 262](#_Toc518054939)

[Daemon thread 264](#_Toc518054940)

[Deadlock 266](#_Toc518054941)

[Java Shutdown Hook 271](#_Toc518054942)

[Summary 274](#_Toc518054943)

[Chapter 11 – I/O Streams 275](#_Toc518054944)

[What is I/O 275](#_Toc518054945)

[Why Need Streams 275](#_Toc518054946)

[Byte Stream Classes 275](#_Toc518054947)

[Character Stream Classes 277](#_Toc518054948)

[Reader/Write Operations with File 279](#_Toc518054949)

[Scanner Class in Java 284](#_Toc518054950)

[Object Serialization and Deserialization 286](#_Toc518054951)

[*transient* Keyword 290](#_Toc518054952)

[Externalization 291](#_Toc518054953)

[File Class and Its Methods 292](#_Toc518054954)

[Summary 300](#_Toc518054955)

[Chapter 12 – Socket Programming 301](#_Toc518054956)

[Fundamentals of a Network 301](#_Toc518054957)

[Java Socket Programming 301](#_Toc518054958)

[Socket and ServerSocket Classes 302](#_Toc518054959)

[Java InetAddress class 304](#_Toc518054960)

[Java DatagramSocket and DatagramPacket Classes 305](#_Toc518054961)

[Java URL Class 307](#_Toc518054962)

[Java URLConnection class 308](#_Toc518054963)

[Java HttpURLConnection Class 309](#_Toc518054964)

[Summary 311](#_Toc518054965)

[Chapter 13 – Reflection 312](#_Toc518054966)

[Understanding the Need of Reflection 312](#_Toc518054967)

[Getting Information about Class’s Modifiers, Fields, Methods, Constructors and Super Classes 314](#_Toc518054968)

[Finding out Constant and Method Declaration Belong to an Interface 315](#_Toc518054969)

[Creating an instance of the class whose name is not known until runtime 318](#_Toc518054970)

[Getting and setting values of an object’s field if field name is unknown until runtime 319](#_Toc518054971)

[Invoking a Method on an Object If the Method is Unknown until Runtime 320](#_Toc518054972)

[Invoking Private Methods 321](#_Toc518054973)

[Summary 323](#_Toc518054974)

[Chapter 14 – Additional Concepts 324](#_Toc518054975)

[var-args (Variable Arguments) 324](#_Toc518054976)

[Static Import 325](#_Toc518054977)

[For-each loop 325](#_Toc518054978)

[Enum (Enumerate) 326](#_Toc518054979)

[printf( ) Method 327](#_Toc518054980)

[Static, Default, and Private Methods 328](#_Toc518054981)

[Date-Time API - Time Package to Work with Local Date & Time 330](#_Toc518054982)

[Lambda Expressions 333](#_Toc518054983)

[HTTPClient API 336](#_Toc518054984)

[Java Generics 338](#_Toc518054985)

[Annotations in Java 339](#_Toc518054986)

[VAR 341](#_Toc518054987)

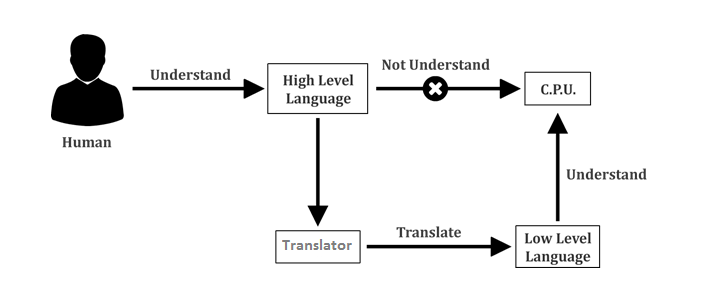
[Java Module 342](#_Toc518054988)

[Summary 343](#_Toc518054989)

# Chapter 01 – Introduction to Java

## What is Java?

Java is a high-level general-purpose, object-oriented,class based and concurrent programming language developed by a team of sun microsystems headed by James Gosling in 1995.



As Java is a general-purpose programming language, it covers wide range in computing field such as Desktop GUI Applications, Mobile Applications, Embedded Systems, Web Applications, Enterprise Applications, etc.Java is open-source to access and can run on all platforms.

## Why was Java Developed?

Java was developed by Sun Microsystems to reduce the implementation dependencies. In c/c++ languages programmers need to design the code based on targeted hardware and operating system and if hardware or os is changed, programmer need to modify the code and also recompile the code.

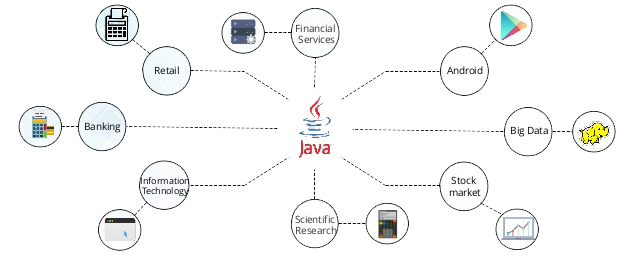
In early 90s when the Internet was new, the webpages designed to access the web were simple and static.

The trend of the Internet started that time, but developers were looking forward to exploring ever more. Eventually, the disability of high-level programming languages of that time restricted them to do so. For different programming language, if you want to run an html page then you need to send your source code to the compiler as well. Developers did not like it because the source code is their heart.

Therefore, the developers decided to design a new language that allowed them to reliably perform advanced functions on the Internet, without sharing their source code. This language was Java.

## Application Areas of Java

Java is extremely popular and has been dominating the software industry from the early 2000’s till today. There are countless places where Java is being used in the real world, starting from e-commerce websites to android apps, from games like *Minecraft* to desktop applications (such as, *Eclipse, Netbeans and IntelliJ*), from scientific application to financial applications, such as, electronic trading systems) and from an open source library to *J2ME* apps.



Java has been used in several domains. Some of them are listed below:

* **Banking:** Banks use Java-based systems to deal with transaction management.
* **Retail:** Billing applications you see in a store or restaurant are written in Java.
* **Information technology (IT):** Java is designed to resolve implementation dependencies.
* **Android:** Android applications are written in Java and use Java API.
* **Financial services:** In finance sectors, Java is used in server-side applications.
* **Stock market:** Java helps to write algorithms for investment status analysis.
* **Big data:** Hadoop MapReduce framework is written using Java.
* **Scientific and research community:** The scientific applicationswritten in Java can deal with big amount of scientific data and statistics.

## History of Java

Java was initially designed for interactive television, but this technology was far too advanced for the digital cable television industry at the time.

The history of Java starts from the *Green* Team. In December of 1990, a team code-named *Green* was formed. The purpose of the team was to create a simple device that could control a variety of electronics products. The team consisted of two programmers; Patrick Naughton and James Gosling, who were accompanied by an engineer, Mike Sheridan. While working on the device, James Gosling realised that what they actually needed was a new programming language. After several days of brainstorming, they came up with an idea of Java. Therefore Gosling is also known as the father of the Java programming language.

So, in June 1991, Gosling and Naughton started their first Java project, called *Oak* with the focus on reliability. This is so because the other programming languages of that time emphasised more on speed, rather than on reliability. In consumer electronics, reliability is more important than speed.   
  
Now because Gosling's goals were to implement a virtual machine and a language, *Oak* had a familiar C-like notation but with better consistency and simplicity.  
  
Although Java 1.0a was available for download in 1994, the first public implementation of Java took place with the version, Java 1.0a2 on 23rd of May 1995. Java 1.0a2 worked with the HotJava browser. It made the promise of "Write Once, Run Anywhere", with free runtimes on popular platforms. It was fairly secure with configurable features. This allowed the users to limit the access to their files and network. Soon the major Web browsers incorporated it into their standard configurations. New versions for large and small platforms (J2EE and J2ME) were designed soon with the advent of Java 2.

Since J2SE 1.4 in 2002, the Java Community Process (JCP) has been governing the overall evolution of Java. The JCP uses Java Specification Requests to specify any additions or changes to the platform.

In 2004, the format of the version name changed. Instead of being J2SE 1.5, it was known as J2SE 5.0. The name format changed once again in 2006 with the launch of Java SE 6. This format is still in use today.

JDK 9 was launched in July 2017 with promise of design and implementation of a standard module system to the Java SE platform. Recently JDK 10 was released on March 20, 2018 with key improvements like local variable types as well as enhancements for compilation & garbage collection.

## Sun-Oracle Deal

On April 20, 2009, Oracle signed a deal with Sun Microsystems to buy their company for $7.4 billion, plunging the enterprise software vendor into the hardware market. Oracle had to face a stiff competition against IBM, Hewlett-Packard Co, Dell Inc. and new entrant Cisco Systems Inc. to crack the deal with Sun Microsystems.

In this deal, Oracle agreed to pay $9.50 per share in cash for Sun or $5.6 billion net of Sun's cash and debt. This move was followed by Oracle's purchases of other companies including PeopleSoft, Siebel and BEA Systems.

*“Java and Solaris were the two main reasons Oracle purchased Sun”*

*- Larry Ellison, CEO Oracle*

Oracle's move on Sun was somewhat motivated by a desire to snap up two pieces of software – the programming language Java and Sun's Solaris operating system. Both are widely used throughout the world. This transaction ended 27 years of independence for Sun, which was founded by four graduate students in California. Sun was originally named to stand for Stanford University Network.

Sun, a pioneer in network computing, was valued at more than $200bn during the dotcom boom of the late 90s. But orders for its servers and storage computers had been slowing and it made losses in three of the past four quarters.

Oracle found this merger with Sun as a logical fit because Sun's Java and Solaris software were already used in Oracle’s own fleet of databases and middleware software.

## Java Versions

From the release of it first version in 1996 to the latest version 10.0 released in March 2018, the Java platform has been actively being developed for almost 20 years. Many changes and improvements have been made to the technology over the years. The following table summarizes all the versions of Java SE from its early days to the latest.

|  |  |  |
| --- | --- | --- |
| Version Name | Code Name | Release Date |
| *JDK 1.0* | *Oak* | January 1996 |
| *JDK 1.1* | *(none)* | February 1997 |
| *J2SE 1.2* | *Playground* | December 1998 |
| *J2SE 1.3* | *Kestrel* | May 2000 |
| *J2SE 1.4* | *Merlin* | February 2002 |
| *J2SE 5.0* | *Tiger* | September 2004 |
| *Java SE 6* | *Mustang* | December 2006 |
| *Java SE 7* | *Dolphin* | July 2011 |
| *Java SE 8* | *Spider* | March 2014 |
| *Java SE 9* |  | September, 21st 2017 |
| *Java SE 10* |  | March, 20th 2018 |

From the above table, we can see that the naming and the version number have been changing over times:

* Versions 1.0 and 1.1 are named as JDK (Java Development Kit).
* From versions 1.2 to 1.4, the platform is named as J2SE (Java 2 Standard Edition).
* From versions 1.5, Sun introduces internal and external versions. Internal version is continuous from previous ones (1.5 after 1.4), but the external version has a big jump (5.0 for 1.5). This could create confusion for you, so keep in mind that version 1.5 and version 5.0 are just two different version names for same thing.

Major versions were released in every two years, however the Java SE 7 took 5 years and Java SE 8 took 3 years to be available to public afterward.

Since Java SE 10, now new versions of Java will be released every six months.

## Platform Independency in Java

**What is a Platform?**

A platform can be defined as hardware, software as well as the sum of hardware and software, which provides the environment to run the program.

A programming language or technology is said to be platform-independent if and only if it can run on all available platforms, with respect to its development and compilation.

In Java, platform independency means:

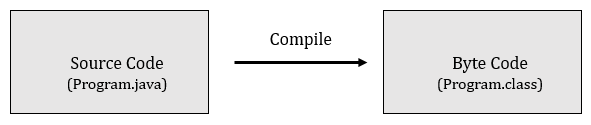
* **Source code is platform Independent (WORA):**

We need not to change the source code according to a specific platform

* **Compiled code is platform Independent (CORA):**

We need not to recompile the program for specific platform

**Source Code to Byte Code**



(Platform Independent)

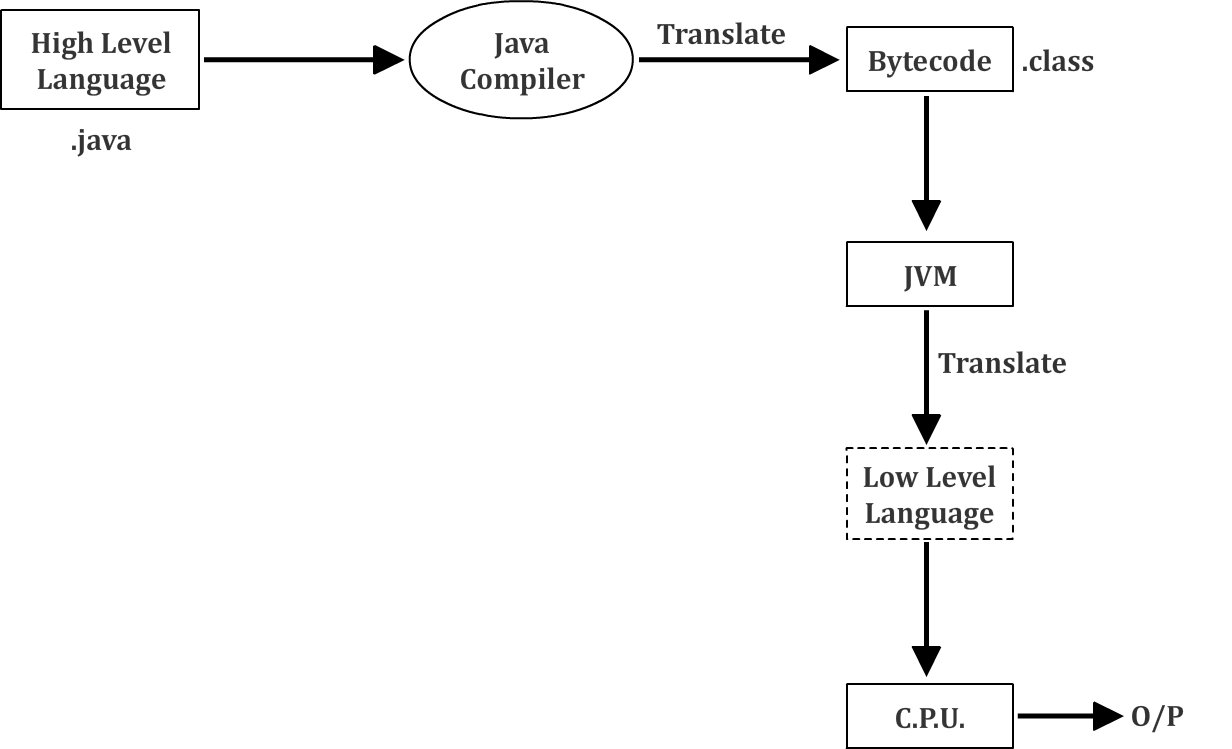
In Java, the source code is compiled and converted into the byte code. This byte code is not the machine code, so the specific platform cannot understand it. Therefore, byte code is further converted into machine code.

**Byte Code to Machine Code**

In Java, the byte code is not the required machine code. Therefore, to get the required machine code we have JVM in Java. JVM interprets the byte code and converts it into the machine specific code. Without the JVM,a machine cannot understand the byte code, hence the program cannot be executed.

You can make the byte codeunderstandable to any platform using the Java Virtual Machine. That is what makes the byte code, platform-independent. But on the other hand,*the JVM is different for each platform which is why it is known as platform-dependent*. Java is a platform-independent language because it does not depend on any type of platform.

The diagram of how a high-level java code is compiled to machine language is shown below:



## Java Platforms

The platforms of the Java programming language are:

* Java Platform, Standard Edition (Java SE) or JDK
* Java Platform, Enterprise Edition (Java EE)
* Java Platform, Micro Edition (Java ME)
* JavaFX

**Java SE**

Java SE API provides the basic functionality of the Java programming language. Java SE defines everything from the basic types and objects of Java to high-level classes, which are used for networking, database access, security and XML parsing.

**Java EE**

The Java EE platform is built on the top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable and secure network applications**.**

**Java ME**

The Java ME platform provides an API for running Java applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development.

**JavaFX**

JavaFX is a platform for developing rich internet applications using a lightweight user-interface API. JavaFX applications use hardware-accelerated graphics and media engines. JavaFX takes their advantage of higher-performance clients, modern look-and-feel and high-level APIs for connecting to networked data sources.

## USP of Java: Java Features

* **Java is FREE:** Cost acts as a critical deciding factor when a programmer plans to learn a programming language, or an organization plans to apply a technology. Dealing with the rising cost shall indeed become overwhelming, until Java comes to your rescue.

Since Java comes as a free resource you don’t have to worry about the cost that you may have to incur, otherwise, to create a Java application. It is because of this fact that Java is one of the most popular languages among individual programmers and various organizations.

* **It’s easy to learn:** Java is considered as one of the best programming languages because it is easy to learn and use. A programmer can easily write a program in Java, once he learns to install JDK, set up PATH, and execute CLASSPATH. It comprises English-like syntaxes that include minimal magic characters like generics angle brackets which make a Java program easy to read and grasp.
* **It is Object-oriented:** Java is an Object-oriented Programming (OOP) language, which is another reason that makes it a popular choice among the programmers around the world. Developing an OOP application is extremely easy while keeping the system flexible, modular, and extensible.  
    
  Once you have learnt the main OOP concepts such as, polymorphism, abstraction, encapsulation, and inheritance, you can easily apply those with Java. If you would review the Java library, you would see there are many resources that describe the best practices and design patterns used in Java. Java is the only language that applies the OOP concepts on a 100% basis.

Besides this, Java also supports the use of object-oriented and SOLID design principles in conjunction to ensure that your object dependency is managed greatly using Dependency Injection principle. Such a version of Java is available in the form of open source platforms like Spring.

* **It is Platform-independent**

Back in 90s when the platform independence first came in the picture, Java gained popularity with its tagline **"write once run anywhere" or** WORA. Programmers’ consistent interest in using Java, triggered many developments improving the language in each phase.

Most Java applications are developed in Windows environment and run on UNIX platform. This again makes Java one of the top programming languages.

* **It has Rich API**

Another reason for the massive success of the Java programming language is its rich API and most significantly it is highly visible since it comes with installation of Java.  
  
Java can be used to design APIs for XML parsing, I/O, networking, database connection, utilities. Rest are covered by open source libraries like Google Guava, Gson, Apache POI, Apache Commons, Jackson, and others

* **Powerful Development Tools e.g. Eclipse, Net Beans**

Java is supported by Eclipse and Netbeans, which have played a big role in making it one of the best programming languages available today.

While these tools assist in code completion, they offer great debugging capability that is critical for the real-world development. This capability is powered by the Integrated Development Environment (IDE). With IDE, Java development becomes much easier, quicker, and smooth. It makes it very easy to search, refactor and read Java code. Coding in IDE is an especially fun, if you code in Notepad or DOS Editor.

Besides IDE, Java platform also support other tools such as ANT and Maven (used to create Java applications), and JConsole, decompilers, Visual VM (used to monitor Heap usage etc.)

* **Great Collection of Open Source Libraries**

With the abundance of open source libraries, a programmer can use Java everywhere. Google, Apache, and other organizations have contributed numerous great libraries that make Java development faster, easier, and economical.  
  
There are frameworks like Spring, Maven, and Struts ensure that Java developers follow great practices of software craftsmanship. They do so by promoting the usage of design patterns.

* **Wonderful Community Support**

A strong and prosperous community is the major strength of Java programming language and its platform. A language would never survive, if there is no community to support and share their knowledge related to the language, no matter, how good it is.  
  
Many active forums including *StackOverflow*, open source organizations, and numerous Java user groups form the part of community that supports Java. They are there to support beginners, intermediate, and even expert Java programmers. In fact, many programmers who have been using open source, also contribute to the community as commiters or testers.

* **Excellent documentation support - Javadocs**

Javadoc is a big part of documentation that describes the Java API in the simplest way possible. It acts as an exceptional point of reference for programmers who use Java. Java would never be this popular without the support of the Javadoc documentation.

* **Java is everywhere**

Yes, Java is everywhere, it is on the desktop, on mobile, on the card, almost everywhere and so are the Java programmers. Java programmers always surpass the experts using other programming languages in number.

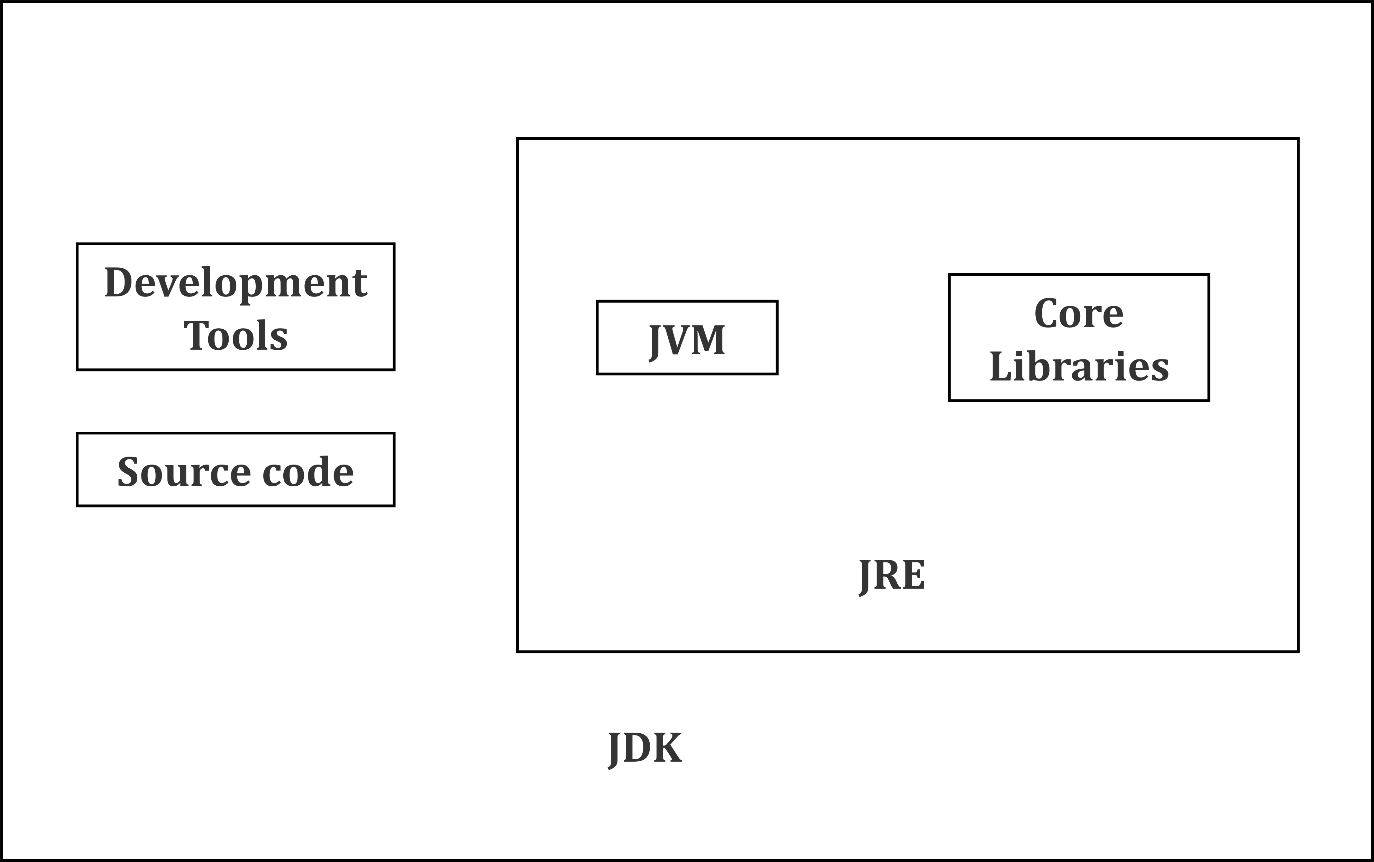
That said, the field of programming languages is vast. There have been languages such as UNIX and C that are still used. Much like Java, they are also most likely to survive another 20-30 years.

On the contrary, the functional programming languages such as, Scala, and JVM languages, have a long way to go to match the level of resources, community, and popularity that Java enjoys.

With Java, you get many career options to choose from. Regardless of what you choose to do (develop games, or mobile, Web, or desktop applications), Java is capable to support you in all your endeavours.

## Difference between JDK, JRE and JVM

JDK, JRE and JVM are commonly used terms in Java and we must understand these terms clearly. The diagram below shows relation between JDK, JRE and JVM.



**JVM (Java Virtual Machine)**

JVM stands for Java Virtual Machine. JVM is an engine that provides runtime environment to drive the Java code or applications. It converts Java byte code into machines language. JVM is also a part of JRE (Java Runtime Environment).

* In other programming languages, the compiler produces machine code for a particular system. However, Java compiler produces code for a virtual machine known as Java Virtual Machine.
* Java code is first compiled into byte code and then the byte code gets interpreted on different machines.
* Byte code is an intermediary language between the host system and the Java source.
* JVM is also responsible for allocating memory space.

**JRE (Java Runtime Environment)**

JRE refers to a runtime environment in which Java byte code can be executed. It implements the JVM and provides all the class libraries and other support files that are used by JVM at runtime. So, JRE is a software package that contains what is required to run a Java program. Essentially, it’s an implementation of the JVM which physically exists.

**JDK (Java Development Kit)**

The JDK is provided free of cost. The JDK completely includes JRE which contains tools for Java programmers. Besides JRE, it includes an interpreter/loader, a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development. In short, it contains JRE + Java development tools.

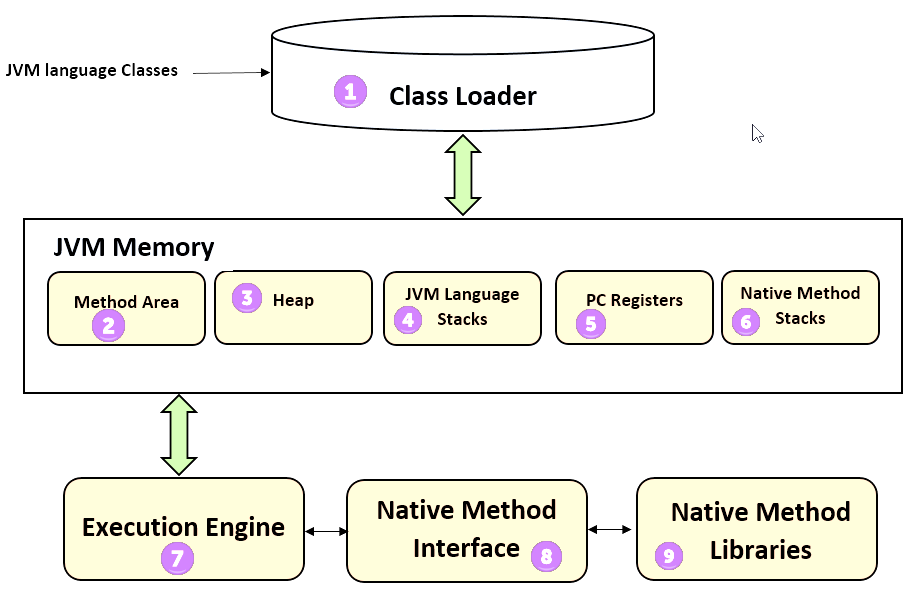
**Commonly Used Development Tools**

Following are some of the commonly used java development tools:

* **javac**: It is java compiler used to compile java files.
* **Java**: It is JVM and used to run the byte code.
* **javap**: It is Java profiler and disassembles one or more class files.
* **javadoc**: It is used for generating API documentation in HTML format from Java source code.
* **jar**: It is a package file format typically used to aggregate many Java class files and associated metadata and resources.
* **jdb**: It is a simple command-line debugger for Java classes.

## JVM Architecture

Let's understand the Architecture of JVM. JVM contains classloader, memory area and execution engine among other elements.



1. **ClassLoader:** The class loader is a subsystem used for loading class files. It performs three major functions, loading, linking and initialization.
2. **Method Area:** The method area of JVM stores class structures like metadata, the constant runtime pool and the code for all the methods.
3. **Heap:** Heap is a memory where all the objects, their related instance variables, and arrays are stored. This is common storage area and shared across multiple threads.
4. **JVM language Stacks:** Java language stacks store local variablesand their partial results. Each thread has its own JVM stack,which is created simultaneously,as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when the method invocation process is complete.
5. **PC (Program Counter) Registers:** In Java, each thread has its own PC register. A PC register stores the address of the Java virtual machine instruction which is being executed currently.
6. **Native Method Stacks**: Native method stacks hold the instruction of native code that depends on the native library. Native methods are written in a language other than Java and Native Method Stacks help executing them.
7. **Execution Engine**: It is a type of software used to test hardware, software, or complete systems. However, note that the test Execution Engine never carries any information about the tested product.
8. **Native Method Interface**: The native method interface is a programming framework that allows the Java code running in JVM to be called by libraries and native applications.
9. **Native Method Libraries:** Native Method Libraries are a collection of the native libraries written in C or C++ libraries that are used by the execution engine

**Execution Engine**

The bytecode which is assigned to the runtime data areas in JVM by the class loader is executed by the execution engine. The execution engine reads the Java bytecode in the unit of instruction. It is similar to a CPU executing the machine commands one by one. Each command of the bytecode consists of a 1-byte OpCode and additional Operand. The Execution Engine gets one OpCode and executes task with the Operand, and then executes the next OpCode.

Java bytecode is written in an English-like language that a human can understand, rather than in machine language. Therefore, the execution engine must convert the bytecode into the language that can be executed by the machine in JVM.

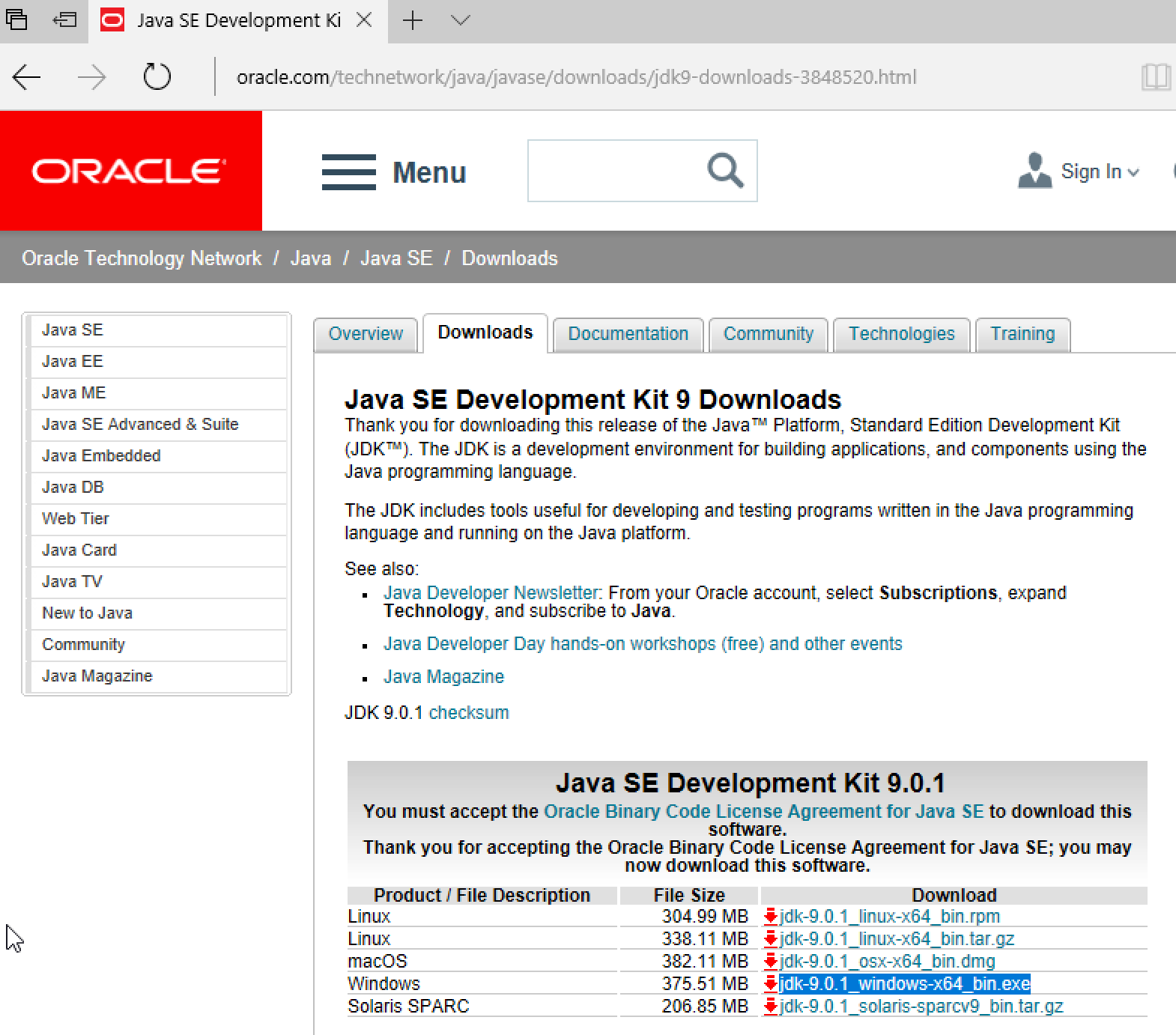
1. **Interpreter:**  It reads the bytecode, interprets and executes it one by one. The interpreter interprets the bytecode faster but executes slowly. This is so because when a method is called multiple times, the interpreter will have to interpret it every time (again and again)
2. **JIT Compiler**: The JIT compiler has been introduced to neutralize the disadvantages of the interpreter. The execution engine runs as an interpreter first, and at the appropriate time, the JIT compiler compiles the entire bytecode to convert it into native code. After that the execution engine does not interpret the method, but directly executes using native code. The execution in native code is much faster than interpreting of instructions one by one. The compiled code can then be executed quickly since the native code is stored in the cache.
   * **Intermediate Code Generator**: It produces the intermediate code.
   * **Code Optimizer**: It is responsible to optimize the intermediate code, generated by the intermediate code generator.
   * **Target Code Generator**: It is responsible for generating the machine code/native code.
   * **Profiler**: Profiler is a special component that is responsible for finding the hotspots that are used to identify if the method is called multiple times or not.
3. **Garbage Collector**: A part of the execution engine, the garbage collector collects/removes the unreferenced objects. The garbage collection can be triggered by calling *“System.gc()”*, but the execution is not guaranteed. Garbage Collector of JVM collects only those objects that are created by using the*new* keyword. So, if you have created any object without *new*, you can use *finalize* method to perform cleanup.

## Installing Java on Windows 10

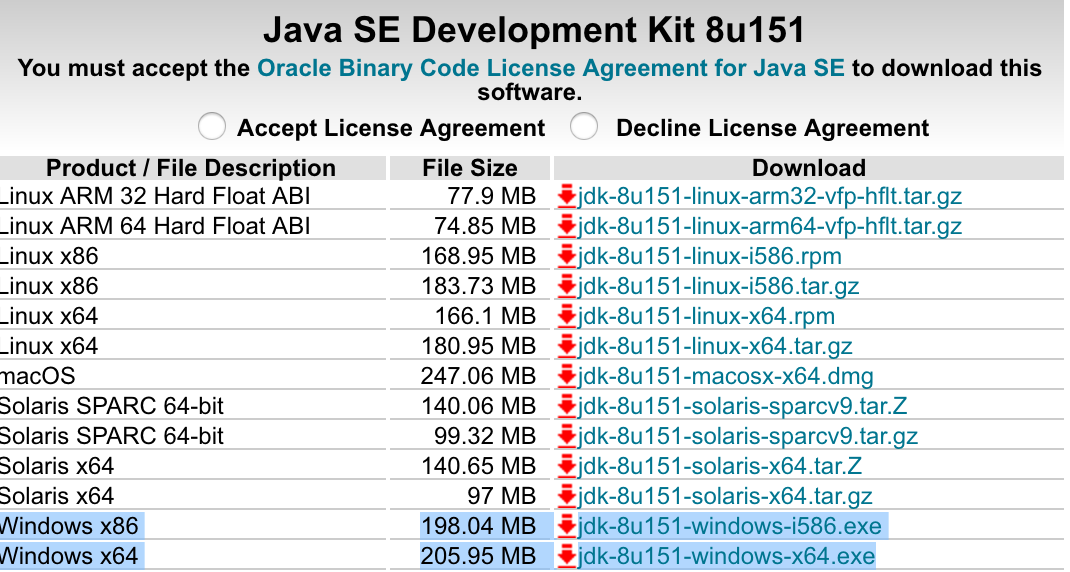
Setting up Java on Windows 10 is very easy and can be done byperforming the steps given below:

1. **Download Java for Windows 10**

The first step of Java setup process involves downloading of Java installation files from Oracle’s website. To install latest version of Java, go to the download link and accept the license agreement. Next, click the download link for Windows (as shown in below image) and save the file.

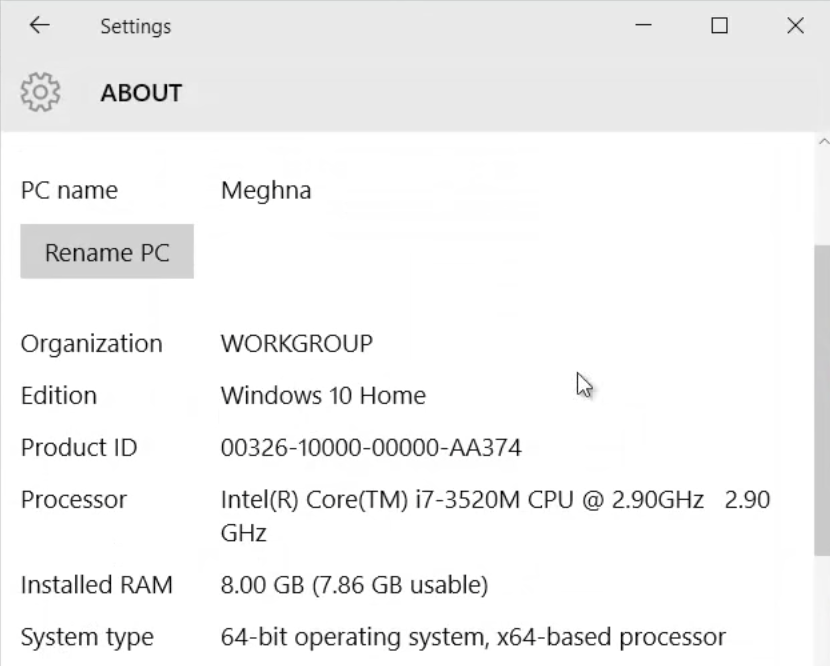


Note: In case, you want to install Java 8 or earlier versions, you need to know whether the Windows 10 in your computer is 32-bit or 64-bit. It is important to note because there two different installers are available for Windows.



If you are using 32-bit Windows 10, then download Windows x86 exe file, otherwise download Windows x64 exe file.

If you are not sure of the version of your Windows, you can follow the given path to check it: Settings > System Settings > About.

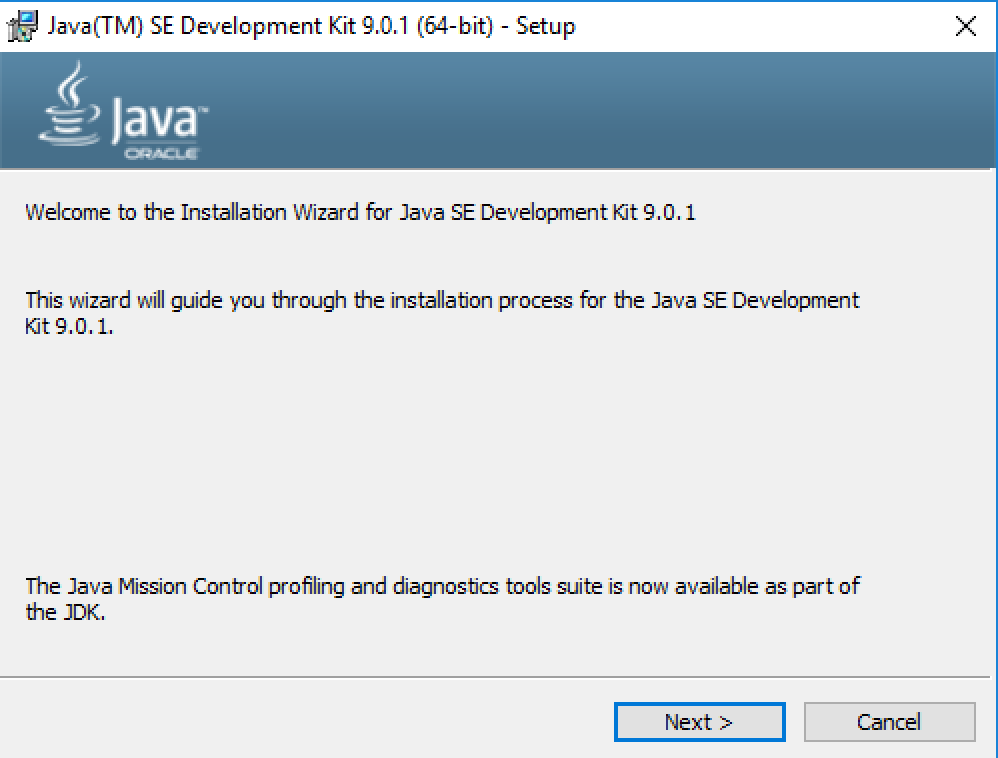


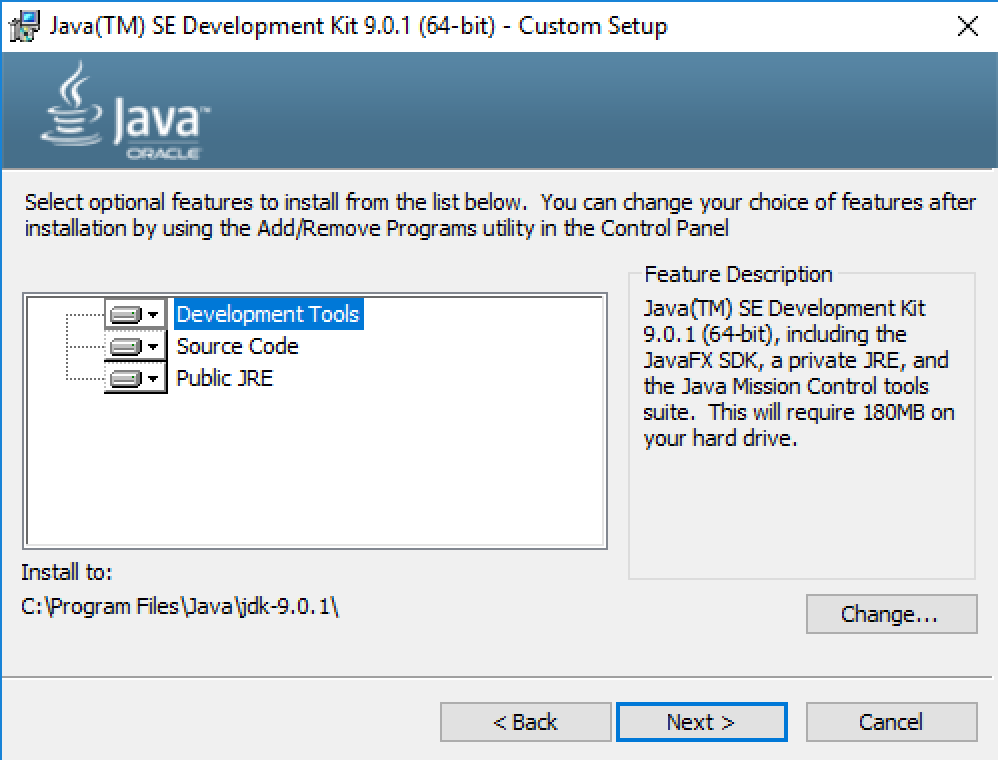
1. **Java Windows 10 Installation Steps**

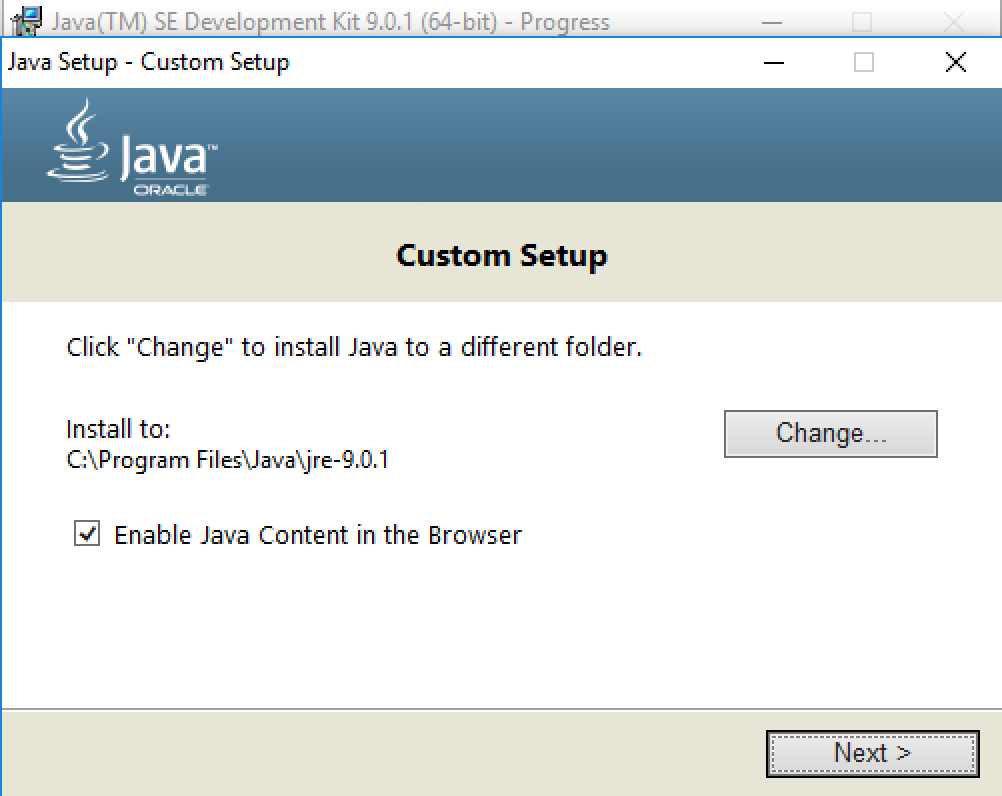
Once all the relevant files are downloaded, the next step is to run the downloaded Java installer exe file. To run the file, double click it and follow the steps displayed on screen.

Generally, you don’t need to check anything and go with default selection. They may slightly differ based on Java version installation but mostly it’s similar to installation screens below.

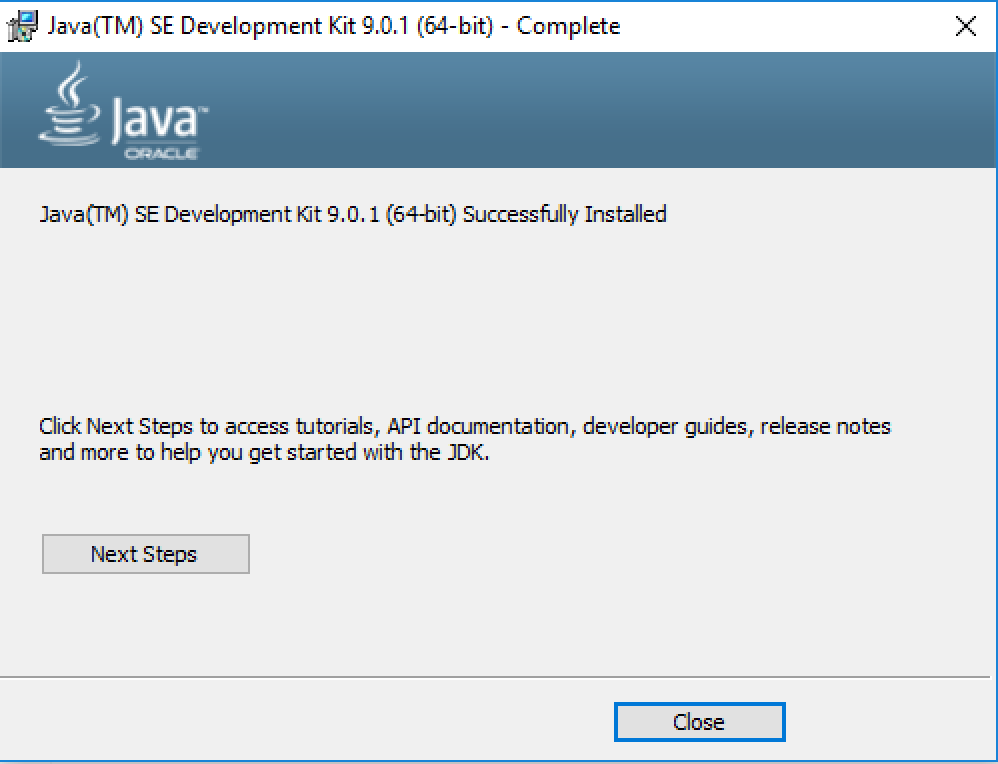






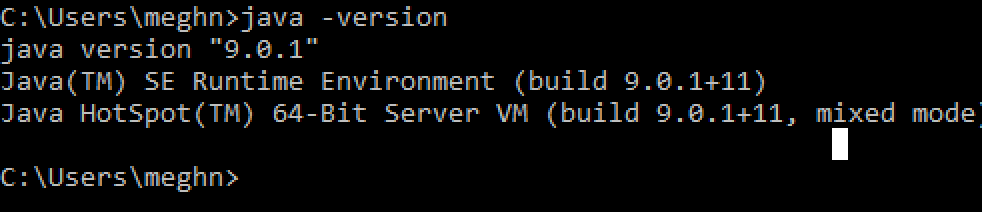






1. **Java on Windows 10 Version Check**

Finally, you need to check if Java has been successfully installed in your system. You can do that by typing java –versioncommand in the command prompt, as shown in below image.



Note: **In the older versions of Java and Windows operating systems**, you need to setup two environment variables to get it working.

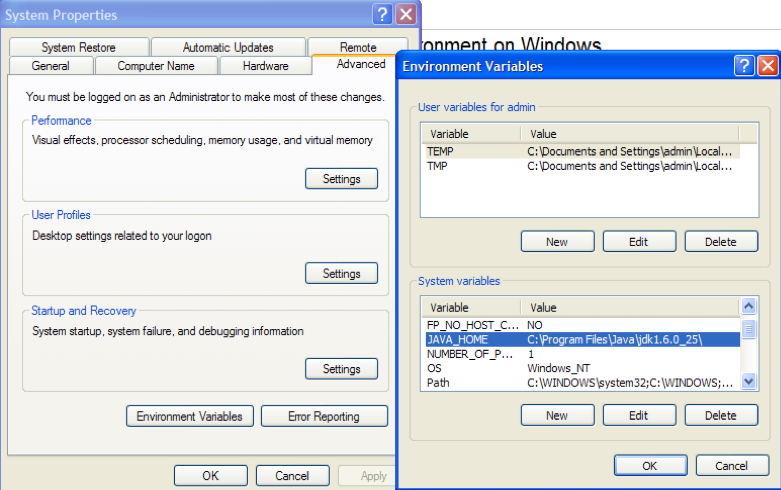
To setup the environment variables, go to:

System Properties (Right Click on My Computer and select Properties) > Advanced > Environment Variables.

In the popup window, *System variables* section, click on *New* button and add a variable with following details:

Variable: JAVA\_HOME

Value: *C:\Program Files\Java\jdk1.6.0\_25*



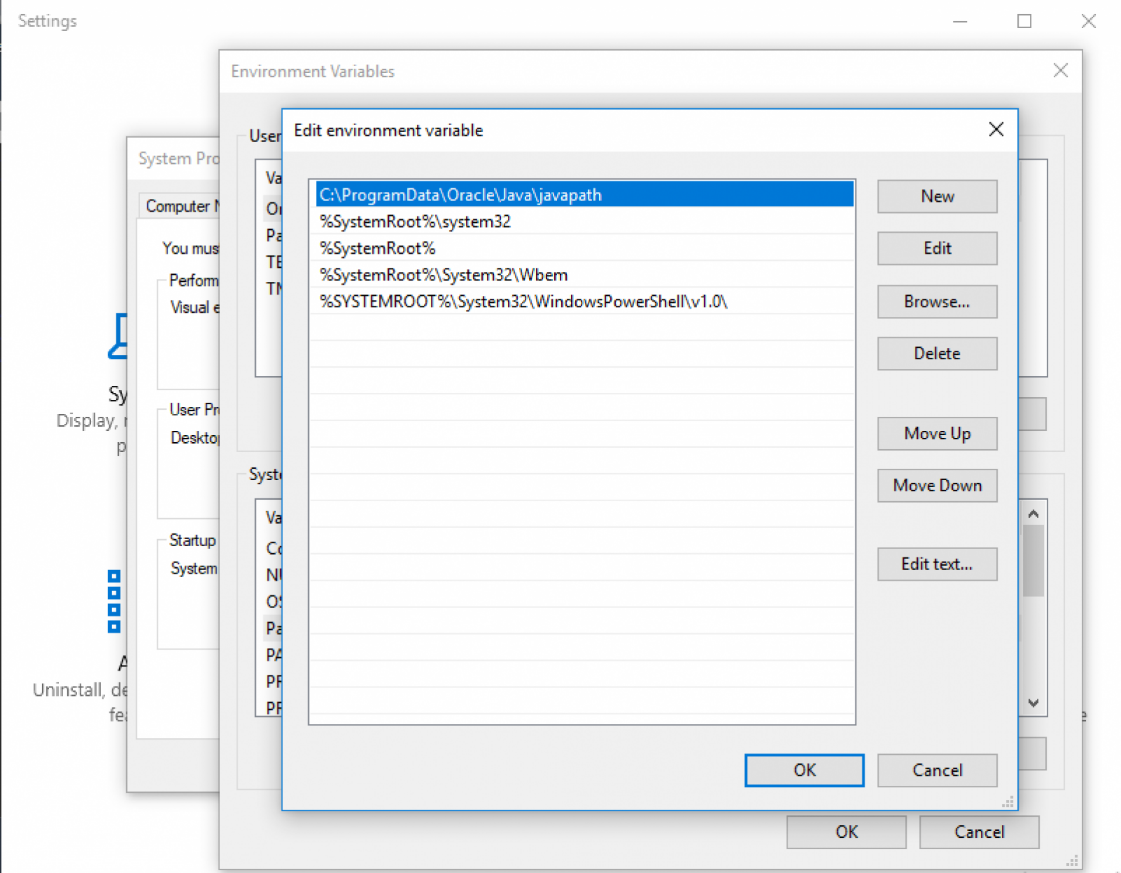
After this, you can update the pre-defined*Path variable*. To do it, select the *Path variable* and click on *Edit* button. In the popup window value section, go to the end and add following:

;C:\Program Files\Java\jdk1.6.0\_25\bin

(Please note that the semicolon ; is used as delimiter,so do not miss that!)

With this, the setup is complete. To check if Javahas been successfully installed in your system, type java –versioncommand in the command prompt,

In Windows 10 and Java 9, you don’t need to perform these steps manually. Java installer takes care of it and you can check the environment variable ***Path*** value, as shown in image below. Note that you do not need to set *JAVA\_HOME* variable.



1. **Download and Install Java IDE**

Although highly recommended, installing Java IDE is an optional step. Java IDE helps a lot in development of Java programs. Install a Java IDE either *Eclipse, Intellij Idea* or *NetBeans IDE*. After you have downloaded the relevant Java IDE platform, install it by running its executable file.

That’s all for downloading and installing Java on Windows 10.

## Summary

* In December of 1990, a team code-named Green was formed to work on creating a simple device that could control a variety of electronics products.
* James Gosling is also known as the father of the Java programming language.
* On April 20, 2009, Oracle signed a deal to purchase Sun Microsystems for $7.4 billion.
* Since Java SE 10, new versions of Java will be released every six months.
* In Java, the source code is compiled and converted into the byte code.
* Using the JVM, we can make the byte code understandable to any platform.
* Java is platform-independent because it does not depend on any type of platform.
* Java SE API provides the basic functionality of the Java programming language.

Chapter 02 – Creating First Java Program

Understanding Text Editors to Write Programs

After installation of Java SE Development Kit (JDK), your first step of the adventure into Java programming world will be to create first Java program.

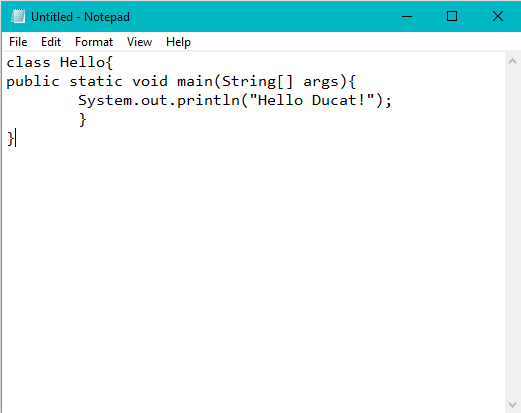
The easiest way to write a simple program is with a text editor. There are various text editors available to write Java program according to operating systems. Some of the popular text editors are:

* Notepad
* Sublime Text
* Notepad++
* Vim
* Atom
* Emacs

Since Windows is a popular choice among users, we can use Notepad to create our first Java Program.

You can also use the text editor of your choice and create a text file with Java program. But make sure to name the text file with extension.java.

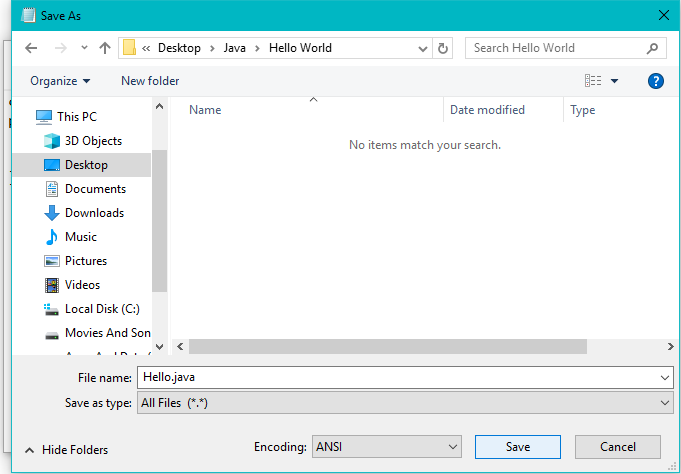
So, let’s create our first Java program *Hello Ducat!* in Notepad.



Every Java program starts from the *main()* method. This program simply prints *Hello Ducat!* to screen.

Once you have written the code in text editor, save the file with .java extension as shown below:

**Note**: You must save the file with same name as name of Java class. In below example, we have saved with Hello.java.



Understanding Java Compiler

Main functionality of a compiler is to convert high level language to low level language which is understand by the machine. The compiler directly convert source code into machine language code which is dependent on particular platform. Wherein the case of Java it is totally different.

We compile the Java source code using Java compiler such as *javac*. Java compiler translates Java source code into Java byte code. Java byte code not an executable or binary file that ready to be run such as in traditional compiler such as C or C++ compiler. Java byte code is a representation of the program in a low-level form similar to machine language code.



Here, we carry on the above example:

To run the java program, copy the path of Hello.java and use CMD to compile and run it.

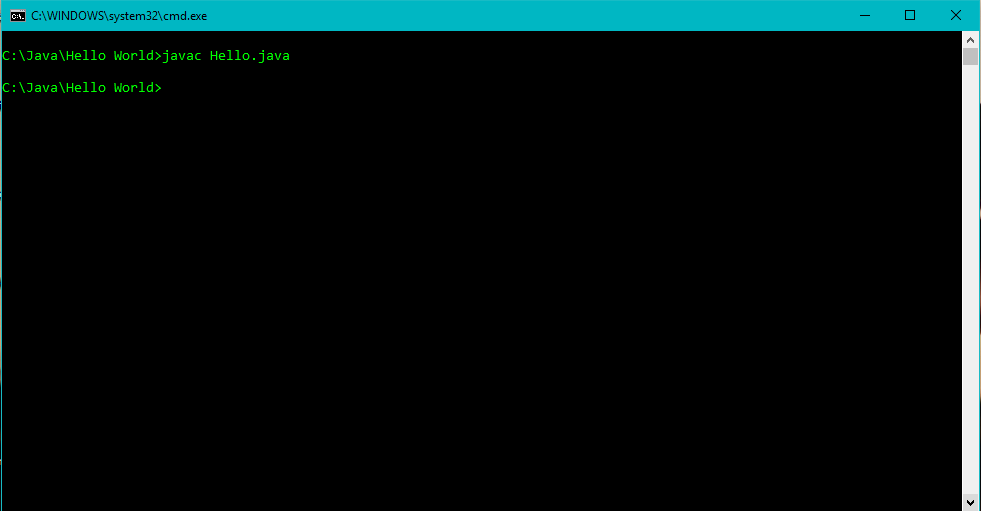
Here we have copied the path

C:\Java\Hello World

Now open CMD and cd to the path we have copied. *javac* keyword is used to compile any java file to class file or we can say byte code. The syntax for *javac* keyword is:

javac <javafile.java>

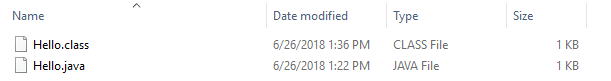
Use the syntax given above to compile Hello.java.



Understanding Byte Code and Its Execution

Java byte code is the essential element of Java Virtual Machine (JVM). The JVM is an emulator that emulates the Java byte code. Java compiler does not directly convert high-level language (direct CPU instruction). It converts the Java language that the developer understands to the Java byte code that the JVM understands. This byte code is stored in the .class file extension.

*javac* compiles the .java file and creates a byte code with same name as Java file name and extension of .class. The compiled file is created in same folder as .java file as shown in the figure below:

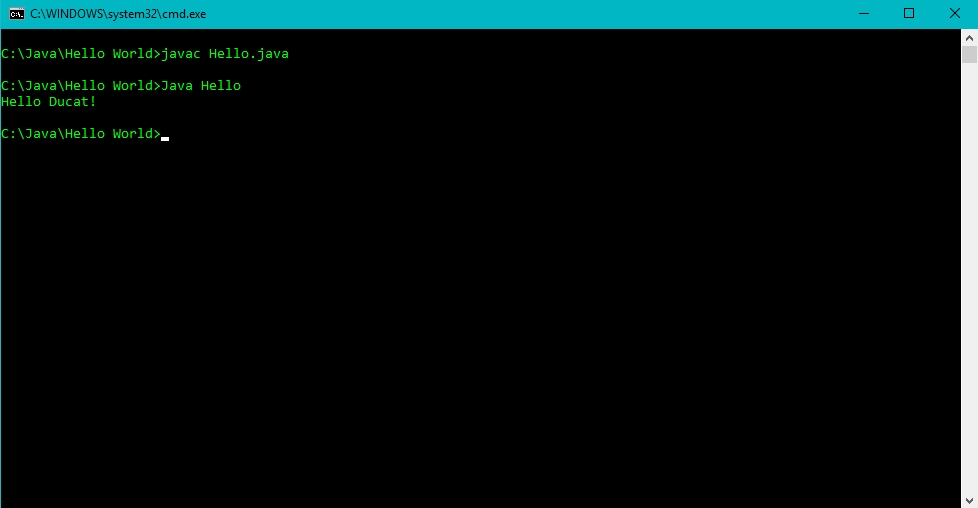


This byte code is actually not native code it is intermediate code. In java there is a virtual machine which converts intermediate code into the native code which is known as JVM.

The JVM works like an interpreter and its basic functionality is to verify byte code. JVM has compiler which is known as JIT (Just In Time) compiler. Therefore, JVM takes byte code and gives it to the JIT and JIT compiler actually compiles the byte code and converts it to machine dependent native code.

After the program is compiled, its time to run the byte code. To run the compiled file use keyword *Java* followed by class name/byte code name and press Enter.

Once the program is executed, following will be the output:



Summary

* After installation of Java SE Development Kit (JDK), your first step is to create first Java program
* The easiest way to write a simple program is with a text editor
* Main functionality of a compiler is to convert high level language to low level language
* Java byte code is a representation of the program in a low-level form similar to machine language code
* In java there is a virtual machine which converts intermediate code into the native code which is known as JVM
* JIT compiler actually compiles the byte code and converts it to machine dependent native code

Chapter 03 – Java Language Fundamentals

Java Identifiers

In programming languages, identifiers are used for identification purpose. Identifiers are the names of variables, methods, classes, packages and interfaces. In Java, a class name, method name, variable name or a label can be used as identifiers.

**For example:**

public class Test

{

public static void main(String[] args)

{

int a = 20;

}

}

In the above Java code, we have used five identifiers. These are:

* **Test**: It is a name of the class being called.
* **main**: It is a method name.
* **String**: It is a predefined class name.
* **args**: It is a variable name.
* **a**:  It is also a variable name.

**Rules for Defining Java Identifiers**

Now note the rules for defining a valid Java identifiers. These rules must be followed to avoid the compile-time error. These rules are also valid for other languages, such as, C, C++.

* **Use of alphanumeric characters is allowed**: The only characters that are accepted for naming the identifiers are alphanumeric characters. They include A-Z, a-z, 0-9, dollar sign ($) and underscore (\_). For example, *fix@* is not a valid java identifier as it contains *@*, as a special character.
* **Identifiers should not start with digits** ([0-9]). For example, *123java* is a not a valid java identifier.
* **Java identifiers are case-sensitive**.
* **No limit on the length of the identifier** is there however it is advisable to use an optimum length of 4 – 15 letters only.
* **Reserved words can’t be used as an identifier**. For example, int while = 20; is an invalid statement as *while* is a reserved word. There are 53 reserved words in Java.

Java Keywords

Keywords are predefined, reserved words used in Java programming that have special meanings for the compiler.

For example, in the statement below, *int* is a keyword. It indicates that the variable score is of integer type (32-bit signed two's complement integer).

int score;

Note that you cannot use the keywords like *int, for, class* as variable names (or identifiers) as they are part of the Java programming language syntax. Here's the complete list of all keywords that are used in Java programming.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Java Keywords List | | | | |
| abstract | assert | boolean | break | byte |
| case | catch | Char | class | const |
| continue | default | Do | double | else |
| enum | extends | Final | finally | float |
| for | goto | If | implements | import |
| instanceof | int | interface | long | native |
| new | package | Private | protected | public |
| return | short | Static | strictfp | super |
| switch | synchronized | This | throw | throws |
| transient | try | Void | volatile | while |

Java Variables

Java variables (or variables used in any other programming language) are containers that hold specified values. A variable is the basic unit of storage in a program.

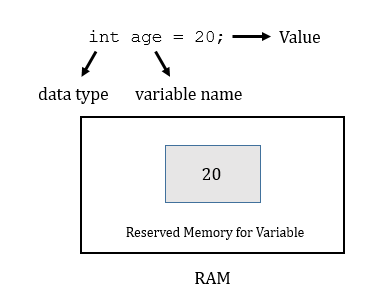
A variable is only a name given to a memory location, all the operations done on the variable effects that memory location. The value stored in a variable can be changed during program execution. In Java, all the variables must be declared before they can be used.

Because Java is a *strongly typed language*, every variable must be declared and initialized or assigned before it is used.

A variable, in the simplest way, is declared by placing a valid type followed by the variable name, with a semicolon at the end. A semi colon at the end of a statement indicates that it ends there.

**Declaring Variables**

We can declare variables in java as follows:



In the above statement, value 20 is stored in a variable age of integer type, where:

* **datatype** is a type of data that can be stored in this variable.
* **variable\_name** is a name given to the variable.
* **value** is the initial value stored in the variable.

**Let’s understand it with help of an example**:

float simpleInterest; //Declaring float variable

int time = 10, speed = 20; //Declaring and Initializing integer variable

char var = 'h'; // Declaring and Initializing character variable

**Initializing Variables**

Initialization of variables refers to the assignment of values to variables. In Java, you can assign a value to variables in the following two ways:

* Static: Static initialization means that the memory is determined for variables when the program starts.
* Dynamic: Dynamic initialization means that in Java, the variables can be declared anywhere in the program. It is because the memory is assigned to them at the time of execution of the statement.

**Let’s understand initialization with help of an example:**

//actual initialization

width = 10;

age = 26.5;

**Types of Variables**

Three types of variables are used in Java. These are:

* Local variables
* Instance variables
* Class / Static variables

Let us now learn about each one of these variables in detail.

**Local Variables**: A variable that is defined within a block/method/constructor is called a local variable. It is created when the:

* Block is entered
* Function is called and destroyed after exiting from the block
* Call returns from the function

The scope of the local variables is limited to the block in which they are declared. i.e. we can access them only within their block.

**Instance Variables:** Instance variables are the non-static variables that are declared in a class outside any given method/constructor/block.

As instance variables are declared in a class, they are created when an object of the class is created and destroyed when the object is destroyed. Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.

**Class or Static Variables:** Static variables are also known as class variables. These variables are declared in the same way as the instance variables are declared. The only difference is that static variables are declared by using the static keyword within a class, outside any method, constructor or block.

Unlike the instance variables, we can only have one copy of a static variable per class, irrespective of the number of objects created. Static variables are created at the beginning of the program execution and destroyed automatically after the execution is over.

To access static variables, we do not need to create any object of that class. We can simply access the variable by using the following command:

class\_name.variable\_name;

Literals in Java

Any constant value that can be assigned to the variable is called a literal/constant. We can specify literals for integers, characters, strings or Boolean characters. For example:

int x = 100; // Here 100 is a constant/literal.

**Integral Literals**

For *Integral* data types (bye, short, int, long), We can specify literals in the following four ways:

* **Decimal literals (Base 10):**In this type of literal, the digits allowed are 0-9.

int x = 101;

* **Octal literals (Base 8):**In this form the allowed digits are 0-7. The octal number should be prefix with 0.

int x = 0146;

* **Hexa-decimal literals (Base 16):**In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive. The hexa-decimal number should be prefix with 0X or 0x.

int x = 0X123Face;

* **Binary literals:**From Java 1.7 onward we can specify literals value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.

int x = 0b1111;

**Char Literal**

For *char* data types, we can specify literals in the following four ways:

* **Single Quote:**We can specify literal to *char* data type as single character within single quote.

char ch = 'a';

* **Char Literal as Integral Literal:** We can specify char literal as integral literal, which represents Unicode value of the character. Such an integral literal can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535.

char ch = 062;

* **Unicode Representation:** We can specify char literals in Unicode representation ‘\uxxxx’. Here xxxx represents 4 hexadecimal numbers.

char ch = '\u0061';// Here /u0061 represent a.

* **Escape Sequence:**Every escape character can be specified as *char* literals.

char ch = '\n';

**String Literal**

Any sequence of characters within double quotes is treated as *string* literals. For example:

String s = "Hello";

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile time expressions, so the following String expression results in a string with three lines of text:

**Example:**

String text = "This is a String literal\n"

+ "which spans not one and not two\n"

+ "but three lines of text.\n";

**Java Boolean Literals**

Boolean literals can take only two values, true and false. Such as:

boolean boolFalse = false;

boolean boolTrue = true;

Note that the words like *true*, *false* and *null* might seem to be keywords, but they are actually literals. They cannot be used as identifiers in a program.

Data Type

A data type is used to specify the size and type of values that can be stored in an identifier. The Java language supports a rich collection of data types, which allows you to select a type as per the need of the application.

In Java, data types are classified into two types:

* Primitive data types—which include Integer, Character, Boolean and Floating Point
* Reference data types—which include Classes, Interfaces and Arrays

**Primitive Data Types**

Let us review each of the primitive data types in detail.

* **Integer:**The integer data types are used to store the whole numbers, for example 123 and −96. They can be further classified into different types, depending upon the size of the values they are used to store. These types include byte, short, int, and long.

Take a note of the range of values that can be stored in each of these data types.

|  |  |  |
| --- | --- | --- |
| Type | Size | Range of Values that can be Stored |
| byte | 1 byte | −128 to 127 |
| short | 2 bytes | −32768 to 32767 |
| int | 4 bytes | −2,147,483,648 to 2,147,483,647 |
| Long | 8 bytes | 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807 |

The range of values that can be stored in a specific type is calculated as:

*−(2n−1) to (2n−1)−1*

Where,

*n* is the number of bits required.

For example, the byte data type requires 1 byte, which is equal to 8 bits. Therefore, the range of values that can be stored in the byte data type is: ­­­­

= −(28−1) to (28−1)−1  
= −27 to (27) -1  
= −128 to 127

* **Floating Point:** Floating point data types are used for representing numbers with a fractional part. They can be of two types, single-precision floating point numbers and double-precision floating point numbers. The single-precision floating point numbers occupy 4 bytes, whereas the double-precision floating point numbers occupy 8 bytes.

Take a note of the range of values that can be stored in each of these data types.

|  |  |  |
| --- | --- | --- |
| Type | Size | Range of Values that can be Stored |
| Float | 4 bytes | 3.4e−038 to 3.4e+038 |
| Double | 8 bytes | 1.7e−308 to 1.7e+038 |

* **Character:** The character data type is used to store the character constants in the memory. Denoted by *char*, it accepts a size of 2 bytes. Essentially it can hold only a single character because it supports the Unicode character sets. It can store a minimum value of ‘u0000’ (or 0) and a maximum value of ‘uffff’ (or 65,535, inclusive).
* **Boolean:** The Boolean data type is used to store values with two states: true or false Values of Boolean data type are not converted implicitly or explicitly (with casts) to any other type.

Operators in Java

An operator is a symbol that is used to perform operations on the values stored in different variables. For example: +, -, \*, /.

The different types of operators used in Java, are:

* Unary operator
* Arithmetic operator
* Shift operator
* Relational operator
* Bitwise operator
* Logical operator
* Ternary operator
* Assignment operator

**Java Unary Operator**

The Java unary operators require just one operand. Unary operators are used to perform various operations, such as:

* Incrementing/decrementing a value by one
* Negating an expression
* Inverting the value of a Boolean

**Example of Java Unary Operators ++ and --**

class OperatorExample{

public static void main(String args[]){

int x=10;

System.out.println(x++);//10 (11)

System.out.println(++x);//12

System.out.println(x--);//12 (11)

System.out.println(--x);//10

}}

Output

10

12

12

10

**Java Arithmetic Operators**

Representing the basic mathematical operations, Java arithmetic operators are used to perform addition, subtraction, multiplication and division.

**Example**

class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

System.out.println(a+b);//15

System.out.println(a-b);//5

System.out.println(a\*b);//50

System.out.println(a/b);//2

System.out.println(a%b);//0

}}

Output

15

5

50

2

0

**Java Left Shift Operator**

Denoted by <<, the Java left shift operator is used to shift all of the bits in a value to the left side by a specified number of times.

**Example**

class OperatorExample{

public static void main(String args[]){

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

System.out.println(20<<2);//20\*2^2=20\*4=80

System.out.println(15<<4);//15\*2^4=15\*16=240

}}

Output

40

80

80

240

**Java Right Shift Operator**

Denoted by >>, the Java right shift operator is used to move the value of the left operand to the right by the number of bits, specified by the right operand.

**Example**

class OperatorExample{

public static void main(String args[]){

System.out.println(10>>2);//10/2^2=10/4=2

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}}

Output

2

5

2

**Java AND Operator Example: Logical && and Bitwise &**

While the bitwise & operator always checks the second condition (regardless of whether the first condition is true or false), the logical && operator checks the second condition, only when the first condition is noted as true.

**Example**

class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a<c);//false && true = false

System.out.println(a<b&a<c);//false & true = false

}}

Output

false

false

**Java OR Operator Example: Logical || and Bitwise |**

The logical || operator checks the second condition, only when the first condition is false.

On the contrary, the bitwise | operator always checks both the conditions, regardless of whether the first condition is true or false.

**Example**

class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a>b||a<c);//true || true = true

System.out.println(a>b|a<c);//true | true = true

//|| vs |

System.out.println(a>b||a++<c);//true || true = true

System.out.println(a);//10 because second condition is not checked

System.out.println(a>b|a++<c);//true | true = true

System.out.println(a);//11 because second condition is checked

}}

Output

true

true

true

10

true

11

**Java Ternary Operator**

Commonly used in Java, Java ternary operator is the only conditional operator that takes three operands. It is used as one liner replacement for *if-then-else* statement.

**Java Ternary Operator Example**

class OperatorExample{

public static void main(String args[]){

int a=2;

int b=5;

int min=(a<b)?a:b;

System.out.println(min);

}}

Output

2

**Java Assignment Operator**

Java assignment operator is one of the most common operators used in the Java programming. It is used to assign the value placed on its right to the operand present on its left.

|  |  |
| --- | --- |
| Operator | Function |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Remainder |

**Java Assignment Operator Example**

class OperatorExample{

public static void main(String args[]){

int a=10;

int b=20;

a+=4;//a=a+4 (a=10+4)

b-=4;//b=b-4 (b=20-4)

System.out.println(a);

System.out.println(b);

}}

Output

14

16

Java Comments

The java comments are the statements that are not executed by the compiler or interpreter. The are used to provide information or explanation about the variables, methods, class or statements used in a program. They can even be used to hide program code for a specific period of time.

**Types of Java Comments**

There are three types of comments in Java. These are:

* **Java single line comment:** The single line comment is used to add one-line comments in a code.

**Syntax:**

//This is single line comment

* **Java multi-line comment:** The multi-line comment is used to a comment with multiple lines in a code.

**Syntax:**

/\*

This

is

multi line

comment

\*/

* **Java documentation comment:** The documentation comment is used to create documentation API. To create a documentation API, you need to use *javadoc tool*.

**Syntax:**

/\*\*

This

is

documentation

comment

\*/

Control Statement

Usually a program executes from the top to bottom, except when we use the control statements. The control statements control the order of execution of commands in a Java program, based on the data values and conditional logic.

In Java, control statements can be divided into the following three categories:

* Condition statements
* Looping statements
* Jump statements

Condition Statements

Condition statements allow you to control the flow of the program execution on the basis of the outcome of an expression or state of a variable known during runtime.  
  
Condition statements can be divided into the following categories:

* The if and if-else statements
* The if-else statements
* The if-else-if statements
* The switch statements

**If Statements**

The first contained statement (that can be a block) of an *if* statement executes only when the specified condition is true. This implies that when the condition is false and there is no *else* keyword then the first contained statement will be skipped and execution continues with the rest of the program. The condition is considered to be an expression that returns a boolean value.

**Example**

import java.util.Scanner;

public class IfDemo

{

    public static void main(String[] args) {

         int age = 25;

        if(age > 18)

            System.out.println("Age is above 18");

    }

  }

Output

Age is above 18

**If-else Statements**  
  
In if-else statements, if the specified condition in the *if* statement is false, then the statement (that can be a block) after the *else* keyword will execute.

**Example**

import java.util.Scanner;

public class IfElseDemo

{

    public static void main( String[] args )

    {

        int age = 15

        if ( age >= 18 )

            System.out.println("Age is above 18 ");

        else

            System.out.println("Age is below 18");

    }

}

Output

Age is below 18

**If-else-if Statements**  
The *if-else-if* statements are executed from the top towards the bottom until the condition is true. As soon as the condition is true, the code inside its body is executed and the control of program will jump outside if-else-if loop.

If all conditions are false, code inside else is executed. Thus, the if-else-if statements take the following structure:

if (condition)  
   statements;   
else if (condition)  
   statements;   
else if (condition)  
   statement;   
else  
   statements;

So whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true, then the else block will execute.  
  
**Example**

class IfElseifTest

{

public static void main(String g[])

{

int i=2;

if(i>3 && i<6)

{

System.out.println("apple");

}

else if(i>=6 && i<8)

{

System.out.println("orange");

}

else if(i>=8)

{

System.out.println("banana");

}

else

{

System.out.println("mango");

}

System.out.println("bye");

}

}

**Switch Statements**

The switch statement in Java is another selection statement that defines multiple paths of execution of a program. It is a multi-way branch statement that gives a better alternative than a large series of if-else-if statements.  
  
**Example**

class SwitchTest

{

public static void main(String g[])

{

int i=3;

switch(i)

{

case 10:

System.out.println("apple");

break;

case 1:

System.out.println("orange");

break;

case 2:

System.out.println("banana");

break;

case 5:

System.out.println("mango");

break;

default:

System.out.println("this is default");

}

System.out.println("bye");

}

}

Output

this is default

bye

While using the switch statements, make sure that you use an expression with a type, either byte, short, int or char. Furthermore, ensure that each of the values specified in the case statement must be of a type compatible with the expression. Although, duplicate case values are not allowed, you may use the break statements in a switch statement. The break statement is used inside the switch to terminate a statement sequence. The break statement is optional in the switch statement.

## Looping Statements

Repeating the same code fragment several times until a specified condition is satisfied is called iteration. Iteration statements execute the same set of instructions until a termination condition is met.  
  
You may use the following loop for iteration statements in Java:

* The while loop
* The for loop
* The do-while loop
* The for each loop

**While Loop**The *while* loop continually executes a statement (that can usually be a block) while a condition is true. The condition must return a boolean value.  
  
**Example**

public class WhileDemo

{

    public static void main( String[] args )

    {

        int i = 0;

        while ( i < 5 )

        {

            System.out.println( "Value :: " + i );

            i++;

        }

    }

}

Output

value :: 0

value :: 1

value :: 2  
value :: 3

value :: 4  
  
  
**Do-while Loop**The *do-while* loop executes at least one time then it will check the expression prior to the next iteration. The only difference between a *while* and a *do-while* loop is that the *do-while* loop evaluates its expression at the bottom of the loop instead of the top.   
  
**Example**

public class DoWhileDemo

{

    public static void main( String[] args )

    {

        int i = 0;

        do

        {

            System.out.println( "value :: " + i );

            i++;

        }

        while ( i < 5);

    }

}

Output

value :: 0

value :: 1

value :: 2  
value :: 3

value :: 4  
  
  
**For Loop**A *for* loop executes a statement (that is usually a block) as long as the boolean condition evaluates to true. A *for* loop is a combination of three elements, an initialization statement, a boolean expression and an increment or decrement statement.  
  
**Syntax:**

for(<initialization>;<condition>;<increment or decrement statement>)

{

<block of code>

}  
  
The initialization block executes at the beginning, that is, before the loop starts. It is used to initialize the loop variable.  
  
The condition statement evaluates every time prior to when the statement (that is usually be a block) executes, if the condition is true then only the statement (that is usually a block) will execute.  
  
The increment or decrement statement executes every time, after the statement (i.e. usually a block).  
  
**Example**

public class WhileDemo

{

    public static void main( String[] args )

    {

        int i = 0;

        while ( i < 5 )

        {

            System.out.println( "Value :: " + i );

            i++;

        }

    }

}

Output

value :: 0

value :: 1

value :: 2  
value :: 3

value :: 4  
  
  
**For Each Loop***For each* loop was introduced in Java 5. This loop is used to traverse the array or collection of elements.   
  
**Example**

public class ForEachDemo

{

    public static void main( String[] args )

    {

        int[] i =

        { 1, 2, 3, 4, 5 };

        for ( int j : i )

        {

            System.out.println( "value :: " + j );

        }

    }

}

Output

value :: 1

value :: 2  
value :: 3

value :: 4

value :: 5

Jump Statements

The jump statements are used in a Java program to unconditionally transfer the control of program to another part of the program. Java provides the following jump statements:

* Break statement
* Continue statement
* Return statement

**Break Statement**The break statement immediately quits the current iteration and shifts the control to the first statement that follows the loop. This is another form of a break that is used in switch statements.  
  
The break statement can take two forms:

* Labelled break statement
* Unlabelled break statement

**Unlabelled Break Statement:** Unlabelled break statement is used to jump control of a program out of the specific loop on the specific condition.  
  
**Example**

public class UnLabeledBreakDemo

{

    public static void main( String[] args )

    {

        for ( int var = 0; var < 5; var++ )

        {

            System.out.println( "Var is : " + var );

            if ( var == 3 )

                break;

        }

    }

}

Output

Var is : 0

Var is : 1

Var is : 2

Var is : 3  
  
**Labelled Break Statement:**This is used for when we want to jump the program control out of nested or multiple loops.  
  
**Example**

public class LabeledBreakDemo

{

    public static void main( String[] args )

    {

        Outer: for ( int var1 = 0; var1 < 5; var1++ )

        {

            for ( int var2 = 1; var2 < 5; var2++ )

            {

                System.out.println( "var1:" + var1 + ", var2:" + var2 );

                if ( var1 == 3 )

                    break Outer;

            }

        }

    }

}

Output

Var1:0, var2:1

Var1:0, var2:2

Var1:0, var2:3

Var1:0, var2:4

Var1:1, var2:1

Var1:1, var2:2

Var1:1, var2:3

Var1:1, var2:4

Var1:2, var2:1

Var1:2, var2:2

Var1:2, var2:3

Var1:2, var2:4

Var1:3, var2:1

**Continue Statement**

The continue statement is used when we want to continue running the loop with the next iteration and skip the rest of the statements of the body for the current iteration.  
  
The continue statement has the following two forms:

* Labelled continue statement
* Unlabelled continue statement

**Unlabelled Continue Statement:**This statement skips the current iteration of the innermost *for*, *while* and *do-while* loop.  
  
**Example**

public class UnlabeledContinueDemo

{

    public static void main( String[] args )

    {

        for ( int var1 = 0; var1 < 4; var1++ )

        {

            for ( int var2 = 0; var2 < 4; var2++ )

            {

                if ( var2 == 2 )

                    continue;

                System.out.println( "var1:" + var1 + ", var2:" + var2 );

            }

        }

    }

}

Output

Var1:0, var2:2

Var1:0, var2:1

Var1:0, var2:3

Var1:1, var2:0

Var1:1, var2:1

Var1:1, var2:3

Var1:2, var2:0

Var1:2, var2:1

Var1:2, var2:3

Var1:3, var2:0

Var1:3, var2:1

Var1:3, var2:3

**Labelled Continue Statement:** This statement skips the current iteration of the loop with the specified label.  
  
**Example**

public class LabeledContinueDemo

{

    public static void main( String[] args )

    {

        Outer: for ( int var1 = 0; var1 < 5; var1++ )

        {

            for ( int var2 = 0; var2 < 5; var2++ )

            {

                if ( var2 == 2 )

                    continue Outer;

                System.out.println( "var1:" + var1 + ", var2:" + var2 );

            }

        }

    }

}

Output

Var1:0, var2:2

Var1:0, var2:1

Var1:1, var2:0

Var1:1, var2:1

Var1:2, var2:0

Var1:2, var2:1

Var1:3, var2:0

Var1:3, var2:1

Var1:4, var2:0

Var1:4, var2:1

**Return Statement**The *return* statement is used to immediately quit the current method and return to the calling method. It is compulsory to use a return statement for non-void methods to return a value.   
  
**Example**

public class ReturnDemo

{

    public static void main( String[] args )

    {

        ReturnDemo returnDemo = new ReturnDemo();

        System.out.println( "No : " + returnDemo.returnCall() );

    }

     int returnCall()

    {

        return 5;

    }

}

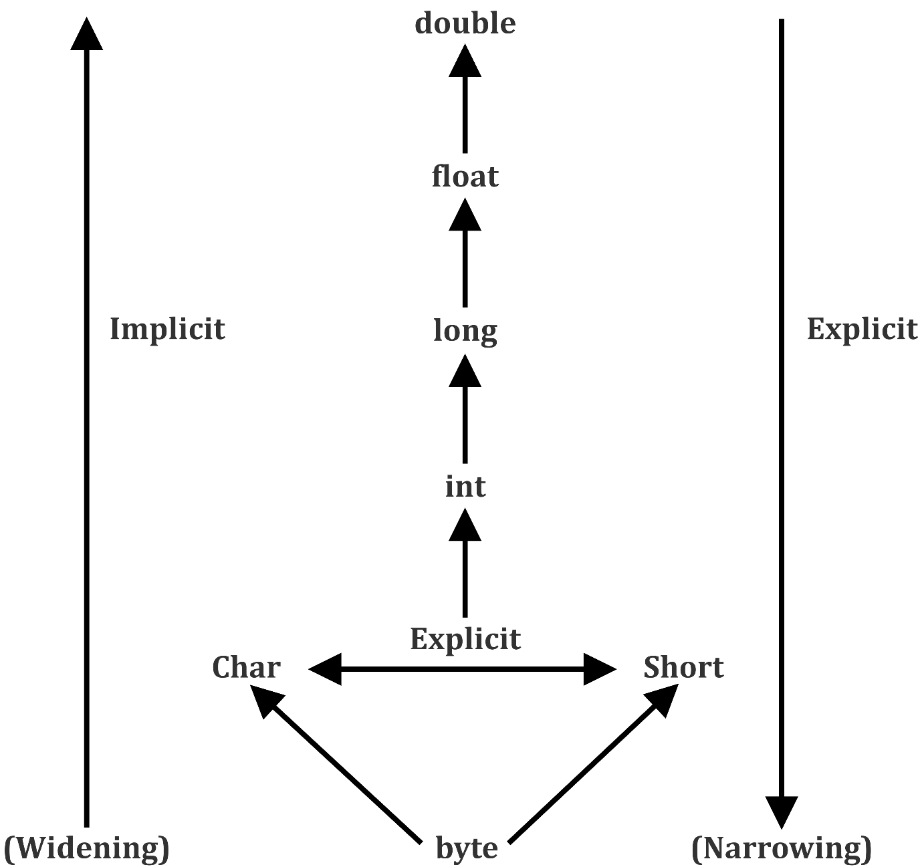
Output  
  
 No : 5

Type Casting

Type Casting in Java is nothing but converting a primitive or interfaceor class in Java into other type. There is a rule in Java that classes or interface which shares the same type hierarchy only can be typecasted. If there is no relationship between then Java will throw *ClassCastException*. Type casting are of two types they are:

1. Implicit Casting (Widening)
2. Explicit Casting (Narrowing)

The diagram or flowchart of these above type casting is given below:



**Data Type Hierarchy**

1. **Widening or Implicit Casting in Java or Automatic type conversion**

Automatic type conversion can happen if both types are compatible and target type is largerthan the source type**.**

**Implicit Casting of a Primitive**

No explicit casting required for the above mentioned sequence.

**Implicit Casting of a Class Type**

When we assignsmaller type to alarger type, there is no need for a casting required. Same applies to the class type as well.

**Example of implicit casting**

public class Test

{

public static void main(String[] args)

{

int i = 100;

long l = i;

float f = l;

System.out.println("Int value "+i);

System.out.println("Long value "+l);

System.out.println("Float value "+f);

}

}

Output

Int value 100

Long value 100

Float value 100.0

1. **Narrowing or Explicit Casting in Java**

When you assign a larger type to a smaller type**,** then explicit casting is required.

**Narrowing a Class Type**

When we assign larger type to a smaller type, then we need to explicitly type cast it.

**Example of Explicit casting**

class Test

{

public static void main(String[] args)

{

double d = 100.04;

//explicit type casting

long l = (long)d;

//explicit type casting

int i = (int)l;

System.out.prinln("Double value "+d);

System.out.println("Long value "+l);

System.out.println("Int value "+i);

}

}

Output

Double value 100.04

Long value 100

Int value 100

Summary

* The only characters that are accepted for naming the identifiers are alphanumeric characters
* The multi-line comment is used to a comment with multiple lines in a code
* The logical || operator checks the second condition, only when the first condition is false
* Java ternary operator is the only conditional operator that takes three operands
* A variable is only a name given to a memory location, all the operations done on the variable effects that memory location
* Type Casting in Java is nothing but converting a primitive or interfaceor class in Java into other type

# Chapter 04 – Object Oriented Technique and Implementation

Why Object Oriented Programming (OOP)

Object oriented programming is a concept that was created because of the need to overcome the problems that were found with using structured programming techniques. While structured programming uses an approach which is top down, OOP uses an approach which is bottom up. Traditionally, programming has placed an emphasis on logic and actions.  
  
Object oriented programming has taken a completely different direction, and will place an emphasis on objects and information. With object oriented programming, a problem will be broken down into a number of units. These units are called objects. The foundation of OOP is the fact that it will place an emphasis on objects and classes.  
  
Objects will be defined, and they will interact inside the system in a number of different ways. There are a number of advantages to be found with using the OOP paradigm, and some of these are simple maintenance, an advanced analysis of complicated programs, and re-usability.   
  
There are a number of programming languages that use OOP, and some of these are Java, C++, and Ada. One concept that you will want to become familiar with is data modeling. Before you can construct an object oriented system, you will first need to find the objects within the system and determine the relationships they have. This process is called data modeling.

OOP Concepts with Real Life Examples

OOP refers to a methodology or paradigm that is used to design a program involving classes and objects. It simplifies the software development and maintenance process by providing some concepts, such as:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

Object

**In Real Life**: Any real worldentity (either logical or physical) that has state and behaviour is known as an object e.g. chair, bike, marker, pen, table, car, human, account, department etc.

**In Programming**: An object is an instance of a class.

**In Memory:** An object is a dynamically allocated block

Syntax to Create an Object

class\_name ref\_variable = new constructor\_name();

There are two steps when creating an object from a class.

**Instantiation:** The 'new' keyword or operator is used to create the object.

**Initialization:** The 'new' keyword is followed by a call to a constructor, this call initializes the new object.

Class

A class is an entity that determines how an object will behave and what the object will contain. In other words, *it is a blueprint or a set of instruction to build specific type of objects*.

Examples:Design of a car,Model of a building

Syntax to Create a Java Class

A class is declared by the use of the *class* keyword. The general form of a class definition is shown here:

Class ClassIdentifier {

instance (object) members

class members

constructors

instance initializers(instance blocks)

class initializers (static blocks)

nested types

}

Declaration of Reference Variables

Declaration of reference variables in Java is similar as we declare variables of built in data types that is,

data\_type ref\_variable;

For example,

int number;

Same way we can declare an object variable that refers to type “Class”. It takes the following syntax,

Class\_name ref\_variable;

For example,

Person p;

will declare object variable “p” that refers to type “Person” in the memory heap.(Here we can consider our class to be a user defined data-type.)

If you declare “p” like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the new operator, as described in the next section.

You must assign an object to “p” before you use it in your code. Otherwise, you will get a compiler error.

Instantiation of Objects

Once we have declared the reference variable we need to instantiate it with an object of the corresponding data-type i.e. class.

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

In short, Instantiating the object means nothing but just allocating the heap memory of the declared object variable of class type.

For above example we can instantiate the object variable “p” like this,

p = new Person();

We can also declare and instantiate the object by following,

Person p = new Person();

**Anonymous object**

Anonymous simply means nameless. An object which has no reference is known as anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, anonymous object is a good approach. For example:

**Syntax:**

new constructor\_name();

**Example**

new Calculation();//anonymous object

Calling method through anonymous object

new Calculation().fact(5);

Let's see the full example of anonymous object in java.

class Calculation{

void fact(int n){

int fact=1;

for(int i=1;i<=n;i++){

fact=fact\*i;

}

System.out.println("factorial is "+fact);

}

public static void main(String args[]){

new Calculation().fact(5);//calling method with anonymous object

}

}

Output

Factorial is 120

Constructors and Initializers

In Java, constructor is a block of codes similar to method. It is called when an instance of object is created, and memory is allocated for the object.

It is a special type of method which is used to initialize the object. The new operator is followed by a call to a constructor, which initializes the new object.

**When a constructor is called**

Every time when an object is created using *new* keyword, at least one constructor is called.

**Rules for creating java constructor**

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

**Types of java constructors**

There are two types of constructors in java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

**Java Default Constructor**

A constructor is called "Default Constructor" when it doesn't have any parameter.

**Syntax of default constructor:**

<class\_name>(){}

**Example of default constructor**

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |

class Bike1{

Bike1(){System.out.println("Bike is created");}

public static void main(String args[]){

Bike1 b=new Bike1();

}

}

Output

Bike is created

**Java parameterized constructor**

A constructor which has a specific number of parameters is called parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.

|  |
| --- |
| **Example of parameterized constructor**  In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor. |

class Student4{

    int id;

    String name;

    Student4(int i,String n){

    id = i;

    name = n;

    }

    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){

    Student4 s1 = new Student4(111,"Karan");

    Student4 s2 = new Student4(222,"Aryan");

    s1.display();

    s2.display();

   }

}

Output

111 Karan

222 Aryan

**Note:**

If a class does not contain any constructor,compiler provides default constructor at the time of compilation.

**Instance Initializer or Instance Block:**

An instance initializer is a line of code (or a block of code) placed outside any method, constructor, or other block of code. Instance blocks are used to initialize instance variables. Instance blocks are executed before constructors regardless of which constructor is used to create the instance.

class Test

{

// Instance Block

{

System.out.println("Instance block");

}

// Constructor

Test()

{

System.out.println("Constructor Called");

}

public static void main(String[] args)

{

Test t = new Test();

}

}

Output:

Instance block

Constructor Called

**Static Block**

Static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

**Example**

class JavaExample{

static int num;

static String mystr;

static{

num = 97;

mystr = "Static keyword in Java";

}

public static void main(String args[])

{

System.out.println("Value of num: "+num);

System.out.println("Value of mystr: "+mystr);

}

}

Output

Value of num: 97

Value of mystr: Static keyword in Java

Adding Attributes to Object

Adding attributes to object means declaring instance or non-static variables.

There are several kinds of variables:

* Member variables in a class—either instance or static variables
* Variables in a method or block of code—these are called *local variables*.
* Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

public int cadence;

public int gear;

public int speed;

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

**Example**

public class Bicycle {

private int cadence;

private int gear;

private int speed;

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

public int getCadence() {

return cadence;

}

public void setCadence(int newValue) {

cadence = newValue;

}

public int getGear() {

return gear;

}

public void setGear(int newValue) {

gear = newValue;

}

public int getSpeed() {

return speed;

}

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

Adding Behaviors (method) to Object

In Java we have two types of methods:

* Instance or non-static methods: represents behaviour of individual object.
* Static or class methods: represents behaviour of all objects or class

Adding behaviour to object means defining instance method in the class . We can understand it with an example.

Here is a syntax of method declaration:

modifier returnType nameOfMethod (Parameter List) {

// method body

}

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:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. 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.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

**Method Calling:**

To call instance method we need object and to call static method we need class name.

objectRef.instanceMethod();

Classname.staticMethod();

**Example**

Class Test{

public void minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

System.out.println(“Min Element:”+min);

}

Public static void main(String args[])

{

Test t=new Test()

t.minFunction(10,8);

}

}

Example-2

Class Test{

Public static void minFunction(int n1, int n2) {

int min;

if (n1 > n2)

min = n2;

else

min = n1;

System.out.println(“Min Element:”+min);

}

Public static void main(String args[])

{

Test.minFunction(10,8);

}

*Static* Keyword and Its Usage

The static keyword is used in java mainly for memory management. It is used with variables, methods, blocks and nested class. It is a keyword that are used for share the same variable or method of a given class. This is used for a constant variable or a method that is the same for every instance of a class. The main method of a class is generally labeled static.

No object needs to be created to use static variable or call static methods, just put the class name before the static variable or method to use them. Static method can not call non-static method.

In java language static keyword can be used for following:

* Static variable (also known as class variable): static int count;
* Static method (also known as class method): static void foo() {}
* Static block: Static {

// some code

}

* Nested class

Class Test{

Static class InnerStatic{

}

}

* Interface static method (Java 8 onwards)

*this* Keyword and Its Usage

Keyword *this* is a reference variable in Java that refers to the current object.

The various usages of '*this*' keyword in Java are as follows:

* It can be used to refer instance variable of current class
* It can be used to invoke or initiate current class constructor
* It can be passed as an argument in the method call
* It can be passed as argument in the constructor call
* It can be used to return the current class instance

**Usage of java this keyword**

Here is given the 6 usage of java this keyword.

1. ***this* can be used to refer current class instance variable:**

The *this* keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

**Example**

class Student{

int rollno;

String name;

float fee;

Student(int rollno,String name,float fee){

this.rollno=rollno;

this.name=name;

this.fee=fee;

}

void display(){System.out.println(rollno+" "+name+" "+fee);}

}

class TestThis2{

public static void main(String args[]){

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

s1.display();

s2.display();

}}

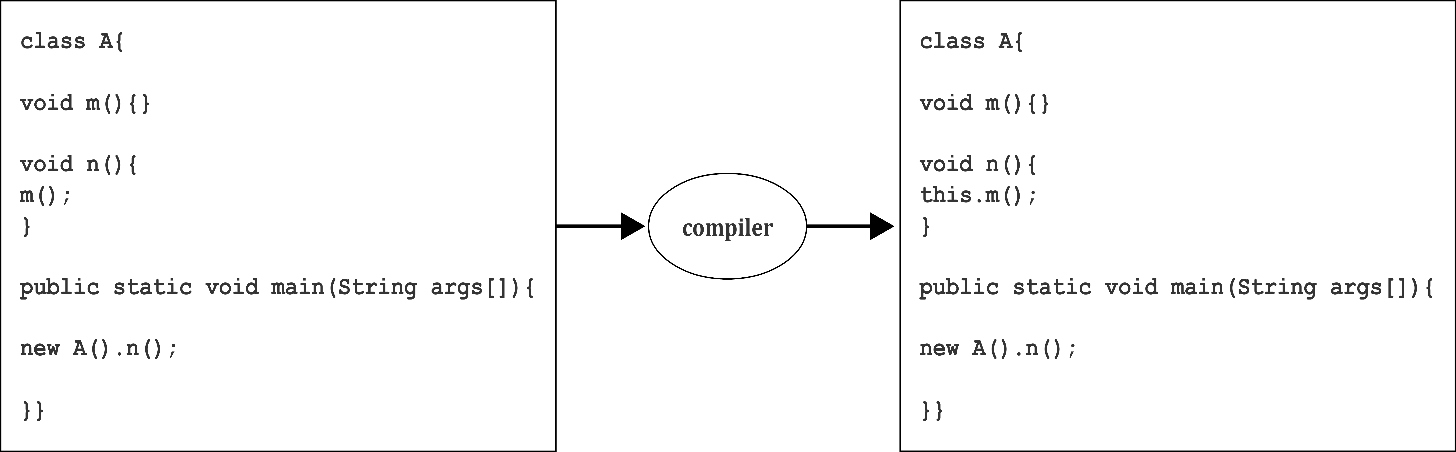
Output

111 ankit 5000

112 sumit 6000

1. ***this* can be used to invoke current class method (implicitly):**

You may invoke the method of the current class by using the *this* keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



**Example**

class A{

void m(){System.out.println("hello m");}

void n(){

System.out.println("hello n");

//m();//same as this.m()

this.m();

}

}

class TestThis4{

public static void main(String args[]){

A a=new A();

a.n();

}}

Output

hello n

hello m

1. ***this()* can be used to invoke current class constructor:**

The *this()* constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Example**

class A{

A(){System.out.println("hello a");}

A(int x){

this();

System.out.println(x);

}

}

class TestThis5{

public static void main(String args[]){

A a=new A(10);

}}

Output

hello a

10

1. ***this* can be passed as an argument in the method call:**

The *this*keyword can also be passed as an argument in the method. It is mainly used in the event handling.

**Example**

class S2{

  void m(S2 obj){

  System.out.println("method is invoked");

  }

  void p(){

  m(this);

  }

  public static void main(String args[]){

  S2 s1 = new S2();

  s1.p();

  }

}

Output

method is invoked

1. ***this* can be passed as argument in the constructor call:**

We can pass *this* keyword in the constructor also. It is useful if we have to use one object in multiple classes.

**Example**

class B{

  A4 obj;

  B(A4 obj){

    this.obj=obj;

  }

  void display(){

    System.out.println(obj.data);//using data member of A4 class

  }

}

class A4{

  int data=10;

  A4(){

   B b=new B(this);

   b.display();

  }

  public static void main(String args[]){

   A4 a=new A4();

  }

}

Output

Output:10

1. ***this* can be used to return the current class instance from the method:**

We can return this keyword as a statement from the method. In such case, return type of the method must be the class type (non-primitive).

**Example**

class A{

A getA(){

return this;

}

void msg(){System.out.println("Hello Java");}

}

class Test1{

public static void main(String args[]){

new A().getA().msg();

}

}

Output

Hello Java

Inheritance

Inheritance is a mechanism of extending an entity i.e.creating a new entity by using an existing entity. Existing entity is known as Base or Super type newly created entity is known as Child or sub type.

**Advantages:**

* It provides code reusability.
* It is used to achieve runtime polymorphism.

**Example**

Initially mobile phones were limited to send message, dial & receive call. Today the mobile companies have upgraded their mobile phones by extending the mobile class functionality and adding new features like camera, music, video, games etc.



**Syntax of Inheritance:**

public class ChildClass extends ParentClass {

  //new variable or methods here

}

Below is a simple example of inheritance in java.

class SuperClassA {

public void foo(){

System.out.println("SuperClassA");

}

}

class SubClassB extends SuperClassA{

public void bar(){

System.out.println("SubClassB");

}

}

public class Test {

public static void main(String args[]){

SubClassB a = new SubClassB();

a.foo();

a.bar();

}

}

**Note:**

Constructors, private members and initializers are not inherited.

Types of Inheritance

Below are the different types of inheritance which is supported by Java:

* **Single Inheritance:**When a class extends another class (only one class) then we call it as Single inheritance.
* **Multiple Inheritance**: Multiple inheritance is nothing but oneclass extendingmorethan one class.
* **Multilevel Inheritance:**In multilevel inheritance a derived class inherits a parent class as well as the derived class acts as the parent class to other class.
* **Hierarchical Inheritance:**In Hierarchical inheritance one parent class will be inherited by many sub classes.
* **Hybrid Inheritance:**Hybrid Inheritance is the combination of both Single and Multiple Inheritance.

Object class as root of Java class hierarchy

The Object class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Variable Hiding

Variable hiding happens when we declare a property in a local scope that has the same name as the one we already have in the outer scope.

**Example of variable hiding**

public class HideVariable {

     private String message = "this is instance variable";

     HideVariable() {

        String message = "constructor local variable";

        System.out.println(message);

    }

     public void printLocalVariable() {

        String message = "method local variable";

        System.out.println(message);

    }

     public void printInstanceVariable() {

        String message = "method local variable";

        System.out.println(this.message);

    }

}

Here we have the message variable declared in 4 different places. The local variables declared inside of the constructor and two methods are hiding the instance variable.

Method Hiding

Method hiding means subclass has defined a static method with the same signature as a static method in the superclass.

In that case the method of superclass is hidden by the subclass.

**Example of method hiding:**

publicclassBaseMethodClass {

    publicstaticvoidprintMessage() {

        System.out.println("base static method");

    }

}

*BaseMethodClass* has a single *printMessage() static* method.

Next, let’s create a child class with the same signature as in the base class:

public class ChildMethodClass extends BaseMethodClass {

    public static void printMessage() {

        System.out.println("child static method");

    }

}

**Here’s how it works:**

ChildMethodClass.printMessage();

The output after calling the *printMessage()* method:

child static method

The *ChildMethodClass.printMessage()*hides the method in *BaseMethodClass*.

Method Overloading

Overloading occurs when two or more methods in one class have the same method name but different parameters.

**Different ways of doing overloading methods**

* The number of parameters in two methods.
* The data types of the parameters of methods.
* The Order of the parameters of methods.

**By changing the number of parameters.**

class Addition{

    // adding two integer values.

    public int add(int a, int b){

        int sum = a+b;

        return sum;

    }

    // adding three integer values.

    public int add(int a, int b, int c){

        int sum = a+b+c;

        return sum;

    }

}

class GFG {

    public static void main (String[] args) {

        Addition ob = new Addition();

        int sum1 = ob.add(1,2);

        System.out.println("sum of the two integer value :" + sum1);

        int sum2 = ob.add(1,2,3);

        System.out.println("sum of the three integer value :" + sum2);

    }

}

Output

sum of the two integer value :3

sum of the three integer value :6

**By changing the Data types of the parameters**

classAddition{

    // adding three integer values.

    publicintadd(inta, intb, intc){

        intsum = a+b+c;

        returnsum;

    }

    // adding three double values.

    publicdoubleadd(doublea, doubleb, doublec){

        doublesum = a+b+c;

        returnsum;

    }

}

classGFG {

    public static void main (String[] args) {

        Addition ob = newAddition();

        intsum2 = ob.add(1,2,3);

        System.out.println("sum of the three integer value :"+ sum2);

        doublesum3 = ob.add(1.0,2.0,3.0);

        System.out.println("sum of the three double value :"+ sum3);

    }

}

Output

sum of the three integer value :6

sum of the three double value :6.0

**By changing the Order of the parameters**

class Student{

public void studentIdentity(String name, int id){

System.out.println("studentName :"+ name +" "+"Id :"+ id);

}

public void studentIdentity(int id, String name){

System.out.println("studentName :"+ name +" "+"Id :"+ id);

}

}

class GFG {

public static void main (String[] args) {

Student student = new Student();

student.studentIdentity("Mohit", 1);

student.studentIdentity("shubham", 2);

}

}

Output

studentName :Mohit Id :1

studentName :shubham Id :2

Method Overriding

Redefining instance method of parent in child class is known as method overriding.

**Example of method overriding:**

class Animal {

void eat() { System.out.println("Animal eating"); }

}

class Dog extends Animal {

void eat() { System.out.println("Dog eating"); }

}

In Java, it’s always the runtime type that determines which instance method is invoked. So if an *Animal* variable refers to a *Dog*object, *a.eat()* will invoke the overriding *Dog.eat* method.

Super Keyword and Its Usage

Thesuper keyword in java is a reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

**Usage of java super Keyword**

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

**1) super is used to refer immediate parent class instance variable.**

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

class Animal{

String color="white";

}

class Dog extends Animal{

String color="black";

void printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}}

Output

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

**2) super can be used to invoke parent class method**

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}

void work(){

super.eat();

bark();

}

}

class TestSuper2{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}}

Output

eating...

barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

**3) super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

class Animal{

Animal(){System.out.println("animal is created");}

}

class Dog extends Animal{

Dog(){

super();

System.out.println("dog is created");

}

}

class TestSuper3{

public static void main(String args[]){

Dog d=new Dog();

}}

Output

animal is created

dog is created

Using final to Prevent Overriding and Inheritance

**Using final to Prevent Inheritance**

When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when creating an immutable class like the predefined String class.

The following fragment illustrates *final* keyword with a class:

final class A

{

// methods and fields

}

// The following class is illegal.

class B extends A

{

// ERROR! Can't subclass A

}

**Note:**

* Declaring a class as final implicitly declares all of its methods as final, too.
* It is illegal to declare a class as both *abstract* and *final* since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. For more on abstract classes, refer abstract classes in java

**Using final to Prevent Overriding**

When a method is declared as final then it cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

The following fragment illustrates *final*keyword with a method:

class A

{

final void m1()

{

System.out.println("This is a final method.");

}

}

class B extends A

{

void m1()

{

// ERROR! Can't override.

System.out.println("Illegal!");

}

}

Up-casting and Down-casting

In your daily coding, you will see and use upcasting and downcasting occasionally. You may hear the terms ‘casting’, ‘upcasting’, ‘downcasting’ from someone or somewhere, and you may be confused about them. As you read on, you will realize that upcasting and downcasting are really simple.

Before we go into the details, suppose that we have the following class hierarchy:

Mammal > Animal > Dog, Cat

Mammal is the super interface. Animal is the abstract class. Dog and Cat are the two concrete sub classes:

**Upcasting**

Upcastingis process of casting a subtype to a supertype, upward to the inheritance tree. Let’s see an example:

|  |  |
| --- | --- |
|  | Dog dog = new Dog();  Animal anim = (Animal) dog;  anim.eat(); |

Here, we cast the Dog type to the Animal type. Because Animal is the supertype of Dog, this casting is called upcasting.

Note that the actual object type does not change because of casting. The Dog object is still a Dog object. Only the reference type gets changed. Hence the above code produces the following output:

|  |  |
| --- | --- |
|  | Dog is eating… |

**Why is Upcasting?**

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

|  |  |
| --- | --- |
|  | publicclassAnimalTrainer {      publicvoidteach(Animal anim) {          anim.move();          anim.eat();      }  } |

Here, the *teach()*method can accept any object which is subtype of Animal. So objects of type Dog and Cat will be upcasted to Animal when they are passed into this method:

Dog dog = newDog();

Cat cat = newCat();

 AnimalTrainer trainer = newAnimalTrainer();

trainer.teach(dog);

trainer.teach(cat);

**Downcasting**

Downcasting is a process of casting to a subtype, downward to the inheritance tree. Let’s see an example:

Animal anim = new Cat();

Cat cat = (Cat) anim;

Here, we cast the Animal type to the Cat type. As Cat is subclass of Animal, this casting is called downcasting.

Unlike upcasting, downcasting can fail if the actual object type is not the target object type. For example:

Animal anim = new Cat();

Dog dog = (Dog) anim;

This will throw a *ClassCastException* because the actual object type is Cat. And a Cat is not a Dog so we cannot cast it to a Dog.

**Why is Downcasting?**

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.

Consider the following example:

public class AnimalTrainer {

public void teach(Animal anim) {

// do animal-things

anim.move();

anim.eat();

// if there's a dog, tell it barks

if (anim instanceof Dog) {

Dog dog = (Dog) anim;

dog.bark();

}

}

}

Here, in the *teach()* method, we check if there is an instance of a Dog object passed in, downcast it to the Dog type and invoke its specific method, *bark()*.

Okay, so far you have got the nuts and bolts of upcasting and downcasting in Java. Remember:

* Casting does not change the actual object type. Only the reference type gets changed.
* Upcasting is always safe and never fails.
* Downcasting can risk throwing a *ClassCastException*, so the *instanceof* operator is used to check type before casting.

Polymorphism

An object can take multiple form, that object is knows as polymorphic object. More specifically child object may also play role of parent.

In Syntax, Polymorphism is nothing but polymorphic up casting.

Parent ref\_variable = new child\_object();

Polymorphism also refers to the concept of doing the same task in different ways. Consider the following examples to understand the concept of polymorphism better:

* When an executive tries to convince a customer by using two differently approaches
* When you try to draw a shape in two different ways
* When people speak the same thing in different languages

Dynamic Binding / Dynamic Method Dispatch

Dynamic binding also called dynamic dispatch is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time. Dynamic binding is also known as late binding or run-time binding.

Dynamic binding is an object oriented programming concept and it is related with polymorphism and inheritance.

Dynamic binding (dispatch) means that a block of code executed with reference to a procedure (method) call is determined at run time.

Dynamic dispatch is generally used when multiple classes contain different implementations of the same method. It provides a mechanism for selecting the function to be executed from various function alternatives at the run-time. In C++, virtual functions are used to implement dynamic binding.

**Example of dynamic binding**

class Flower

{

public void smell()

{

System.out.pritnln("Class Flower");

}

}

class Rose extends Flower

{

public void smell()

{

System.out.println("Class Rose");

}

public static void main(String[] args)

{

Flower fl=new Rose ();

fl.smell();

}

}

Output

Class Rose

Implementing Runtime Polymorphism

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at run-time. This is how Java implements run-time polymorphism. When an overridden method is called by a reference, Java determines the version of that method to execute it. It does so on the basis of the type of object it refers to. In simple words, the type of object which it refers, determines the version of the overridden method to be called.

In short,

Dynamic Binding + Upcasting = Runtime Polymorphism

**Example**

class Game

{

public void type()

{ System.out.println("Indoor & outdoor"); }

}

Class Cricket extends Game

{

public void type()

{ System.out.println("outdoor game"); }

public static void main(String[] args)

{

Game gm = new Game();

Cricket ck = new Cricket();

gm.type();

ck.type();

gm=ck; //gm refers to Cricket object

gm.type(); //calls Cricket's version of type

}

}

Output:

Indoor & outdoor

Outdoor game

Outdoor game

Notice the last output. This is so because of *gm = ck*. Now *gm.type()*will call *Cricket* version of *type* method. Here, *gm* refers to *cricket* object.

Now, note the advantages of the dynamic method dispatch.

* Dynamic method dispatch allows Java to support the overriding of methods that is central for run-time polymorphism.
* It allows a class to specify methods that are common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
* It also allows the subclasses to add its specific methods subclasses to define the specific implementation of some.

Abstraction

Abstraction refers to the concept of hiding internal details and showing the functionality. A phone call is an example of abstraction - we don't know the internal processing of how a call is terminated on the receiving number.

In Java, we use abstract class and interface to achieve abstraction.

abstract keyword (abstract classes and methods)

**Abstract Class**

An abstract class is a class that can never be instantiated.It is used to provide abstraction. Note that an abstract class does not provide 100% abstraction because it may contain a concrete method as well.

**Syntax:**

abstract class class\_name { }

**Note the key points about abstract classes:**

* An abstract class may or may not have an abstract method. But, if any class has even a single abstract method, then it must be declared abstract.
* Abstract classes can have constructors, member variables and normal methods.
* Abstract classes are never instantiated.
* When you extend an abstract class with abstract method, you must define the abstract method in the child class or make the child class abstract.

**Example of Abstract class**

abstract class Student

{

public void name() // concrete (non-abstract) method

{

System.out.println("Name is Adam");

}

public void marks() // concrete (non-abstract) method

{

System.out.println("Marks scored are 80");

}

public static void main(String args[])

{

Student s1 = new Student(); // Error raised, see the errror in screenshot

}

}

Output

Student.java:13: error: Student is abstract; cannot be instantiated

Student s1 = new Student(); // Error raised, see the errror in screenshot

^

1 error

**Note:**

Concrete methods of abstract class can be executed on it’s class object

**Abstract Method**

The methods that are declared without any body within an abstract class are called abstract methods. The method’s body, in this case, is defined by its subclass. An abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

**Syntax:**

abstract return\_type function\_name (); //No definition

**Abstract method in an abstract class**

//abstract class

abstract class Sum{

/\* These two are abstract methods, the child class

\* must implement these methods

\*/

public abstract int sumOfTwo(int n1, int n2);

public abstract int sumOfThree(int n1, int n2, int n3);

//Regular method

public void disp(){

System.out.println("Method of class Sum");

}

}

//Regular class extends abstract class

class Demo extends Sum{

/\* If I don't provide the implementation of these two methods, the

\* program will throw compilation error.

\*/

public int sumOfTwo(int num1, int num2){

return num1+num2;

}

public int sumOfThree(int num1, int num2, int num3){

return num1+num2+num3;

}

public static void main(String args[]){

Sum obj = new Demo();

System.out.println(obj.sumOfTwo(3, 7));

System.out.println(obj.sumOfThree(4, 3, 19));

obj.disp();

}

}

Output

10

26

Method of class Sum

**Note the key points about abstract method**:

* Abstract methods don’t have body, they just have method signature.
* If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.
* If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Understanding Interfaces

In the Java programming language, an *interface* is a reference type, similar to a class, which can contain *only* constants, method signatures, default methods, static methods,private methods and nested types. Method bodies exist only for default methods ,static methods and private methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

In short,

**Interface is collection of**:

* Public Static Final Variable
* Public Abstract Method
* Public Static Method
* Public Default Method
* Private default or static Method

Interfaces are defined when there is need of contract for child classes. Each contract is called abstract method.

**Example:**

public interface MyInterface {

public String hello = "Hello";

public void sayHello();

}

A class that implements an interface must implement all the methods declared in the interface. The methods must have the exact same signature (name + parameters) as declared in the interface. The class does not need to implement (declare) the variables of an interface. Only the methods.

It is not possible to create instance of interface. If any instance is created in interface, this instance is of child class.

**Why do we use interface?**

* Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
* It is also used to achieve loose coupling.
* Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

You can use interfaces in Java as a way to achieve polymorphism too.

**Declaring Interfaces**

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

**Example**

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations\

}

**Implementing Interfaces**

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

**Example**

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

public int noOfLegs() {

return 0;

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}

Output

Mammal eats

Mammal travels

**Extending Interfaces**

An interface can extend another interface in the same way that a class can extend another class. The***extends*** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

**Example**

// Filename: Sports.java

public interface Sports {

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Encapsulation

Binding or wrapping the code and data together into a single unit is known asencapsulation. Consider an example of a capsule, which wraps in different medicines to understand the concept better.

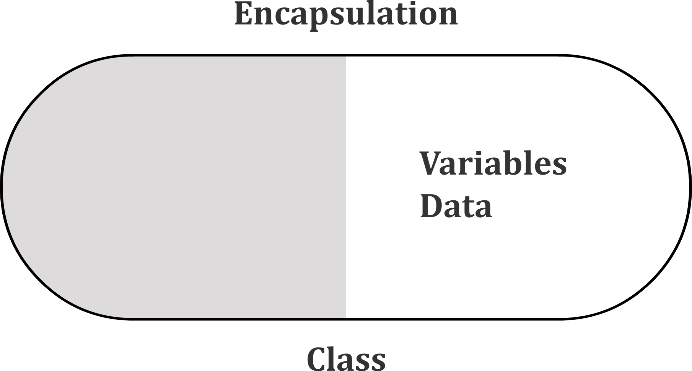


A Java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

When mobile A is connected with mobile B via Bluetooth whereas mobile B is already connected to mobile C then A is not allowed to connect C via B. This is due to encapsulation that implements the desired level of abstraction.

Encapsulation simply means binding object state (fields) and behavior (methods) together. If you are creating a class, you are doing encapsulation.

* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
* Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.



**Advantages of Encapsulation**:

* **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.

**Example**

class Emp

{

private int sal;

public void setSal(int sal)

{

if(sal>0)

this.sal=sal;

}

public int getSal()

{

return sal;

}

}

class Test

{

public static void main(String g[])

{

Emp e=new Emp();

e.setSal(5000);

System.out.println(e.getSal());

}

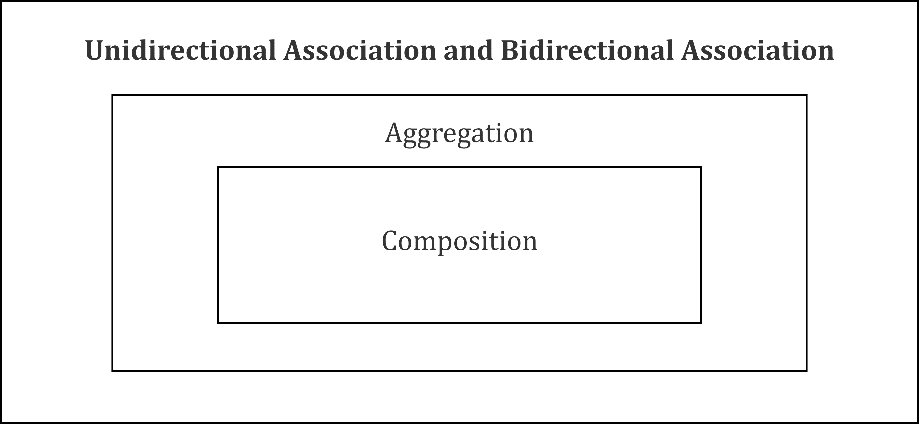
}

## Association with Implementation

Associationis a relationship which describes the reasons for the relationship and the rules that govern the relationship.

Association can be of four types:

1. Unidirectional
2. Bidirectional
3. Aggregation
4. Composition



**Unidirectional Association** is a specialized form of association where one object is associated with another object, but the reverse is not true. It is like one way communication.

Let’s take examples of multiple students can associate with a single teacher.

public class Teacher

{

public string aTeacher = string.Empty;

public Teacher(string teacherName)

{

aTeacher = teacherName;

}

public void Associaton(IList<Student> students)

{

Console.WriteLine("Following are the students");

foreach (var student in students)

{

Console.WriteLine(student.aStudent);

}

Console.WriteLine("associated with a teacher " + aTeacher);

Console.WriteLine();

}

}

**Bidirectional Association** is a type of association where one object is related with other objects and the reverse is also true. It is like two way communication.

Let’s take examples of multiple students can associate with a single teacher and a single student can associate with multiple teachers.

public class Teacher

{

public string aTeacher = string.Empty;

public Teacher(string teacherName)

{

aTeacher = teacherName;

}

public void Associaton(IList<Student> students)

{

Console.WriteLine("Following are the students");

foreach (var student in students)

{

Console.WriteLine(student.aStudent);

}

Console.WriteLine("associated with a teacher " + aTeacher);

Console.WriteLine();

}

}

public class Student

{

public string aStudent = string.Empty;

public Student(string studentName)

{

aStudent = studentName;

}

public void Associaton(IList<Teacher> teachers)

{

Console.WriteLine("Following are the teachers");

foreach (var teacher in teachers)

{

Console.WriteLine(teacher.aTeacher);

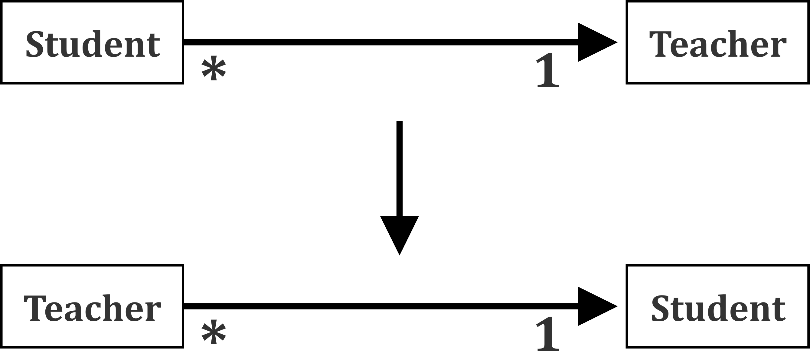
}

Console.WriteLine("associated with a student " + aStudent+".");

Console.WriteLine();

}

}



**Aggregation** is a specialized form of Association where all objects have their own life cycle but there is ownership and child object cannot belong to another parent object.

If object ‘a’ related with object b and relationship rule stated that object ‘a’ is taking a more important role in the relationship or it is the owner of the relationship, then the association is called as aggregation.

Let’s take examples of department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department, teacher object will not destroy.

public class BiologyDempartment

{

private IList<Teacher> teachers;

public BiologyDempartment(IList<Teacher> teachers)

{

this.teachers = teachers;

}

}

public class Teacher

{

string teacherName;

public Teacher(string teacherName)

{

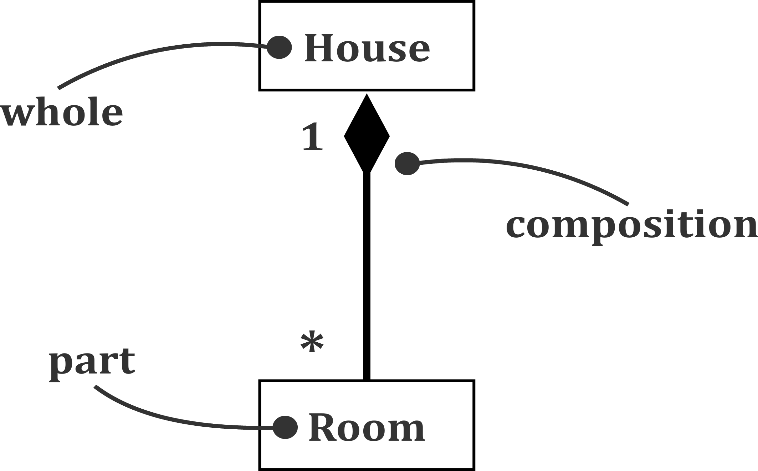
this.teacherName = teacherName;

}

}

**Composition** is again a specialized form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. A child object does not have its life cycle and if parent object deletes, all child objects will also be deleted.

Let’s take again an example of relationships between House and rooms. House can contain multiple rooms. There is no independent life of room and any room cannot belong to two different houses if we delete the house room will automatically delete. Each class referenced is considered to be part-of the aggregate class.



public class House

{

IList<Room> rooms = new List<Room>();

}

public class Room

{

}

Reading Input from Keyboard

In Java, there are three different ways for reading input from the user in the command line environment (console).

1. **Using system.in**

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader.

**Advantages**

* The input is buffered for efficient reading.

**Drawback:**

* The wrapping code is hard to remember.

**Example using system.in**

importjava.io.BufferedReader;

importjava.io.IOException;

importjava.io.InputStreamReader;

publicclassTest

{

    publicstaticvoidmain(String[] args) throwsIOException

    {

        //Enter data using BufferReader

        BufferedReader reader =

                   newBufferedReader(newInputStreamReader(System.in));

        // Reading data using readLine

        String name = reader.readLine();

        // Printing the read line

        System.out.println(name);

    }

}

Input:

Ducat

Output:

Ducat

**Example using scanner class**

|  |
| --- |
| // Java program to demonstrate working of Scanner in Java  importjava.util.Scanner;    classGetInputFromUser  {      publicstaticvoidmain(String args[])      {          // Using Scanner for Getting Input from User          Scanner in = newScanner(System.in);            String s = in.nextLine();          System.out.println("You entered string "+s);            inta = in.nextInt();          System.out.println("You entered integer "+a);            floatb = in.nextFloat();          System.out.println("You entered float "+b);      }  } |

Input:

Ducat

12

3.4

Output:

You entered string Ducat

You entered integer 12

Enter a float

You entered float 3.4

1. **Using Command Line Argument**

The arguments that are passed through the command line are known as command line arguments. The String arguments passed are stored in the array specified in the *main()* declaration. *args[]* is now a three element String array. These elements are accessed in the same way as the elements of a normal array.

**Advantages:**

* Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
* Regular expressions can be used to find tokens.

**Drawback:**

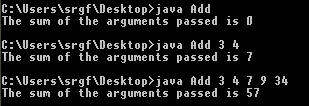
* The reading methods are not synchronized

**Example:**

The following is the complete Add program which is capable of adding any number of integers passed as command line arguments.

public class Add {  
  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int i = 0; i < args.length; i++) {  
            sum = sum + Integer.parseInt(args[i]);  
        }  
        System.out.println("The sum of the arguments passed is " + sum);  
    }  
}

Executions from the command line:



1. **Using Console Class**

The Console class has been introduced with JDK6. It has been becoming a preferred way for reading user’s input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

**Advantages:**

* Reading password without echoing the entered characters.
* Reading methods are synchronized.
* Format string syntax can be used.

**Drawback:**

* Does not work in non-interactive environment (such as in an IDE).

**Example**

// Java program to demonstrate working of System.console()

// Note that this program does not work on IDEs as

// System.console() may require console

publicclassSample

{

    publicstaticvoidmain(String[] args)

    {

        // Using Console to input data from user

        String name = System.console().readLine();

        System.out.println(name);

    }

}

Understanding Packages

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package.

**Built-in Packages**  
These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

1. **java.lang:**Contains language support classes (e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:**Contains classes for supporting input / output operations.  
   **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support; for date / time operations.
3. **java.applet:**Contains classes for creating Applets.
4. **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
5. **java.net:**Contain classes for supporting networking operations.

**User-defined packages**

These are the packages that are defined by the user.

**Advantage of Java Package**

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2. Java package provides access protection.
3. Java package removes naming collision.

**How to use a class of package?**

* **Using fully qualified classname:**

Let’s say we have to create a class to call system date.

class Test{

public static void main(String[] args){

System.out.println(LocalDate.now()); //will give syntax error

}

}

Output

Test.java:5: error: cannot find symbol

System.out.println(LocalDate.now());

^

symbol: variable LocalDate

location: class Test

1 error

**Note**: *LocalDate* class has a static method .*now*(). But this class comes under *java.time* package, which compiler does not know. That why user will get above error.

In order to avoid error occurred in above program, we will have to write *java.time* package before *LocalDate.now*()

class Test{

public static void main(String[] args){

System.out.println(java.time.LocalDate.now());

}

}

Output

2018-06-28

* **Importing the class:**

Let us see the second method to show system date.

import java.time.LocalDate;

class Test{

public static void main(String[] args){

System.out.println(LocalDate.now());

}

}

Output

2018-06-28

* **Importing the all classes and interfaces of a subpackage:**

import java.time.\*;

class Test{

public static void main(String[] args){

System.out.println(LocalDate.now());

}

**}**

Output

2018-06-28

**Creating Userdefind Package**

**Step-1**

Write package statement in java file

package packageName;

**Example**

package india;

class Ducat

{

Public static void main(String[] args)

{

System.out.println(“Hello Ducat”);

}

}

Ducat.java

**Step-2**

Compile java file by using following command

Javac –d . Ducat.java

Here “–d” means package directory creation or updation and . (dot) between –d and Ducat means package will be created in current working directory.

**Step-3**

Run the class by using following command

Java india.Ducat

Output

Hello Ducat

Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

* Visible to the package, the default. No modifiers are needed
* Visible to the class only (private)
* Visible to the world (public)
* Visible to the package and all subclasses (protected)

**Default Access Modifier - No Keyword**

Default access modifier means that you do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

**Example**

Variables and methods can be declared without any modifiers. Note the following example:

String version = "1.5.1";

boolean processOrder() {

return true;

}

**Private Access Modifier - Private**

Methods, variables and constructors that are declared *private* can be accessed within the declared class itself only.

*Private* access modifier is the most restrictive access level. Class and interfaces cannot be declared *private*.

Variables that are declared *private* can be accessed outside the class, if *public* getter methods are present in the class.

With a *private* modifier, an object can encapsulate itself and hide the stored data from the outside world.

**Example**

The following class uses private access control −

public class Logger {

private String format;

public String getFormat() {

return this.format;

}

public void setFormat(String format) {

this.format = format;

}

}

In the above example, the *format* variable of the *Logger* class is declared *private*. Therefore, other classes cannot retrieve or set its value directly.

**Public Access Modifier - Public**

A class, method, constructor, interface can be declared *public* can be accessed from any other class. Therefore, fields, methods, blocks declared inside a *public* class can be accessed from any class belonging to the Java Universe.

However, if the *public* class that you are trying to access is in a different package, then the *public* class still needs to be imported. Because of class inheritance, all *public* methods and variables of a class are inherited by its subclasses.

**Example**

The following function uses public access control −

public static void main(String[] arguments) {

// ...

}

The *main()* method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as Java) to run the class.

**Protected Access Modifier - Protected**

Variables, methods and constructors, which are declared protected in a superclass can be accessed by only the subclasses in other package, or by any class within the package of the protected members' class.

The *protected* access modifier cannot be applied to classes and interfaces. Methods, fields can be declared *protected*, however methods and fields in an interface cannot be declared protected.

*Protected* access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

**Example**

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

class AudioPlayer {

protected boolean openSpeaker(Speaker sp) {

// implementation details

}

}

class StreamingAudioPlayer {

boolean openSpeaker(Speaker sp) {

// implementation details

}

}

Here, if you define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If you define it as public, then it would become accessible to all the outside world. But the intention, here, is to expose this method to its subclass only, therefore you must use the *protected* modifier.

Nested Classes

In Java, just like methods, variables of a class can also have another class as its member. Writing a class within another is allowed in Java. The class written within another class is called the nested class, and the class that holds the inner class is called the outer class.

**Syntax**

Use the following syntax to write a nested class. Here, the class *Outer\_Demo*is the outer class and the class *Inner\_Demo* is the nested class.

class Outer\_Demo {

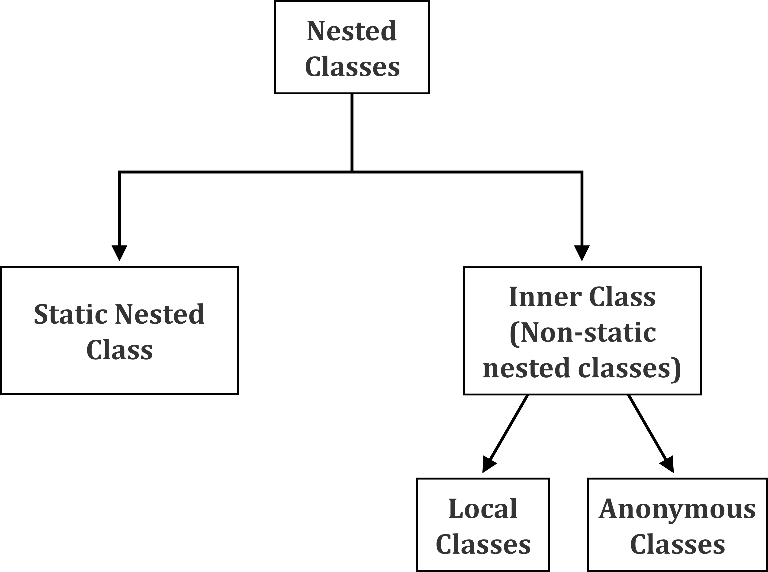
class Nested\_Demo {

}

}

Nested classes can be of two types:

* **Non-static nested classes**: These are the non-static members of a class.
* **Static nested classes**: As the name suggests, these are the static members of a class.



**Static Nested Class**

A static inner class is a nested class that is a static member of the outer class. It can be accessed without instantiating the outer class, by using other static members. Just like the static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is:

**Syntax**

class MyOuter {

static class Nested\_Demo {

}

}

Instantiating a static nested class is a bit different from instantiating an inner class.

**Non-static Nested Classes (Inner Classes)**

Inner classes are a used as a security mechanism in Java. As you know, a class cannot be associated with the access modifier private, however, when a class is a member of other class, then the inner class can be declared as private. And this is also used to access the private members of a class.

Inner classes can be of two types, depending on how and where they are defined. These are:

* Method-local inner class
* Anonymous inner class

**Method-local Inner Class**

In Java, a class within a method is considered to have a local type. Just as the scope of the local variables is restricted within the method, it is limited for the inner class as well.

A method-local inner class can be instantiated only within the method (where the inner class is defined). The following program shows how to use a method-local inner class.

**Example**

public class Outerclass {

// instance method of the outer class

void my\_Method() {

int num = 23;

// method-local inner class

class MethodInner\_Demo {

public void print() {

System.out.println("This is method inner class "+num);

}

} // end of inner class

// Accessing the inner class

MethodInner\_Demo inner = new MethodInner\_Demo();

inner.print();

}

public static void main(String args[]) {

Outerclass outer = new Outerclass();

outer.my\_Method();

}

}

Output

This is method inner class 23

**Anonymous Inner Class**

An inner class, when declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, you have to declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is:

**Syntax**

AnonymousInner an\_inner = new AnonymousInner() {

public void my\_method() {

........

........

}

};

The following program shows how to override the method of a class using anonymous inner class.

**Example**

abstract class AnonymousInner {

public abstract void mymethod();

}

public class Outer\_class {

public static void main(String args[]) {

AnonymousInner inner = new AnonymousInner() {

public void mymethod() {

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

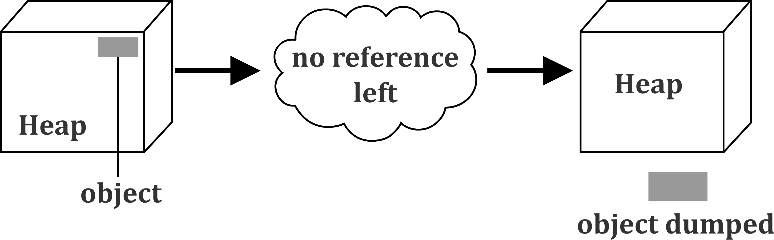
Output

This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class and interface using an anonymous inner class.

Garbage Collection

In Java, an object is destructed from the memory, automatically by the Java Virtual Machine (JVM). When there is no reference to an object, it is assumed to be no longer needed. Hence, the memory occupied by the object is released. This technique is called garbage collection.



The garbage collection cannot be forced explicitly. We may request JVM for executing the garbage collection by calling *System.gc()* method. However, this does not guarantee that JVM will perform the garbage collection.

**Advantages of Garbage Collection**

Note the key advantages of garbage collection process:

* It allows a peace of mind to a programmer. He doesn't need to worry about dereferencing an object.
* It is done automatically by JVM.
* It increases the efficiency of the memory function and decreases the chances for memory leak.

Summary

* Abstraction refers to the concept of hiding internal details and showing the functionality
* Methods in the same class with the same name are called overloaded
* The new operator instantiates a new object by allocating memory for it
* Declarations can appear alone or as a part of object creation
* A class can contain one or more static initializer blocks
* The garbage collection cannot be forced explicitly
* The static keyword is used in java mainly for memory management
* Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype

Chapter 05: Arrays

General Definition of Array

An array is a sequence of elements of the same data type. An array cannot have a mixture of different data types as its elements.

Advantages of an Array

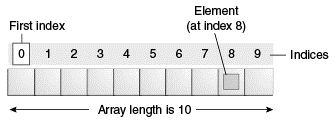
* **Code Optimization:** It makes the code optimized, so you can retrieve or sort the data easily.
* **Randomaccess**: You can get any data located at an index position.

Disadvantage of an Array

* **Contiguous Memory:** Array items are in contiguous memory location. So, we cannot place one item on one location and another item far from it. This is not possible. Array items need to be adjacent and in order. This case may lead to out of memory situation.

Arrays in Java

Array in Java is an object that contains elements of a similar data type. It is a data structure where similar elements are stored. Array in java is index-based, the first element of the array is stored at 0 index.



**Types of Array in Java**

There are two types of array in Java:

* One Dimensional or 1-D Array
* Multidimensional Array

1-D Array

A list of items-group in a single variable name with only one index is called 1-D array.

The general form of a one-dimensional array declaration is:

type var-name[];

Or

type[] var-name;

To create an array, you need to perform three steps:

1. Declare the array
2. Create memory space
3. Initialize the Array Values
4. **Declare the Array**

The general form of a one-dimensional array declaration is:

type var-name[];

OR

type[] var-name;

An array declaration has two components:

1. The *type* and
2. The *name*

The *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like array of int type, we can also create an array of other primitive data types like char, float, double..etc or user-defined data type(objects of a class). Thus, the element type for the array determines what type of data the array will hold.

int intArray[];

or int[] intArray;

Although the above first declaration establishes the fact that intArray is an array variable, no array actually exists**.** It simply tells to the compiler that this(intArray) variable will hold an array of the integer type. To link intArray with an actual, physical array of integers, you must allocate one using *new* and assign it to intArray.

1. **Create Memory Space**

After a declaration of an array, we must allocate a memory space for the declared array. This is done with the help of a new operator. The general form is:

var-name = new type [size];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array. That is, to use *new*to allocate an array, you must specify the type and number of elements to allocate.

**Example**

int intArray[]; //declaring array

intArray = new int[20]; // allocating memory to array

OR

int[] intArray = new int[20]; // combining both statements in one

You should note that you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java, all arrays are dynamically allocated.

**Anonymous Array**

In a situation, where the size of the array and variables of an array are already known, array literals can be used.

int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

// Declaring array literal

* The length of this array determines the length of the created array
* There is no need to write the new int[] part in the latest versions of Java

1. **Initialize the Array Values by using Literals**

We can store values at the time of declaration. The compiler allocates the required space depending upon the list of values. The general form is:

datatype array\_name[] = { list of values};

When we have a small number of literal values that we want to keep in an array, we can initialize it by listing the values between curly braces, separated by a comma. For example, we might use the following code in a program that processes playing cards.

String[] SUITS = {

"Clubs", "Diamonds", "Hearts", "Spades"

};

String[] RANKS = {

"2", "3", "4", "5", "6", "7", "8", "9", "10",

"Jack", "Queen", "King", "Ace"

};

**Example of 1-D Array**

class ArrayBasic

{

public static void main(String g[])

{

int a[]=new int[3];

System.out.println(a);

System.out.println(a.length);//size of array

System.out.println(a[0]);

//System.out.println(a[-1]);//run time error

System.out.println("end of main");

}

}

**Accessing Java Array Elements using for Loop**

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of an array can be accessed using Java for Loop.

**Example**

public class forDemo

{

    public static void main( String[] args )

    {

                for(int i=0;i<5;i++)

        {

            System.out.println( "Value :: " + i );

        }

    }

}

Output

value :: 0

value :: 1

value :: 2  
value :: 3

value :: 4

**Accessing Java Array Elements using for each Loop**

public class ForEachDemo

{

    public static void main( String[] args )

    {

        int[] i =

        { 1, 2, 3, 4, 5 };

        for ( int j : i )

        {

            System.out.println( "value :: " + j );

        }

    }

}

Output

value :: 1

value :: 2  
value :: 3

value :: 4

value :: 5

**Arrays of Objects**

An array in Java can be created either with any data type or with any object like Student objects, Employee objects, Integer, Date etc. Following example on Array of Objects creates Student array and Integer array.

**Example**

class Student

{

int marks;

}

public class ArrayOfObjects

{

public static void main(String args[])

{

Student std[] = new Student[3]; // array of reference variables of Student

std[0] = new Student(); // convert each reference variable into Student object

std[1] = new Student();

std[2] = new Student();

std[0].marks = 40; // assign marks to each Student element

std[1].marks = 50;

std[2].marks = 60;

System.out.println("\n3 students average marks: " + (std[0].marks+std[1].marks+std[2].marks)/3);

}

}

Output

3 students average marks: 50

15 20 25 30 35

In above example, an array of objects is created just like an array of primitive type data items in the following way:

Student std[] = new Student[3];

The array object std[] is not an array of Student objects but an array of Student reference variables. Each variable should be converted separately into an object. That is, here std[0], std[1] and std[2] are Student reference variables. Without conversion, at this stage, if you do std[0].marks = 40, raises error.

std[0] = new Student();

std[1] = new Student();

std[2] = new Student();

Each reference variable is converted into an object. Now std[0], std[1] and std[2] are Student objects. They can be used as objects anywhere.

std[0].marks = 40;  
std[1].marks = 50;  
std[2].marks = 60;

Each Student object is assigned with marks.

Multidimensional Array

The Multidimensional Array in Java programming language is nothing but an Array of Arrays. If the data is linear we can use the One Dimensional Array but to work with multi-level data we have to use Multidimensional Array.

Two-Dimensional Array in Java is the simplest form of Multidimensional Array.

**Declaration of Multidimensional Array in Java**

Following code snippet will show you the declaration of a multidimensional array in Java Programming Language:

Data\_Type[][] Array\_Name;

* **Data\_Type:** This will decide the type of elements it will accept. For example, If we want to store integer values then, the Data Type will be declared as int, If we want to store Float values then, the Data Type will be float etc
* **Array\_Name:**This is the name you want to give it to an array. For example Car, students, age, marks, department, employees etc

Similarly, you can declare the remaining type of arrays:

int [][] an integer array; // declaring an two dimensional array of Integers

byte[][] anByteArray; // declaring an two dimensional array of Bytes

short[][] anShortArray; // declaring an two dimensional array of Shorts

long[][] anLongArray; // declaring an two dimensional array of Longs

float[][] anFloatArray; // declaring an two dimensional array of Floats

double[][] anDoubleArray; // declaring an two dimensional array of Doubles

boolean[][] anBooleanArray; // declaring an two dimensional array of Booleans

char[][] anCharArray; // declaring an two dimensional array of Chars

String[][]anStringArray; // declaring an two dimensional array of Strings

**Create Multidimensional Array in Java**

In order to create a multidimensional array, we have to use the *new* operator as we shown below:

Data\_Type[][]Array\_Name=new int[Size\_of\_2D][Size\_of\_all\_1D];

If you have already initialized an array then

double[][]anStudentArray;// Declaration of Two dimensional array in java

 // Creating an Java two dimensional Array

anStudentArray=newint[5][3];

**Example of Multidimensional Array**

class ArrayBasic

{

public static void main(String g[])

{

int a[][]=new int[4][3];

System.out.println(a);

System.out.println(a.length);

System.out.println(a[0]);

System.out.println(a[0].length);

System.out.println(a[0][0]);

}

}

Jagged Array

A jagged array is an array of arrays such that member arrays can be of different sizes, i.e., we can create 2-D arrays but with a variable number of columns in each row. These type of arrays are also known as jagged arrays.

Following is a Java program to demonstrate the jagged array

// Program to demonstrate 2-D jagged array in Java

class JaggedArray1

{

public static void main(String[] args)

{

int a[][]={{10,9,8,7},{6,5,4},{3,2},{1}};

for(int aa[]:a)

{

for(int e:aa)

{

System.out.print(e);

}

System.out.print("\n");

}

}

}

Operations on Arrays

Various operations can be done on arrays and few of them are discussed here:

**Copying an Array**

1. **Using arraycopy() Method**

The arraycopy() method belongs to the System class. It is used to copy the contents of one array to the other.

|  |  |
| --- | --- |
| Type | Description |
| *public static void* |  |
| Method  *arraycopy(Object src, int srcPos, Object dest, int destPos, int length)* | Copies an array from the source array, beginning at the specified position, to the specified position of the destination array.  length – the number of array elements to be copied. |

**Example:**

publicclassArrayCopying{

   publicstaticvoidmain(String[]args){

       intSource[]={5,6,7,8,9,10};

       intDestination[]=newint[5];

       System.arraycopy(Source,1,Destination,0,5);

       System.out.print(Arrays.toString(Destination));

   }

}

Output:

[6, 7, 8, 9, 10]

1. **Using copyof() and copyofrange() Method**

The Arrays class has two commonly used methods for copying the content of array i.e. copyOf() and copyOfRange() method. The Generic type T[] can be replaced with any other primitive type.

|  |  |
| --- | --- |
| Type | Description |
| *public static T[]*  *public static T[]* |  |
| Method  *copyOf(T[] original, int newLength)*  *copyOfRange(T[] original, int from, int to)* | Returns a new array which is a copy of the array specified and padded with 0s to obtain the specified length. newLength is the number of array elements to be copied.  Returns a new array containing the specified range from the original array, truncated or padded with nulls to obtain the required length. |

Example:

publicclassArrayCopying{

   publicstaticvoidmain(String[]args){

       intSource[]={5,6,7,8,9,10};

       intDestination[]=java.util.Arrays.copyOf(Source,8);//0 is padded for index beyond source

       System.out.println(Arrays.toString(Destination));

       intDestination2[]=java.util.Arrays.copyOfRange(Source,0,5);

       System.out.println(Arrays.toString(Destination2));

   }

}

**Manipulation of Array**

The java.util.Arrays class provides several methods for manipulating arrays (copying, sorting, searching etc.)

**Commonly Used Methods of the Class ‘Arrays’:**

|  |  |
| --- | --- |
| Methods | Description |
| *static int binarySearch(int[] a, int key)* | Searches the specified array for the given value using the binary search algorithm and returns the value. |
| *static int binarySearch(int[] a, int fromIndex, int toIndex, int key)* | Searches between the ranges specified in the specified array for the given value using the binary search algorithm and return the value. |
| *static boolean equals(int[] a, int[] a2)* | Returns true if the given arrays are equal. |
| *static void fill(int[] a, int val)* | Assigns the value ‘val’ to each element in the array specified. |
| *static int hashCode(int[] a)* | Returns a hashcode of the specified array. |
| *static void parallelSort(int[] a)* | Sorts the specified array into ascending order. This works well with an array with a huge number of elements. |
| *static void sort(int[] a)* | Sorts the specified array into ascending order. This is a serial sorting method and works well with small to large arrays. |

Summary

* An array is a better and convenient way to store the data of same data type with the same size
* A list of items-group in a single variable name with only one index is called 1-D array
* Networks share resources, exchange files and electronic communications
* The Arrays class has two commonly used methods for copying the content of array i.e. copyOf() and copyOfRange() method
* Array in Java is an object that contains elements of similar data type
* Array makes the code optimized, so you can retrieve or sort the data easily
* Array in java is index-based, the first element of the array is stored at 0 index

Chapter 06 – Command Line Arguments and Wrapper Classes

Command Line Argument

Any input value passed through the command prompt at the time of running of the program is known as command line argument. By default, the command line arguments are treated as string values and these values are stored in a string array of *main ()* method.

**Syntax to Compile and Run CMD programs**

Compile By -> Javac Mainclass.java

Run By -> Java Mainclass value1 value2 value3 ...........

Here value1, value2 and value3 are command line arguments.

**Example of command-line argument in java**

class SumDemo

{

public static void main(String args[])

{

System.out.println("Sum: "+args[0]);

}

}

**Compile and run above program**

Compile By > Javac SumDemo.java

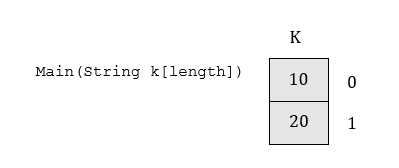
Run By > Java SumDemo 10 20

Output

Sum: 30

During the execution of this program, following sequence of steps takes place:

* Class loader sub-system loads SumDemo in main memory along with Command line argument (10, 20).
* The JVM takes the loaded class SumDemo along with Command line arguments (10, 20) and places the number of values in the length variable i.e. 2.
* The JVM looks for *main()* and places the Command in the *main()* in the form of string class as below:



* Hence all the command line arguments of Java send to main() method in the form of an array of object of String class.
* JVM calls the main() method with respect to load class SumDemo that is SumDemo.main().

Note that, except + operator, no other numeric operator is allowed in command line arguments.

Wrapper Classes in Java

A wrapper class is a class whose object wraps or contains a primitive data type. When an object of a wrapper class is created, it contains a field. In this field, primitive data types can be stored. In other words, a primitive value can be wrapped into a wrapper class object.

**The need for Wrapper Classes**

* Wrapper classes convert primitive data types into objects. Objects are required to modify the arguments passed into a method since primitive types are passed by value.
* The classes in *java.util* package handles only objects and hence Wrapper classes are helpful in this case also.
* Data structures in the Collection framework, such as *ArrayList* and Vector, store only objects (reference types) and not the primitive types.
* An object is needed to support synchronization in multithreading.

In Java, the eight classes of *java. Lang* package are known as the wrapper classes. The list of eight wrapper classes is given below:

**Primitive Data types and their Corresponding Wrapper Class**

|  |  |
| --- | --- |
| Primitive Data Type | Wrapper Class |
| char | Character |
| byte | Byte |
| short | Short |
| Long | Integer |
| Float | Float |
| double | Double |
| boolean | Boolean |

**Autoboxing and Unboxing**

**Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example, conversion of int to Integer, long to Long, double to Double etc.

**Example**

// Java program to demonstrate Autoboxing

import java.util.ArrayList;

classAutoboxing

{

    publicstaticvoidmain(String[] args)

    {

        charch = 'a';;

        // Autoboxing- primitive to Character object conversion

        Character a = ch;

        ArrayList<Integer> arrayList = newArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects

        arrayList.add(25);

        // printing the values from object

        System.out.println(arrayList.get(0));

    }

}

Output

25

**Unboxing:** It is the reverse process of autoboxing. Converting an object of a wrapper class automatically to its corresponding primitive type is known as unboxing. For example conversion of Integer to int, Long to long, Double to double etc.

// Java program to demonstrate Unboxing

import java.util.ArrayList;

class Unboxing

{

    public static void main(String[] args)

    {

        Character ch = 'a';

        // unboxing - Character object to primitive conversion

        chara = ch;

        ArrayList<Integer> arrayList = newArrayList<Integer>();

        arrayList.add(24);

        // unboxing because get method returns an Integer object

        intnum = arrayList.get(0);

        // printing the values from primitive data types

        System.out.println(num);

    }

}

Output

24

Parsing of Numeric Strings

While operating upon strings, there are times when we need to convert a number represented as a string into an integer type. The method generally used to convert String to Integer in Java is *parseInt()*.

**How to use *parseInt()* method in Java?**

There are two variants of this method:

(1) public static int parseInt(String s) throws NumberFormatException

- This function parses the string argument as a signed decimal integer.

(2) public static int parseInt(String s, int radix) throws NumberFormatException

- This function parses the string argument as a signed integer in the radix specified by the second argument.

**Example**

public classGFG

{

    public static void main(String args[])

    {

        intdecimalExample = Integer.parseInt("20");

        intsignedPositiveExample = Integer.parseInt("+20");

        intsignedNegativeExample = Integer.parseInt("-20");

        System.out.println(decimalExample);

        System.out.println(signedPositiveExample);

        System.out.println(signedNegativeExample);

    }

}

Output

20

20

-20

String Representation of Primitives

String representation of primitives can be performed using Integer.toString(int) or String.valueOf(int):

1. **Using Integer.toString(int)**

The Integer class has a static method that returns a String object representing the specified *int* parameter.

**Syntax:**

public static String toString(int i)

The argument *i* is converted and returned as a string instance. If the number is negative, the sign will be preserved.

**Example**

classGfG

{

  publicstaticvoidmain(String args[])

  {

    inta = 1234;

    intb = -1234;

    String str1 = Integer.toString(a);

    String str2 = Integer.toString(b);

    System.out.println("String str1 = "+ str1);

    System.out.println("String str2 = "+ str2);

   }

}

Output

String str1 = 1234

String str2 = -1234

1. **Using String.valueOf(int)**

**Example**

classGfG

{

  publicstaticvoidmain(String args[])

  {

    intc = 1234;

    String str3 = String.valueOf(c);

    System.out.println("String str3 = "+ str3);

  }

}

Output

String str3 = 1234

Summary

* Any input value passed through the command prompt at runtime is known as command line argument.
* A Wrapper class is a class whose object wraps or contains a primitive data types.
* Automatic conversion of primitive types to the object of their corresponding Wrapper classes is known as autoboxing.
* Unboxing is the exact reverse process of autoboxing.
* The method generally used to convert String to Integer in Java is *parseInt()*.
* String representation of primitives can be performed using Integer.toString(int) or String.valueOf(int).

# Chapter 07 – Exception Handling

## Runtime Errors

Runtime errors are the unwanted conditions or events that occur when you try to run your program, and occurrence of such condition terminates program abnormally. Runtime errors can occur in three following situations:

* When the Java Virtual Machine steps in and shows that you've done something wrong. This happens in case of syntax errors where program crashes and execution gets interrupted. Such "program crashes" are often annoying, but ultimately for the best.
* When the JVM is unable to print an error message and your program just quits working completely. With such "hanging programs", it's obvious that something is wrong, but, other than the symptoms, you do not receive any help in solving the error.
* The last situation of errors is really problematic because such errors occur silently and without warning. These are the logical errors, which cause your program to behave incorrectly, but, unless you're on the lookout, you might not even notice that anything is wrong.

## Types of Runtime Errors

Runtime errors can be of two types:

1. Java exceptions
2. Error

**Java Exceptions**

Exceptions are those runtime errors that can be recovered in the execution of program. Java code written by the user may contain unintentional errors like computational errors (division by zero, or the square root of a negative value) or accessing null pointer and so on. These errors are detected by Java runtime environment. If such an error occurs, Java throws an exception.

**Error**

Errors are those runtime errors of Java program execution that cannot be recovered.   
For example  
  
boolean valid = true;

while(valid) {

System.out.println("Hi");

}

**What is exception handling?**

Exception Handling is a mechanism to handle runtime errors so that we can avoid abnormal program termination.

**Advantage of Exception Handling**

The core advantage of exception handling is to maintain the normal flow of the application. Exception normally disrupts the normal flow of the application that is why exception handling is used. Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

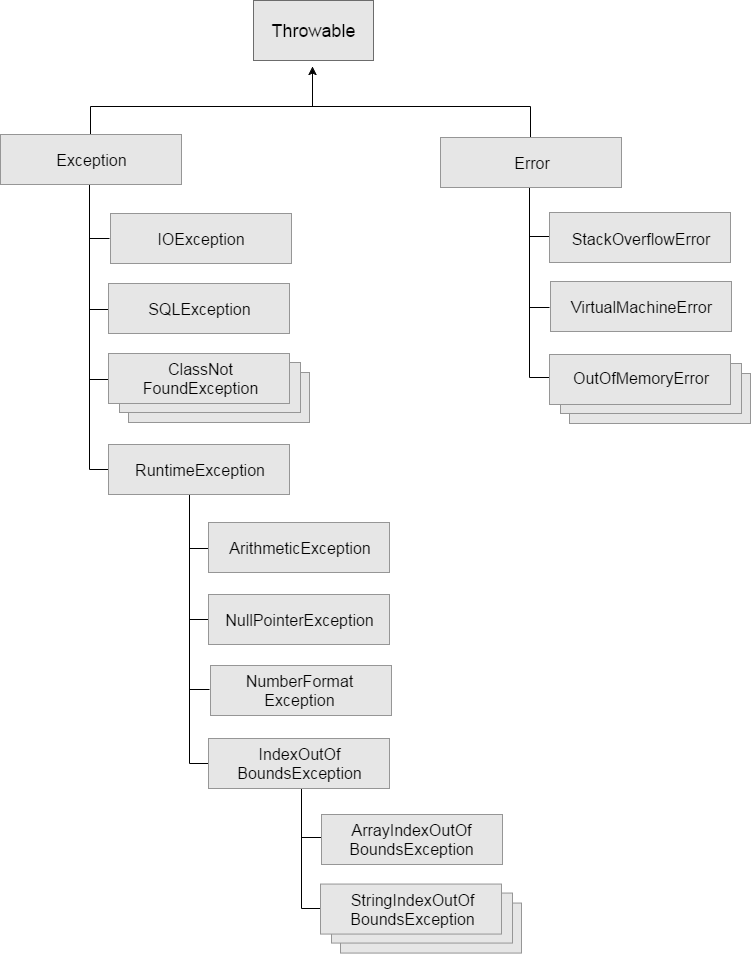
statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in your program and an exception occurs at statement 5. Usually in this case, rest of the statements will not be executed i.e. statement 6 to 10 will not run. But if exception handling is performed here, rest of the statementswill also be executed. This is why exception handling is used in java.

**Hierarchy of Java Exception classes**



All exceptions and error types are subclasses of class ***Throwable***, which is base class of hierarchy.One branch is headed by ***Exception***. This class is used for exceptional conditions that the programs should catch. *NullPointerException* is an example of such an exception. Another branch, ***Error*** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). *StackOverflowError* is an example of such an error.

## Types of Exception

According to the sun microsystem, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

**Checked Exception**

The classes that extend *Throwable* class except *RuntimeException* and *Error* are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

**Unchecked Exception**

The classes that extend *RuntimeException* are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are checked at runtime rather than compile-time.

**Error**

Error is an exception that is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Exception Handler

Once runtime receives the exception object, it tries to find the handler for the exception. Exception Handler is a block of code that can process the exception object. The handler is said to be “catching the exception”. The logic to find the exception handler is to simply start the search in the method where the error occurred. If the appropriate exception handler is found, the exception object is passed to the handler to process it. If there no appropriate exception handler is found, the program terminates by printing information about the exception.

## Java Exception Handling Keywords

Java provides specific keywords to create an exception handler block.

1. **try-catch**
2. **finally**
3. **throw**
4. **throws**

## Try and Catch

**Try block**

The *try* block contains a logically related set of statements where an exception can occur. A *try* block must be followed by *catch* block or *finally* block or both.

**The syntax of try block**

try{

//statements that may cause an exception

}

While writing a program, if you think that certain statements in a program can throw an exception, then enclose them in *try* block and handle that exception.

**Catch block**

A catch block is used where you need to handle the exceptions. This block must follow the *try* block. A single *try* block can have several *catch* blocks associated with it. You can catch different exceptions in different *catch* blocks. When an exception occurs in *try* block, the corresponding *catch* block, which handles the exception, executes. For example, if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

**The syntax of try catch in java**

try

{

//statements that may cause an exception

}

catch (exception\_type e)‏

{

//error handling code

}

## Multiple Catch Clauses

You can specify two or more *catchclauses*, each catching a different type of exception.

When an exception is thrown, each *catch* statement is inspected in an order and the first one whose type matches is executed.

After one catch statement executes, the others are bypassed, and execution continues after the try-catch block.

public class Main {

public static void main(String args[]) {

try {

int a = args.length;

System.out.println("a = " + a);

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch (ArithmeticException e) {

System.out.println("Divide by 0: " + e);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

In case of multiple catch statements, exception subclasses must come before any of their superclasses.

## Polymorphic Behavior of *catch()* Block

If a catch block handle multiple type of exceptions and contains argument of exception class.

**Syntax:**

try{

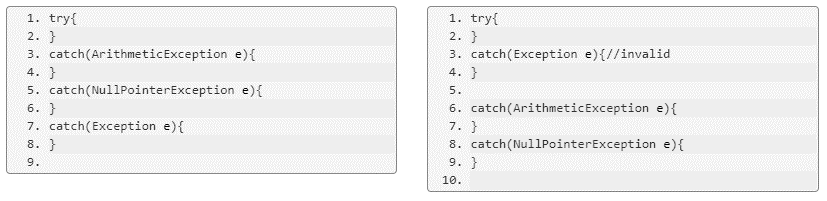
}

catch(Exception e)

{

}

Note: We can use both specific and polymorphic catch block for a single try block but in this case the polymorphic catch must be the last handler for try block.



**Piping in catch**

Since JDK7 onwards, catch statement started using piping between exceptions.

**Syntax**

public static void main(String[] args) {

try

{

}

catch(FirstException | SecondException | ThirdException e){

}

## 

## Nested Try Statements

When a *try-catch* block appears in another *try* block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception. If a match is found then that catch block executes.

If neither catch block nor parent catch block handles exception then the system generatesa message for the exception, similar to what you see when you don't handle the exception.

Let’s see the syntax first then discuss this with an example.

**The syntax of Nested Try Statement**

//Main try block

try {

statement 1;

statement 2;

//try-catch block inside another try block

try {

statement 3;

statement 4;

//try-catch block inside nested try block

try {

statement 5;

statement 6;

}

catch(Exception e2) {

//Exception Message

}

}

catch(Exception e1) {

//Exception Message

}

}

//Catch of Main(parent) try block

catch(Exception e3) {

//Exception Message

}

## Finally, throw and throws

**Finally**

The *finally* block contains all the crucial statements that must be executed whether an exception occurs or not. The statements present in this block always execute regardless of whether an exception occurs in try block or not, such as closing a connection, stream etc.

**The syntax of finally block**

try {

//Statements that may cause an exception

}

catch {

//Handling exception

}

finally {

//Statements to be executed

}

**Example of finally block**

Here you can see that the exception occurred in try block which has been handled in catch block after that finally block got executed.

class Example

{

public static void main(String args[]) {

try{

int num=121/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("Number should not be divided by zero");

}

/\* Finally block will always execute

\* even if there is no exception in try block

\*/

finally{

System.out.println("This is finally block");

}

System.out.println("Out of try-catch-finally");

}

}

Output

Number should not be divided by zero

This is finally block

Out of try-catch-finally

The *finally* block is always executed after the code in the preceding *try* block. It doesn’t matter if the *try* block is used to throw an exceptionor not.

To use a try-finally clause:

* Enclose in a try block the code that has multiple exit points, and
* Put in a finally block the code that must happen no matter how the try block is exited.

try {

//Statements that may cause an exception

} finally {

//Statements to be executed

}

statement

In this case no exception is thrown.

**Example:**

static int giveMeThatOldFashionedBoolean(boolean bVal) {

try {

if (bVal) {

return 1;

}

return 0;

}

finally {

System.out.println("Got old fashioned.");

}

}

**Throw**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. You can throw either checked or unchecked exception. The *throw* keyword is mainly used to throw custom exceptions.

**Syntax:**

throw ThrowableInstance

**Example:**

throw new ArithmeticException("/ by zero");

But this exception must be of type *Throwable* or a subclass of *Throwable*. For example, *Exception* is a sub-class of *Throwable* and user-defined exceptions typically extend *Exception* class. Unlike C++, in Java data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement matching the type of exception. If it finds a match, control is transferred to that statement else next enclosing try block is checked and so on. If no matching catch is found then the default exception handler stops the program.

|  |
| --- |
| // Java program that demonstrates the use of throw  class ThrowExcep  {      static void fun()      {          try          {              throw new NullPointerException("demo");          }          catch(NullPointerException e)          {              System.out.println("Caught inside fun().");              throw e; // rethrowing the exception          }      }        public static void main(String args[])      {          try          {              fun();          }          catch(NullPointerException e)          {              System.out.println("Caught in main.");          }      }  } |

Output

Caught inside fun().

Caught in main.

**Throws**

*throws* is a keyword in Java which is used in the signature of the method to indicate that the method can throw one of the listed type exceptions. The caller of these methods must handle the exception using a try-catch block.  
Syntax:

typemethod\_name(parameters)

It throws *exception\_list*which is a comma-separated list of all the exceptions.

In a program, if there is a chance of rising an exception then compiler always warns us about it and you should compulsorily handle that checked exception. Otherwise, you will get compile time error saying unreported exception XYZ must be caught or declared to be thrown. To prevent this compile time error, you can handle the exception in two ways:

* By using try catch
* By using throws keyword

You can use *throws* keyword to delegate the responsibility of exception handling to the caller (a method or JVM) where caller method is responsible to handle that exception.

|  |
| --- |
| // Java program to illustrate error in case  // of unhandled exception  class tst  {      public static void main(String[] args)      {          Thread.sleep(10000);          System.out.println("Hello Geeks");      }  } |

Output

error: unreported exception InterruptedException; must be caught or declared to be thrown

## Creating Custom Exceptions

Java provides facility to create your own exceptions which are basically derived classes of Exception. For example, MyException in below code extends the Exception class.

|  |
| --- |
| // A Class that represents a user-defined exception  classMyException extendsException  {      publicMyException(String s)      {          // Call constructor of parent Exception          super(s);      }  }    // A Class that uses above MyException  publicclassMain  {      // Driver Program      publicstaticvoidmain(String args[])      {          try          {              // Throw an object of user defined exception              thrownewMyException("Java");          }          catch(MyException ex)          {              System.out.println("Caught");               // Print the message from MyException object              System.out.println(ex.getMessage());          }      }  } |

Output

Caught

Java

**Notes:**  
1. User-defined exception must extend *Exception* class.  
2. The exception is thrown using *throw* keyword.

## Assertion

An assertion allows testing the correctness of any assumptions that have been made in the program.

An assertion is achieved using the assert statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named *AssertionError*. It is mainly used for testing purposes during development.

The assert statement is used with a Boolean expression and can be written in two different ways.

First way:

* assert expression;

Second way:

* assert expression1 : expression2;

Example of Assertion:-

|  |
| --- |
| // Java program to demonstrate syntax of assertion  import java.util.Scanner;   class Test  {      public static void main( String args[] )      {          int value = 15;          assert value >= 20 : " Underweight";          System.out.println("value is "+value);      }  } |

Run on IDE

Output:

value is 15

After enabling assertions

Output:

Exception in thread "main" java.lang.AssertionError: Underweight

## Summary

* When the JVM is unable to print an error message and your program just quits working completely
* The handler is said to be **“catching the exception”**
* *throws* is a keyword in Java which is used in the signature of the method to indicate that the method can throw one of the listed type exceptions
* The classes that extend *Throwable* class except *RuntimeException* and *Error* are known as checked exceptions
* Java provides facility to create your own exceptions which are basically derived classes of Exception
* Exception Handling is a mechanism to handle runtime errors such as ClassNotFound
* When a *try-catch* block appears in another *try* block then it is called the nested try catch block

Chapter 08 – Working with Strings and Various Operations

String: An Overview

A string is an object that represents a sequence of immutable characters It differs from a string in C or C++, where it is simply an array of characters.

String Class

String class is a final class which is found in java.lang package. The java.lang.String class provides a lot of methods to work on string. Java string is not a primitive data type like int and long, rather it is an object that represents immutable sequence of char values. It is more like an array of characters.

Creating a String Object

String object can be created in a number of ways. Some of the frequently used ways are:

* **By using a string literal:** A string literal is a simple string enclosed in double quotes " ". It is treated as a string object.

String str1 = "Hello";

* **By using another string object:**

String str2 = new String(str1);

* **By using the *new* keyword:**

String str3 = new String("Java");

* **By using *+* operator (concatenation):**

String str4 = str1 + str2;

Or

String str5 = "hello"+"Java";

Operations on String

String operations are mostly useful for automation testing practice. It can be used to manipulate a string in several ways to meet the specific requirements. In automation testing, there are many situations where strings are required for manipulation.

**Concatenating String**

String concatenation forms a new string *that is* the combination of multiple strings. In Java, two or more string objects can be concatenated in the following two ways:

* **Using + operator:**

class StringConcatenation1{

public static void main(String args[]){

String s="Ducat"+"Java";

System.out.println(s);//DucatJava

}

}

* **Using *concat()* method:**

class StringConcatenation2{

public static void main(String args[]){

String s1=new String(“Ducat”);

String s2=new String(“Java”);

String s = s1.concat(s2);

System.out.println(s);//DucatJava

}

}

* **CharAt:** This method helps us to find the character at the given index of the String. The index starts from 0. This means that the first character of the string is at the 0th index. The last character is at the *length()* – 1index.

**Syntax:**public char charAt(int index)

**Example:**

Public class testCharAt{

Public static void main( Stringargs[] ){

String str = “String operation in Java.”;

Char start = str.charAt(0);

Char last = str.charAt(str.length() - 1);

Char random = str.charAt(10);

System.out.println (start); // will print ‘S’

System.out.println (last);// will print ‘.’

System.out.println (random);// will print ‘r’

}

}

* **compareTo:** This method is used to compare the two strings lexicographically. It returns *int* value in form of the comparison result. It returns 0, when both the strings are equal and less than 0, when the argument string is greater than this string. It will return a value greater than 0, when the argument string is less than this string.

class StringcompareTo{

public static void main(String args[]){

String s1 = "Sachin";

String s2 = "Sachin";

String s3 = "Tendulkar";

System.out.println(s1.compareTo(s2)); //0

System.out.println(s1.compareTo(s3)); //-1(because s1 < s3)

System.out.println(s3.compareTo(s1)); //1(because s3 > s1 )

}

}

The *compareTo* method can also be used to compare any two objects as it is a part of the Java object class. So, it is not limited to only the String comparisons.

* **ContentEquals:** This method compares String with StringBuffer and returns Boolean value.

public class StringContentEquals {

public static void main(String args[]) {

String str1 = "One";

String str2 = "Two";

StringBuffer str3 = new StringBuffer( "One");

StringBuffer str4 = new StringBuffer( "Two");

System.out.println(str1.contentEquals(str3)); //true

System.out.println(str2.contentEquals(str3)); //false

System.out.println(str1.contentEquals(str4)); //false

System.out.println(str2.contentEquals(str4)); //true

}

}

* **Equals and equalsIgnoreCase:** The *equals( )* method is used to compare two strings for equality. It is written as:

boolean equals(Object str)

Here, *str* is the string object being compared to the invoking string object. It returns *true,* when the strings contain the same characters in the same order. It returns *false*, when the strings do not contain the same characters. Note that this comparison is case-sensitive.

To perform a comparison that ignores case differences, you can call *equalsIgnoreCase()* method*.* This method considers alphabets written in uppercase (A-Z) to be the same as the ones written in lowercase (a-z). It has the following general form:

boolean equalsIgnoreCase(String str)

Here, *str* is the string object which is being compared to the invoking string object. It also returns *true,* when the strings contain the same characters in the same order, and *false* when they don’t return the same characters.

Below is an example that demonstrates the use of *equals( )* and *equalsIgnoreCase( )*:

class equalsDemo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good-bye";

String s4 = "HELLO";

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));

System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));

System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));

}

}

Output

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

* **getChars:** The *getChars( )*method is used if you need to extract more than one character at a time. It has following general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Below is an example of *getChars()*:

class getCharsDemo {

public static void main(String args[]) {

String s = "This is a demo of the getChars method.";

int start = 10;

int end = 14;

char buf[] = new char[end - start];

s.getChars(start, end, buf, 0);

System.out.println(buf);

}

}

Output

demo

* **replace:**The *replace( )*method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example:

String s = "Hello".replace('l', 'w');

With this command, the program puts the string “Hewwo” into s.

* **Split**:You can split a string using this method. String is broken around the given delimiter (regular expression). It returns an array of Strings. Let’s understand it with an example:

class StringSplit {

public static void main(String args[]) {

String s = new String("Demo:of:the:Split:method.");

String buf[] = s.split(“:”);

System.out.println(buf);

}

}

Output

Demo

of

the

Split

method.

* **Substring:** You can extract a substring using *substring( )*. It can be written two ways:

The first form is

String substring(int startIndex)

Here, *startIndex*specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex*and runs to the end of the invoking string.

The second form of *substring( )*allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

Here, *startIndex*specifies the beginning index, and *endIndex*specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

**Example for substring:**

class StringReplace {

public static void main(String args[]) {

String org = "This is a test. This is, too.";

String search = "is";

String sub = "was";

String result = "";

int i;

do { // replace all matching substrings

System.out.println(org);

i = org.indexOf(search);

if(i != -1) {

result = org.substring(0, i);

result = result + sub;

result = result + org.substring(i + search.length());

org = result;

}

} while(i != -1);

}

}

Output

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

* **toUppercase and toLowerCase:** These two methods are used to convert the questioned string to all uppercase letters and lowercase letters, respectively.

**Example of toUppercase**

class StringtoUppercase {

public static void main(String args[]){

String s1 = new String(“DucatJava”);

s1 = s1. toUppercase();

System.out.println(s1);//DUCATJAVA

}

}

**Example of toLowercase**

class StringtoLowercase {

public static void main(String args[]){

String s1 = new String(“DucatJava”);

s1 = s1.toLowercase();

System.out.println(s1);//ducatjava

}

}

Character Extraction in Java

A string is treated as an object in Java, so we can’t directly access the characters that comprise a string. However, there are several ways in Java to extract characters from a string class object. These predefined methods are:

* **charAt():** charAt() method is used to extract a single character at an index. It has the following syntax.

**Syntax**:

char charAt(int index)

**Example of charAt**

class temp

{

    public static void main(String...s)

    {

        String str="Hello";

        char ch=str.charAt(2);

        System.out.println(ch);

    }

}

**Output**

I

In above example *ch* will contain character l. We must take care that the index should not be negative and should not exceed string length.

* **getChars():** It is used to extract more than one character. *getChars()* has following syntax.

void getChars(int stringStart, int stringEnd, char arr[], int arrStart)

Here, *stringStart* and *stringEnd* is the starting and ending index of the substring. *arr* is the character array that will contain the substring. It will contain the characters starting from *stringStart* to *stringEnd-1*. *arrStart* is the index inside *arr* where the substring will be copied. The *arr* array should be large enough to store the substring.

**Example**

class temp

{

public static void main(String...s)

{

String str="Hello World";

char ch[]=new char[4];

str.getChars(1,5,ch,0);

System.out.println(ch);

}

}

Output

ello

* **getBytes():** The *getBytes()* extracts characters from a string object and then converts them in a byte array. It has the following syntax.

**Syntax**

byte [] getBytes()

**Example of getBytes**

String str="Hello";

byte b[]=str.getBytes();

* **toCharArray():** It is used as an alternative to *getChars()* method. The *toCharArray()* converts all the characters in a string object into an array of characters. It is the best and easiest way to convert a string into a character array. It has the following syntax:

**Syntax:**

char [] toCharArray()

**Example of toCharArray**

class temp

{

public static void main(String...s)

{

String str="Hello World";

char ch[]=str.toCharArray();

System.out.println(ch);

}

}

Output

Hello World

String Comparison

In Java, we can compare strings on the basis of their content and reference. To compare strings for the purpose of:

* Authentication *by equals()* method is used
* Sorting *by compareTo()* method is used
* Reference matching *by == operator* is used

Let’s review each of these methods in detail.

**String compare by equals() method**

The string *equals()* method compares the original content of the string. It compares values of string for equality. String class provides two methods:

* *public boolean equals*(Object another) compares this string to the specified object
* *public boolean equalsIgnoreCase*(String another) compares this String to another string, ignoring case

**Example of by equals() method**

class Teststringcomparison1{

 public static void main(String args[]){

   String s1="Sachin";

   String s2="Sachin";

   String s3=new String("Sachin");

   String s4="Saurav";

   System.out.println(s1.equals(s2));//true

   System.out.println(s1.equals(s3));//true

   System.out.println(s1.equals(s4));//false

 }

}

Output

true

true

false

**String compare by == operator**

The = = operator compares references not values. We can understand it with an example:

class Teststringcomparison3{

 public static void main(String args[]){

   String s1="Sachin";

   String s2="Sachin";

   String s3=new String("Sachin");

   System.out.println(s1==s2);//true (because both refer to same instance)

   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)

 }

}

Output

true

false

**String compare by compareTo() method**

The string *compareTo()* method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables.

s1 == s2 :0

s1 > s2   :positive value

s1 < s2   :negative value

class Teststringcomparison4{

 public static void main(String args[]){

   String s1="Sachin";

   String s2="Sachin";

   String s3="Ratan";

   System.out.println(s1.compareTo(s2));//0

   System.out.println(s1.compareTo(s3));//1(because s1>s3)

   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

 }

}

Output

0

1

-1

StringBuffer Class and Methods

Java *StringBuffer* class is used to create mutable (modifiable) string. The *StringBuffer* class in java is same as the *String* class except that it is mutable i.e. it can be changed.

Important Constructors of *StringBuffer* class are:

|  |  |
| --- | --- |
| Constructor | Description |
| *StringBuffer()* | Creates an empty string buffer with the initial capacity of 16 |
| *StringBuffer(String str)* | Creates a string buffer with the specified string |
| *StringBuffer(int capacity)* | Creates an empty string buffer with the specified capacity as length |

Important methods of StringBuffer class are:

|  |  |  |
| --- | --- | --- |
| Modifier and Type | Method | Description |
| *public synchronized StringBuffer* | *append(String s)* | It is used to append the specified string with this string. The *append()* method is overloaded like append(char), append(boolean), append(int), append(float), append(double). |
| *public synchronized StringBuffer* | *insert(int offset, String s)* | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| *public synchronized StringBuffer* | *replace(int startIndex, int endIndex, String str)* | It is used to replace the string from specified *startIndex* and *endIndex*. |
| *public synchronized StringBuffer* | *delete(int startIndex, int endIndex)* | It is used to delete the string from specified startIndex and endIndex. |
| *public synchronized StringBuffer* | *reverse()* | It is used to reverse the string. |
| *public int* | *capacity()* | It is used to return the current capacity. |
| *public void* | *ensureCapacity(int minimumCapacity)* | It is used to ensure the capacity at least equal to the given minimum. |
| *public char* | *charAt(int index)* | It is used to return the character at the specified position. |
| *public int* | *length()* | It is used to return the length of the string i.e. total number of characters. |
| *public String* | *substring(int beginIndex)* | It is used to return the substring from the specified *beginIndex*. |
| *public String* | *substring(int beginIndex, int endIndex)* | It is used to return the substring from the specified *beginIndex* and *endInde.* |

Now let us review some of the important methods.

**StringBuffer append() Method**

The *append()* method concatenates the given argument with this string.

class StringBufferExample{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**StringBuffer insert() Method**

The *insert()* method inserts the given string with this string at the given position.

class StringBufferExample2{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

**StringBuffer replace() Method**

The replace() method replaces the given string from the specified beginIndex and endIndex.

class StringBufferExample3{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

**StringBuffer delete() Method**

The *delete()* method of *StringBuffer* class deletes the string from the specified beginIndex to endIndex.

class StringBufferExample4{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

**StringBuffer reverse() Method**

The *reverse()* method of *StringBuffer* class reverses the current string.

class StringBufferExample5{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**StringBuffer capacity() Method**

The *capacity()* method of *StringBuffer* class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example, if your current capacity is 16, it will be (16\*2)+2=34.

class StringBufferExample6{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

**StringBuffer ensureCapacity() Method**

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBufferExample7{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |  |
| --- | --- | --- |
| No. | String | StringBuffer |
| 1) | String class is immutable. | *StringBuffer* class is mutable. |
| 2) | String is slow and consumes extra memory when you concatenate too many strings as every time it creates new instance. | *StringBuffer* is fast and consumes less memory when you concatenate strings. |
| 3) | String class overrides the *equals()* method of Object class. So you can compare the contents of two strings by *equals()* method. | *StringBuffer* class doesn't override the *equals()* method of Object class. |

String Builder Class

Java *StringBuilder* class is used to create mutable (modifiable) string. The Java *StringBuilder* class is same as *StringBuffer* class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

|  |  |
| --- | --- |
| Constructor | Description |
| *StringBuilder()* | It creates an empty string Builder with the initial capacity of 16 (i.e. 16 empty elements). |
| *StringBuilder(String str)* | It creates a string Builder with the specified string. |
| *StringBuilder(int length)* | It creates an empty string Builder with the specified capacity as length. |

Important methods of StringBuilder class

|  |  |
| --- | --- |
| Method | Description |
| *public StringBuilder append(String s)* | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| *public StringBuilder insert(int offset, String s)* | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| *public StringBuilder replace(int startIndex, int endIndex, String str)* | It is used to replace the string from specified *startIndex* and *endIndex*. |
| *public StringBuilder delete(int startIndex, int endIndex)* | It is used to delete the string from specified *startIndex* and *endIndex*. |
| *public StringBuilder reverse()* | It is used to reverse the string. |
| *public int capacity()* | It is used to return the current capacity. |
| *public void ensureCapacity(int minimumCapacity)* | It is used to ensure the capacity at least equal to the given minimum. |
| *public char charAt(int index)* | It is used to return the character at the specified position. |
| *public int length()* | It is used to return the length of the string i.e. total number of characters. |
| *public String substring(int beginIndex)* | It is used to return the substring from the specified *beginIndex*. |
| *public String substring(int beginIndex, int endIndex)* | It is used to return the substring from the specified *beginIndex* and *endIndex*. |

Now let us review some of the important methods.

**StringBuilder append() Method**

The StringBuilder append() method concatenates the given argument with this string.

class StringBuilderExample{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

**StringBuilder insert() Method**

The StringBuilder insert() method inserts the given string with this string at the given position.

class StringBuilderExample2{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

**StringBuilder replace() Method**

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

class StringBuilderExample3{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

**StringBuilder delete() Method**

The *delete()* method of *StringBuilder* class deletes the string from the specified *beginIndex* to *endIndex*.

class StringBuilderExample4{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

**StringBuilder reverse() Method**

The *reverse()* method of *StringBuilder* class reverses the current string.

class StringBuilderExample5{

public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

**StringBuilder capacity() Method**

The *capacity()* method of *StringBuilder* class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBuilderExample6{

public static void main(String args[]){

StringBuilder sb=new StringBuilder();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

**StringBuilder ensureCapacity() Method**

The *ensureCapacity()* method of *StringBuilder* class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBuilderExample7{

public static void main(String args[]){

StringBuilder sb=new StringBuilder();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

}

Difference between StringBuffer and StringBuilder

There are many differences between *StringBuffer* and *StringBuilder*, such as:

|  |  |  |
| --- | --- | --- |
| No. | StringBuffer | StringBuilder |
| 1) | *StringBuffer* is synchronizedi.e. it is considered to be thread safe. This means two threads can't call the methods of *StringBuffer* simultaneously. | *StringBuilder* is non-synchronized i.e. it is not considered to be thread safe. This means two threads can call the methods of *StringBuilder* simultaneously. |
| 2) | *StringBuffer* is less efficient than *StringBuilder*. | *StringBuilder* is more efficient than *StringBuffer*. |

Summary

* A string is an object that represents a sequence of characters
* The *getChars( )*method is used if you need to extract more than one character at a time
* String operations are mostly useful for automation testing practice
* The *capacity()* method of *StringBuffer* class returns the current capacity of the buffer
* The Java *StringBuilder* class is the same as the *StringBuffer* class, except that it is non-synchronized
* The *ensureCapacity()* method of *StringBuilder* class ensures that the given capacity is the minimum to the current capacity

# Chapter 09 – Windows Programming

## **Java AWT(Abstract Window Toolkit)**

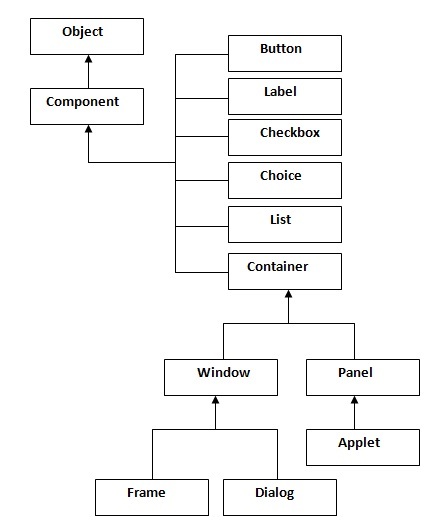
**As the name suggests, Java AWT** is an API that is used to develop GUI or window-based applications in Java.

Java AWT components are heavyweight and platform-dependent, that is, these components heavily draw upon the resources of the operating system. They are displayed according to the view of the operating system.

The java.awt package provides classes for AWT API, such as TextField, Label, TextArea, Button, CheckBox, Choice, List and the like.

## Introduction to AWT Components

The hierarchy of Java AWT classes isas follows:



**Container**

The container is a component in AWT, which can contain other components like buttons, text fields, labels etc. The classes that extend a container class are known as a container such as a Frame, Dialog and Panel.

**Window**

The window, a type of container, has no borders and menu bars. You must use a frame, dialogue or another window for creating a window.

**Panel**

The panel is a type of a container that doesn't contain title bar and menu bars. It can have other components such as button and textfield.

**Frame**

The frame is a type of a container that contains title bar and can have menu bars. It can have other components such as button and textfield as well.

## Layout Manager

Layout means the arrangement of components within the container. In another way, we can say that placing the components at a particular position within the container. The task to layout the controls is done automatically by the Layout Manager.

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

* It is very tedious to handle a large number of controls within the container.
* Often the width and height information of a component is not given when we need to arrange them.

Java provides various layout manager to position the controls. The properties like size,shape and arrangement vary from one layout manager to another layout manager. When the size of the applet or the application window changes, the size, shape and arrangement of the components also change in response. This means that the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the *LayoutManager* interface.

Following are the interfaces defining functionalities of Layout Managers.

|  |  |
| --- | --- |
| **Sr. No.** | **Interface & Description** |
| 1 | [**LayoutManager**](https://www.tutorialspoint.com/awt/awt_layoutmanager.htm)  The LayoutManager interface declares those methods which need to be implemented by the class whose object will act as a layout manager. |
| 2 | [**LayoutManager2**](https://www.tutorialspoint.com/awt/awt_layoutmanager2.htm)  The LayoutManager2 is the sub-interface of the *LayoutManager*.This interface is for those classes that know how to layout containers based on layout constraint object. |

**AWT Layout Manager Classes:**

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

|  |  |
| --- | --- |
| Sr. No. | LayoutManager& Description |
| 1 | **BorderLayout**  The borderlayout is a default layout for Frame.It arranges the components to fit in the five regions: east, west, north, south and centre. |
| 2 | **CardLayout**  The CardLayout object treats each component in the container as a card. Only one card is visible at a time. |
| 3 | **FlowLayout**  The FlowLayout is the default layout for Panel.It layouts the components in a directional flow. |
| 4 | **GridLayout**  The GridLayout manages the components in form of a rectangular grid. |
| 5 | **GridBagLayout**  This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically, horizontally or along their baseline without requiring the components of the same size. |

**Example of BorderLayout class**

import java.awt.\*;

public class Border {

Frame f;

Border(){

f=new Frame();

Button b1=new Button("NORTH");;

Button b2=new Button("SOUTH");;

Button b3=new Button("EAST");;

Button b4=new Button("WEST");;

Button b5=new Button("CENTER");;

f.add(b1,BorderLayout.NORTH);

f.add(b2,BorderLayout.SOUTH);

f.add(b3,BorderLayout.EAST);

f.add(b4,BorderLayout.WEST);

f.add(b5,BorderLayout.CENTER);

f.setSize(300,300);

f.setVisible(true);

}

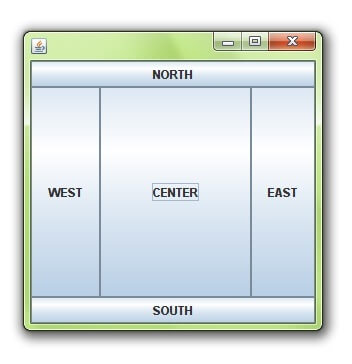
public static void main(String[] args) {

new Border();

}

}

Output



**Example of CardLayout class**

import java.awt.\*;

import java.awt.event.\*;

public class CardLayoutExample extends Frame implements ActionListener{

CardLayout card;

Panel deck;

Button b1,b2,b3;

CardLayoutExample(){

deck=new Panel();

card=new CardLayout(40,30);

//create CardLayout object with 40 hor space and 30 ver space

deck.setLayout(card);

b1=new Button("Apple");

b2=new Button("Boy");

b3=new Button("Cat");

b1.addActionListener(this);

b2.addActionListener(this);

b3.addActionListener(this);

deck.add("a",b1);deck.add("b",b2);deck.add("c",b3);

}

public void actionPerformed(ActionEvent e) {

card.next(deck);

}

public static void main(String[] args) {

CardLayoutExample cl=new CardLayoutExample();

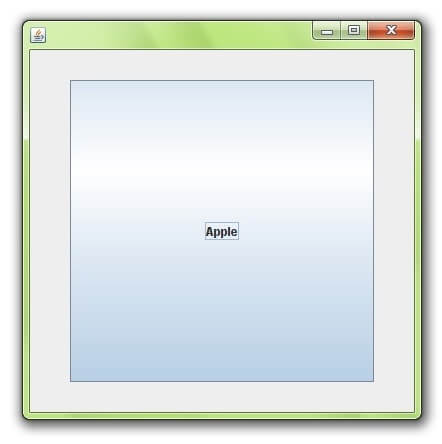
cl.setSize(400,400);

cl.setVisible(true);

}

}

Output



**Example of FlowLayout class**

import java.awt.\*;

public class MyFlowLayout{

Frame f;

MyFlowLayout(){

f=new Frame();

Button b1=new Button("1");

Button b2=new Button("2");

Button b3=new Button("3");

Button b4=new Button("4");

Button b5=new Button("5");

f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);

f.setLayout(new FlowLayout(FlowLayout.RIGHT));

//setting flow layout of right alignment

f.setSize(300,300);

f.setVisible(true);

}

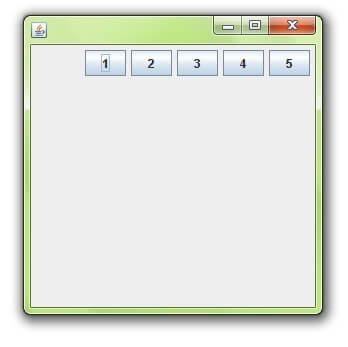
public static void main(String[] args) {

new MyFlowLayout();

}

}

Output



**Example of GridLayout class**

import java.awt.\*;

public class MyGridLayout{

Frame f;

MyGridLayout(){

f=new Frame();

Button b1=new Button("1");

Button b2=new Button("2");

Button b3=new Button("3");

Button b4=new Button("4");

Button b5=new Button("5");

Button b6=new Button("6");

Button b7=new Button("7");

Button b8=new Button("8");

Button b9=new Button("9");

f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);

f.add(b6);f.add(b7);f.add(b8);f.add(b9);

f.setLayout(new GridLayout(3,3));

//setting grid layout of 3 rows and 3 columns

f.setSize(300,300);

f.setVisible(true);

}

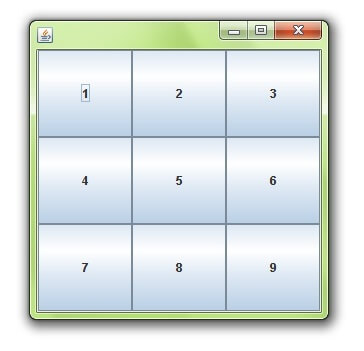
public static void main(String[] args) {

new MyGridLayout();

}

}

Output



**Example of GridBagLayout class**

import java.awt.Button;

import java.awt.GridBagConstraints;

import java.awt.GridBagLayout;

public class GridBagLayoutExample extends Frame{

public static void main(String[] args) {

GridBagLayoutExample a = new GridBagLayoutExample();

}

public GridBagLayoutExample() {

GridBagLayoutgrid = new GridBagLayout();

GridBagConstraints gbc = new GridBagConstraints();

setLayout(grid);

setTitle("GridBag Layout Example");

GridBagLayout layout = new GridBagLayout();

this.setLayout(layout);

gbc.fill = GridBagConstraints.HORIZONTAL;

gbc.gridx = 0;

gbc.gridy = 0;

this.add(new Button("Button One"), gbc);

gbc.gridx = 1;

gbc.gridy = 0;

this.add(new Button("Button two"), gbc);

gbc.fill = GridBagConstraints.HORIZONTAL;

gbc.ipady = 20;

gbc.gridx = 0;

gbc.gridy = 1;

this.add(new Button("Button Three"), gbc);

gbc.gridx = 1;

gbc.gridy = 1;

this.add(new Button("Button Four"), gbc);

gbc.gridx = 0;

gbc.gridy = 2;

gbc.fill = GridBagConstraints.HORIZONTAL;

gbc.gridwidth = 2;

this.add(new Button("Button Five"), gbc);

setSize(300, 300);

setPreferredSize(getSize());

setVisible(true);

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

}

}

Output



## **Java Font and Color Classes**

**Font Class**

The Font class states fonts, which are used to render text in a visible way.

**Class declaration**

Following is the declaration for *java.awt.Font* class:

public class Font

extends Object

implements Serializable

**Field**

Following are the fields for *java.awt.geom.Arc2D* class:

* **static int BOLD**- The bold style constant.
* **static int CENTER\_BASELINE**-The baseline used in ideographic scripts like Chinese, Japanese, and Korean when laying out text.
* **static String DIALOG**-A String constant for the canonical family name of the logical font "Dialog".
* **static String DIALOG\_INPUT**- A String constant for the canonical family name of the logical font "DialogInput".
* **static int HANGING\_BASELINE**- The baseline used in Devanigiri and similar scripts when laying out text.
* **static int ITALIC**- The italicized style constant.
* **static int LAYOUT\_LEFT\_TO\_RIGHT**-- A flag to layoutGlyphVector indicating that text is left-to-right as determined by Bidi analysis.
* **static int LAYOUT\_NO\_LIMIT\_CONTEXT**- A flag to layoutGlyphVector indicating that text in the char array after the indicated limit should not be examined.
* **static int LAYOUT\_NO\_START\_CONTEXT**- A flag to layoutGlyphVector indicating that text in the char array before the indicated start should not be examined.
* **static int LAYOUT\_RIGHT\_TO\_LEFT**- A flag to layoutGlyphVector indicating that text is right-to-left as determined by Bidi analysis.
* **static String MONOSPACED**- A String constant for the canonical family name of the logical font "Monospaced".
* **protected String name**- The logical name of this Font, as passed to the constructor.
* **static int PLAIN**-The plain style constant.
* **protected float pointSize**- The point size of this Font in float.
* **static int ROMAN\_BASELINE**-The baseline used in most Roman scripts when laying out text.
* **static String SANS\_SERIF**- A String constant for the canonical family name of the logical font "SansSerif".
* **static String SERIF**-A String constant for the canonical family name of the logical font "Serif".
* **protected int size**- The point size of this Font, rounded to the integer.
* **protected int style**- The style of this Font, as passed to the constructor.
* **static int TRUETYPE\_FONT**- Identify a font resource of type TRUETYPE.
* **static int TYPE1\_FONT**- Identify a font resource of type TYPE1.

**Class constructors**

|  |  |
| --- | --- |
| S.No. | Constructor & Description |
| 1 | **protected Font() ()**  Creates a new Font from the specified font. |
| 2 | **Font(Map<? extends AttributedCharacterIterator.Attribute,?> attributes)**  Creates a new Font from the specified font. |
| 3 | **Font(String name, int style, int size)**  Creates a new Font from the specified font. |

**Class methods**

|  |  |
| --- | --- |
| S.No. | Method & Description |
| 1 | **setfont()**  setfont() is a method used to set the font of the text present in program. |
| 2 | **getfont()**  getfont() is a method used to pick the font from the program and provide it as output. |

**Let us see an example of setfont() below:**

import java.awt.\*;

public class FontClass extends Frame

{

Font f;

public FontClass()

{

f=new Font("Arial",Font.ITALIC,20);

Label l=new Label("Welcome to Java");

l.setFont(f);

l.setForeground(Color.red);

add(l);

setVisible(true);

setSize(300,300);

}

public static void main(String[] args)

{

new FontClass();

}

}

**Methods Inherited**

This class inherits methods from the following class:

* java.lang.Object

## Event Classes

The event classes represent an event. Although there are numerous event classes available in Java, we shall discuss only the commonly used ones in this chapter.

**EventObject class**

*The EventObject* class is the root class from which all *event state objects* shall be derived. All events are constructed with a reference to an object. This class is defined in *java.util* package.

**Class declaration**

Following is the declaration for *java.util.EventObject* class:

public class EventObject

extends Object

implements Serializable

**Field**

Following are the fields for *java.util.EventObject* class:

* **Protected Object source** - The object on which the event initially occurred.

**Class constructors**

|  |  |
| --- | --- |
| **S.N.** | **Constructor & Description** |
|  | **EventObject(Object source):** It is used to construct a prototypicalevent. |

**Class methods**

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
|  | **Object getSource():** This represents the object on which the event initially occurred. |
|  | **String toString():** It returns a string representation of this EventObject. |

**Methods inherited**

This class inherits methods from the following classes:

* java.lang.Object

The table below lists the commonly used event classes in Java.

|  |  |
| --- | --- |
| Sr. No. | Control & Description |
|  | **AWTEvent:** It is the root event class for all AWT events. This class and its subclasses supersede the original java.awt.Event class. |
|  | **ActionEvent:** The ActionEvent is generated when a button is clicked or the item of a list is double-clicked. |
|  | **InputEvent:** The InputEvent class is a root event class that is designed for all component-level input events. |
|  | **KeyEvent:** This event class is used to create Key events. On entering a character, the Key event is generated. |
|  | **MouseEvent:** This event indicates that a mouse action has occurred in a component. |
|  | **TextEvent:** The object of this class represents the text events. |
|  | **WindowEvent:** The object of this class represents the change in state of a window. |
|  | **AdjustmentEvent:** The object of this class represents the adjustment event emitted by Adjustable objects. |
|  | **ComponentEvent:** The object of this class represents the change in state of a window. |

## Event Listeners

The event listeners represent the interfaces that are responsible for handling events. Although Java offers numerous event listener classes, we shall focus only on the commonly used ones in this chapter. Every method of an event listener method includes a single argument as an object that belongs to the*EventObject* class, as its subclass. For example, *mouse event listener methods* will accept an instance of *MouseEvent*, where *MouseEvent* derives from *EventObject*.

**EventListner interface**

It is a marker interface that every *listener interface* has to extend. This class is defined in *java.util* package.

**Class declaration**

The *java.util.EventListener* interface can be declared as follows:

*public interface EventListener*

The table below lists the commonly used event listeners in Java.

|  |  |
| --- | --- |
| Sr. No. | Control & Description |
|  | **ActionListener:** This interface is used for receiving the action events. |
|  | **ComponentListener:** This interface is used for receiving the component events. |
|  | **ItemListener:** This interface is used for receiving the item events. |
|  | **KeyListener:** This interface is used for receiving the key events. |
|  | **MouseListener:** This interface is used for receiving the mouse events. |
|  | **TextListener:** This interface is used for receiving the text events. |
|  | **WindowListener:** This interface is used for receiving the window events. |
|  | **AdjustmentListener:** This interface is used for receiving the adjustment events. |
|  | **ContainerListener:** This interface is used for receiving the container events. |
|  | **MouseMotionListener:** This interface is used for receiving the mouse motion events. |
|  | **FocusListener:** This interface is used for receiving the focus events. |

**Let us see an example of ActionListener below:**

**AwtListenerDemo.java**

import java.awt.\*;

import java.awt.event.\*;

public class AwtListenerDemo {

private Frame mainFrame;

private Label headerLabel;

private Label statusLabel;

private Panel controlPanel;

public AwtListenerDemo(){

prepareGUI();

}

public static void main(String[] args){

AwtListenerDemo awtListenerDemo = new AwtListenerDemo();

awtListenerDemo.showActionListenerDemo();

}

private void prepareGUI(){

mainFrame = new Frame("Java AWT Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(new GridLayout(3, 1));

mainFrame.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent windowEvent){

System.exit(0);

}

});

headerLabel = new Label();

headerLabel.setAlignment(Label.CENTER);

statusLabel = new Label();

statusLabel.setAlignment(Label.CENTER);

statusLabel.setSize(350,100);

controlPanel = new Panel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);

}

private void showActionListenerDemo(){

headerLabel.setText("Listener in action: ActionListener");

ScrollPane panel = new ScrollPane();

panel.setBackground(Color.magenta);

Button okButton = new Button("OK");

okButton.addActionListener(new CustomActionListener());

panel.add(okButton);

controlPanel.add(panel);

mainFrame.setVisible(true);

}

class CustomActionListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

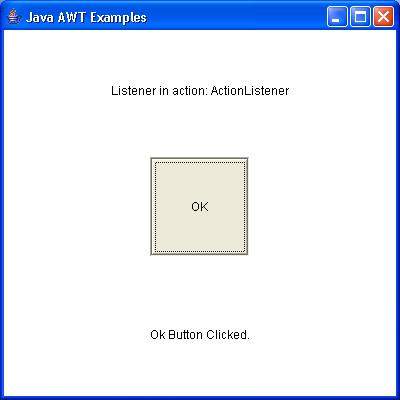
statusLabel.setText("Ok Button Clicked.");

}

}

}

**This will give Following Output:**

****

## Java Adapter Classes

Java adapter classes provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So, it savesthe code.

The adapter classes are found in **java.awt.event, java.awt.dnd** and **javax.swing.event** packages.

The Adapter classes with their corresponding listener interfaces are given below:

**java.awt.event Adapter classes**

|  |  |
| --- | --- |
| Adapter class | Listener interface |
| WindowAdapter | WindowListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener |
| MouseMotionAdapter | MouseMotionListener |
| FocusAdapter | FocusListener |
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| HierarchyBoundsAdapter | HierarchyBoundsListener |

**ava.awt.dnd Adapter classes**

|  |  |
| --- | --- |
| Adapter class | Listener interface |
| DragSourceAdapter | DragSourceListener |
| DragTargetAdapter | DragTargetListener |

**javax.swing.event Adapter classes**

|  |  |
| --- | --- |
| Adapter class | Listener interface |
| MouseInputAdapter | MouseInputListener |
| InternalFrameAdapter | InternalFrameListener |

Let us see an example which shows implementation of listener interfaces and adapter class.

**Example**

**Keylistener**

import java.awt.\*;

import java.awt.event.\*;

public class AwtListenerDemo {

private Frame mainFrame;

private Label headerLabel;

private Label statusLabel;

private Panel controlPanel;

private TextField textField;

public AwtListenerDemo(){

prepareGUI();

}

public static void main(String[] args){

AwtListenerDemo awtListenerDemo = new AwtListenerDemo();

awtListenerDemo.showKeyListenerDemo();

}

private void prepareGUI(){

mainFrame = new Frame("Java AWT Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(new GridLayout(3, 1));

mainFrame.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent windowEvent){

System.exit(0);

}

});

headerLabel = new Label();

headerLabel.setAlignment(Label.CENTER);

statusLabel = new Label();

statusLabel.setAlignment(Label.CENTER);

statusLabel.setSize(350,100);

controlPanel = new Panel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);

}

private void showKeyListenerDemo(){

headerLabel.setText("Listener in action: KeyListener");

textField = new TextField(10);

textField.addKeyListener(new CustomKeyListener());

Button okButton = new Button("OK");

okButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

statusLabel.setText("Entered text: " + textField.getText());

}

});

controlPanel.add(textField);

controlPanel.add(okButton);

mainFrame.setVisible(true);

}

class CustomKeyListener implements KeyListener{

public void keyTyped(KeyEvent e) {

}

public void keyPressed(KeyEvent e) {

if(e.getKeyCode() == KeyEvent.VK\_ENTER){

statusLabel.setText("Entered text: " + textField.getText());

}

}

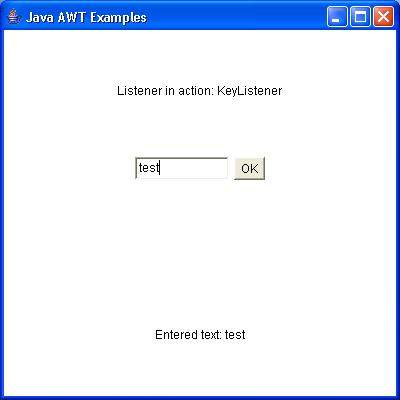
public void keyReleased(KeyEvent e) {

}

}

}

Output:



**Keyadapter**

import java.awt.\*;

import java.awt.event.\*;

public class AwtAdapterDemo {

private Frame mainFrame;

private Label headerLabel;

private Label statusLabel;

private Panel controlPanel;

public AwtAdapterDemo(){

prepareGUI();

}

public static void main(String[] args){

AwtAdapterDemo awtAdapterDemo = new AwtAdapterDemo();

awtAdapterDemo.showKeyAdapterDemo();

}

private void prepareGUI(){

mainFrame = new Frame("Java AWT Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(new GridLayout(3, 1));

mainFrame.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent windowEvent){

System.exit(0);

}

});

headerLabel = new Label();

headerLabel.setAlignment(Label.CENTER);

statusLabel = new Label();

statusLabel.setAlignment(Label.CENTER);

statusLabel.setSize(350,100);

controlPanel = new Panel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);

}

private void showKeyAdapterDemo(){

headerLabel.setText("Listener in action: KeyAdapter");

final TextField textField = new TextField(10);

textField.addKeyListener(new KeyAdapter() {

public void keyPressed(KeyEvent e) {

if(e.getKeyCode() == KeyEvent.VK\_ENTER){

statusLabel.setText("Entered text: " + textField.getText());

}

}

});

Button okButton = new Button("OK");

okButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

statusLabel.setText("Entered text: " + textField.getText());

}

});

controlPanel.add(textField);

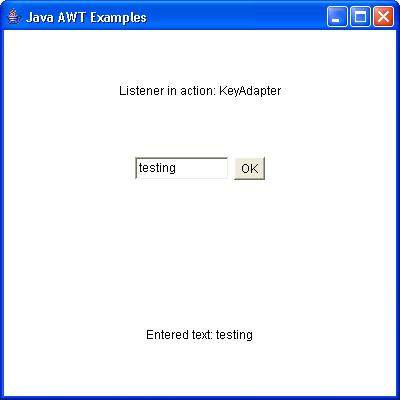
controlPanel.add(okButton);

mainFrame.setVisible(true);

}

}

Output:



## Graphics Class

The Graphics class is the abstract superclass for all graphics contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well.

A Graphics object encapsulates all state information required for the basic rendering operations that Java supports. State information includes the following properties.

* The Component object on which to draw
* A translation origin for rendering and clipping coordinates
* The current clip
* The current color
* The current font
* The current logical pixel operation function
* The current XOR alternation color

**Class declaration**

Following is the declaration for java.awt.Graphics class:

public abstract class Graphics

extends Object

**Class constructors**

|  |  |
| --- | --- |
| S.No. | Constructor & Description |
| 1 | **Graphics() ()**  Constructs a new Graphics object. |

**Class methods**

|  |  |
| --- | --- |
| S.N. | Method & Description |
| 1 | **drawLine()**  The drawLine() method draws a line on the graphics context in the current color that begins at startX,startY and ends at endX,endY. If (x1, y1) and (x2, y2) are the same point, it will draw a point. There is no method specific to drawing a point. |
| 2 | **drawRect()**  The drawRect() method draws a rectangle on the drawing area in the current color from (x, y) to (x+width, y+height). If width or height is negative, nothing is drawn. |
| 3 | **fillRect()**  The fillRect() method draws a filled rectangle on the drawing area in the current color from (x, y) to (x+width-1, y+height-1). The filled rectangle is one pixel smaller to the right and bottom than requested. If width or height is negative, nothing is drawn. |
| 4 | **drawOval ()**  The drawOval() method draws an oval in the current color within an invisible bounding rectangle from (x, y) to (x+width, y+height). You cannot specify the oval's center point and radii. If width and height are equal, you get a circle. If width or height is negative, nothing is drawn. |
| 5 | **fillOval ()**  The fillOval() method draws a filled oval in the current color within an invisible bounding rectangle from (x, y) to (x+width-1, y+height-1). You cannot specify the oval's center point and radii. Notice that the filled oval is one pixel smaller to the right and bottom than requested. If width or height is negative, nothing is drawn. |

**The following example illustrates the drawLine() method**:

import java.awt.\*;

public class Lines extends Frame

{

public void paint(Graphics g)

{

g.drawLine(0, 0, 100, 100);

g.drawLine(0, 100, 100, 0);

g.drawLine(40, 25, 250, 180);

g.drawLine(75, 90, 400, 400);

g.drawLine(20, 150, 400, 40); //line

g.drawLine(5, 290, 80, 19); //line

g.drawLine (5, 75, 5, 75); // point

g.drawLine (50, 5, 50, 5); // point

}

public static void main(String[] args)

Lines l=new Lines();

l.setSize(300,300);

l.setVisible(true);

}

**The following example illustrates the drawOval() and fillOval() method**:

import java.awt.\*;

public class RectanglesDrawing extends Frame

{

public void paint(Graphics g)

{

g.setColor(Color.blue);

g.drawRect(50, 80, 150, 100);

g.setColor(Color.magenta);

g.fillRect(230, 80, 150, 100);

}

public static void main(String[] args)

RectanglesDrawing rd=new RectanglesDrawing();

rd.setSize(300,300);

rd.setVisible(true);

}

}

**The following example illustrates the drawRect() and fillRect() method:**

import java.awt.\*;

public class Ovals extends Applet

{

public void paint(Graphics g)

{

g.drawOval(10, 10, 50, 50);

g.fillOval(100, 10, 75, 50);

g.drawOval(190, 10, 90, 30);

g.fillOval(70, 90, 140, 100);

}

public static void main(String[] args)

Ovals rd=new Ovals ();

rd.setSize(300,300);

rd.setVisible(true);

}

}

## **Painting**

In Java, custom painting is done via the java.awt.Graphics class, which manages a graphics context, and provides a set of *device-independent* methods for drawing texts, figures and images on the screen on different platforms.

The java.awt.Graphics is an abstract class, as the actual act of drawing is system-dependent and device-dependent. Each operating platform will provide a subclass of Graphics to perform the actual drawing under the platform, but conform to the specification defined in Graphics.we have already implemented painting in above examples.

## **Problems with AWT**

The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents or peers. This means that the look and feel of a component is defined by the platform, not by java.

Because the AWT components use native code resources, they are referred to as heavy weight. The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms.

This variability threatened java’s philosophy: write once, run anywhere. Second, the look and feel of each component was fixed and could not be changed. Third, the use of heavyweight components caused some frustrating restrictions. Due to these limitations Swing came and was integrated to java. Swing is built on the AWT.

## **Java Swing**

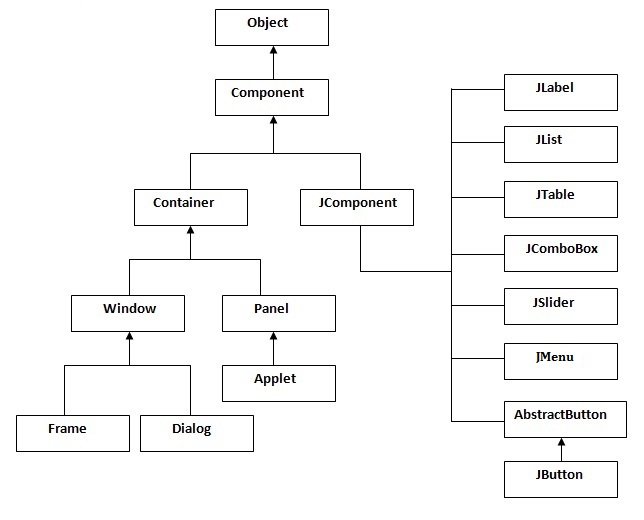
A described, Java Swing is a part of JFC that is used to create window-based applications. It is built on the top of AWT-based API and written in Java.

Unlike AWT, Java Swing is used to design platform-independent and lightweight components.

The *javax.swing* package provides classes for JavaSwing API, such as *JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser*and so on.

**Hierarchy of Java Swing Classes**

The hierarchy of JavaSwing API is as follows:



**Java Swing Examples**

In Java, you can adopt two ways to create a frame. These are by:

* Creating the object of *Frame class* (association)
* Extending *Frame class* (inheritance)

We can write the code to incorporateSwing inside the *main()*, constructor or any other method.

**Swing example:**

In the below example we would be using several swing components. The below swing program would create a login screen.

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JPasswordField;

import javax.swing.JTextField;

public class SwingFirstExample {

public static void main(String[] args) {

// Creating instance of JFrame

JFrame frame = new JFrame("My First Swing Example");

// Setting the width and height of frame

frame.setSize(350, 200);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

/\* Creating panel. This is same as a div tag in HTML

\* We can create several panels and add them to specific

\* positions in a JFrame. Inside panels we can add text

\* fields, buttons and other components.

\*/

JPanel panel = new JPanel();

// adding panel to frame

frame.add(panel);

/\* calling user defined method for adding components

\* to the panel.

\*/

placeComponents(panel);

// Setting the frame visibility to true

frame.setVisible(true);

}

private static void placeComponents(JPanel panel) {

/\* We will discuss about layouts in the later sections

\* of this tutorial. For now we are setting the layout

\* to null

\*/

panel.setLayout(null);

// Creating JLabel

JLabel userLabel = new JLabel("User");

/\* This method specifies the location and size

\* of component. setBounds(x, y, width, height)

\* here (x,y) are cordinates from the top left

\* corner and remaining two arguments are the width

\* and height of the component.

\*/

userLabel.setBounds(10,20,80,25);

panel.add(userLabel);

/\* Creating text field where user is supposed to

\* enter user name.

\*/

JTextField userText = new JTextField(20);

userText.setBounds(100,20,165,25);

panel.add(userText);

// Same process for password label and text field.

JLabel passwordLabel = new JLabel("Password");

passwordLabel.setBounds(10,50,80,25);

panel.add(passwordLabel);

/\*This is similar to text field but it hides the user

\* entered data and displays dots instead to protect

\* the password like we normally see on login screens.

\*/

JPasswordField passwordText = new JPasswordField(20);

passwordText.setBounds(100,50,165,25);

panel.add(passwordText);

// Creating login button

JButton loginButton = new JButton("login");

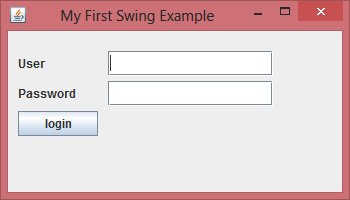
loginButton.setBounds(10, 80, 80, 25);

panel.add(loginButton);

}

}

**This will give following Output:**

****

## 

## Controls

Below is the full list of JavaSwing controls:

* Java Swing Button Control
* Java Swing CheckBox Control
* Java Swing ComboBox Control
* Java Swing List Control
* Java Swing Menu Bar Control
* Java Swing Password Field Control
* Java Swing Radio Button Control
* Java Swing ScrollBar Control
* Java Swing ScrollPane Control
* Java Swing Slider Control
* Java Swing Spinner Control
* Java Swing TabbedPane Control
* Java Swing Table Control
* Java Swing TextArea Control
* Java Swing TextField Control
* Java Swing ToggleButton Control
* Java Swing ToolBar Control
* Java Swing Tree Control

## Look and Feel of Swing Components

Swing is *GUI Widget Toolkit* for Java. It is an API for providing Graphical User Interface to Java Programs. Unlike AWT, Swing components are written in Java and therefore are platform-independent. Swing provides platform specific Look and Feel and also an option for pluggable Look and Feel, allowing application to have Look and Feel independent of underlying platform.

Initially there were very few options for colors and other settings in Java Swing, which made the entire application look boring and monotonous. With the growth in Java framework, new changes were introduced to make the UI better and thus giving developer opportunity to enhance the look of a Java Swing Application.

“Look” refers to the appearance of GUI widgets and “feel” refers to the way the widgets behave.

**Available Look and Feels**

Sun’s JRE provides the following Look &Feels:

* **CrossPlatformLookAndFeel:** this is the “Java L&F” also known as “Metal” that looks the same on all platforms. It is part of the Java API (javax.swing.plaf.metal) and is the default.Set cross-platform Java L&F (also called “Metal”).
* **SystemLookAndFeel:** here, the application uses the L&F that is default to the system it is running on. The System L&F is determined at runtime, where the application asks the system to return the name of the appropriate L&F.  
  For Linux and Solaris, the System L&Fs are “GTK+” if GTK+ 2.2 or later is installed, “Motif” otherwise. For Windows, the System L&F is “Windows”.
* **MotifLookAndFeel:**
* **Nimbus Look and Feel**:Nimbus is a polished cross-platform look and feel introduced in the Java SE 6 Update 10 (6u10) release. The following screen capture, from SwingSet3 shows the Nimbus look and feel.
* **Multiplexing**: a way to have the UI methods delegate to a number of different look and feel implementations at the same time.

**We will see different Look and Feel themes with the help of a simple calculator program:**

**CrossPlatformLookAndFeel:**

// Java program to illustrate

// CrossPlatformLookAndFeel

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.UIManager;

import javax.swing.JFrame;

class Awt implements ActionListener {

JFrame f;

JButton addbut, subbut, mulbut, divbut, b5;

JTextField t1, t2, t3;

JLabel l, l1;

Awt()

{

f = new JFrame("Cross Platform Look and Feel");

t1 = new JTextField(" ");

t2 = new JTextField(" ");

t3 = new JTextField(" ");

addbut = new JButton("Add");

subbut = new JButton("Sub");

mulbut = new JButton("Mul");

divbut = new JButton("Div");

l = new JLabel();

l1 = new JLabel();

}

public void awt1()

{

f.setLayout(new GridLayout(3, 2));

f.setVisible(true);

f.add(t1);

f.add(t2);

f.add(t3);

f.add(addbut);

f.add(subbut);

f.add(mulbut);

f.add(divbut);

f.add(l);

f.add(l1);

addbut.addActionListener(this);

subbut.addActionListener(this);

mulbut.addActionListener(this);

divbut.addActionListener(this);

f.pack();

}

public void actionPerformed(ActionEvent e)

{

String s = new String(e.getActionCommand());

l.setText(s);

if ((s).equals("Add")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a + b;

t3.setText(c.toString());

}

else if ((s).equals("Sub")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a - b;

t3.setText(c.toString());

}

else if ((s).equals("Mul")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a \* b;

t3.setText(c.toString());

}

}

public static void main(String args[])

{

try {

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

}

catch (Exception e) {

System.out.println("Look and Feel not set");

}

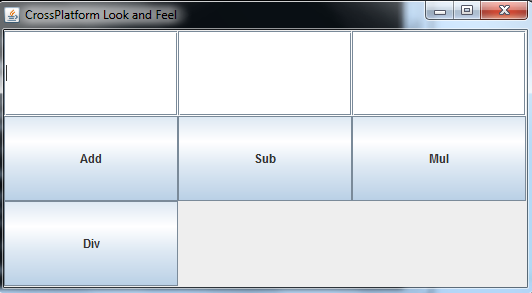
Awt a = new Awt();

a.awt1();

}

}

**This will give following Output:**

****

**SystemLookAndFeel:**

// Java program to illustrate

// **SystemLookAndFeel**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.UIManager;

import javax.swing.JFrame;

class Awt implements ActionListener {

JFrame f;

JButton addbut, subbut, mulbut, divbut, b5;

JTextField t1, t2, t3;

JLabel l, l1;

Awt()

{

f=new JFrame("System Look and Feel");

t1 = new JTextField(" ");

t2 = new JTextField(" ");

t3 = new JTextField(" ");

addbut = new JButton("Add");

subbut = new JButton("Sub");

mulbut = new JButton("Mul");

divbut = new JButton("Div");

l = new JLabel();

l1 = new JLabel();

}

public void awt1()

{

f.setLayout(new GridLayout(3, 2));

f.setVisible(true);

f.add(t1);

f.add(t2);

f.add(t3);

f.add(addbut);

f.add(subbut);

f.add(mulbut);

f.add(divbut);

f.add(l);

f.add(l1);

addbut.addActionListener(this);

subbut.addActionListener(this);

mulbut.addActionListener(this);

divbut.addActionListener(this);

f.pack();

}

public void actionPerformed(ActionEvent e)

{

String s = new String(e.getActionCommand());

l.setText(s);

if ((s).equals("Add")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a + b;

t3.setText(c.toString());

}

else if ((s).equals("Sub")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a - b;

t3.setText(c.toString());

}

else if ((s).equals("Mul")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a \* b;

t3.setText(c.toString());

}

}

public static void main(String args[])

{

try {

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

}

catch (Exception e) {

System.out.println("Look and Feel not set");

}

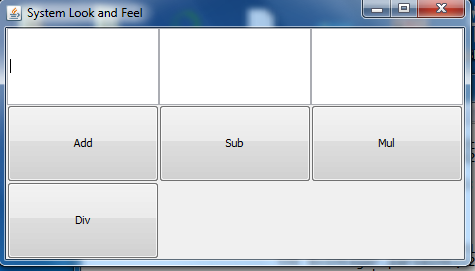
Awt a = UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");

a.awt1();

}

}

**This will give following output:**

****

**MotifLookAndFeel:**

// Java program to illustrate

// **MotifLookAndFeel**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.UIManager;

import javax.swing.JFrame;

class Awt implements ActionListener {

JFrame f;

JButton addbut, subbut, mulbut, divbut, b5;

JTextField t1, t2, t3;

JLabel l, l1;

Awt()

{

f = f=new JFrame("Motif Look and Feel");

t1 = new JTextField(" ");

t2 = new JTextField(" ");

t3 = new JTextField(" ");

addbut = new JButton("Add");

subbut = new JButton("Sub");

mulbut = new JButton("Mul");

divbut = new JButton("Div");

l = new JLabel();

l1 = new JLabel();

}

public void awt1()

{

f.setLayout(new GridLayout(3, 2));

f.setVisible(true);

f.add(t1);

f.add(t2);

f.add(t3);

f.add(addbut);

f.add(subbut);

f.add(mulbut);

f.add(divbut);

f.add(l);

f.add(l1);

addbut.addActionListener(this);

subbut.addActionListener(this);

mulbut.addActionListener(this);

divbut.addActionListener(this);

f.pack();

}

public void actionPerformed(ActionEvent e)

{

String s = new String(e.getActionCommand());

l.setText(s);

if ((s).equals("Add")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a + b;

t3.setText(c.toString());

}

else if ((s).equals("Sub")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a - b;

t3.setText(c.toString());

}

else if ((s).equals("Mul")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a \* b;

t3.setText(c.toString());

}

}

public static void main(String args[])

{

try {

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

}

catch (Exception e) {

System.out.println("Look and Feel not set");

}

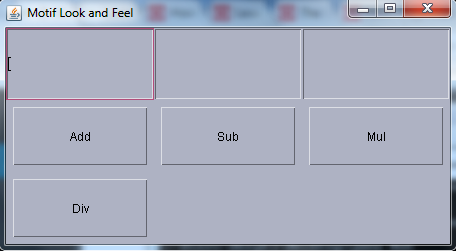
Awt a = UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");

a.awt1();

}

}

**This will give the following output:**

****

**NimbusLookAndFeel:**

// Java program to illustrate

// **NimbusLookAndFeel**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.UIManager;

import javax.swing.JFrame;

class Awt implements ActionListener {

JFrame f;

JButton addbut, subbut, mulbut, divbut, b5;

JTextField t1, t2, t3;

JLabel l, l1;

Awt()

{

f = f=new JFrame("Nimbus Look and Feel");

t1 = new JTextField(" ");

t2 = new JTextField(" ");

t3 = new JTextField(" ");

addbut = new JButton("Add");

subbut = new JButton("Sub");

mulbut = new JButton("Mul");

divbut = new JButton("Div");

l = new JLabel();

l1 = new JLabel();

}

public void awt1()

{

f.setLayout(new GridLayout(3, 2));

f.setVisible(true);

f.add(t1);

f.add(t2);

f.add(t3);

f.add(addbut);

f.add(subbut);

f.add(mulbut);

f.add(divbut);

f.add(l);

f.add(l1);

addbut.addActionListener(this);

subbut.addActionListener(this);

mulbut.addActionListener(this);

divbut.addActionListener(this);

f.pack();

}

public void actionPerformed(ActionEvent e)

{

String s = new String(e.getActionCommand());

l.setText(s);

if ((s).equals("Add")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a + b;

t3.setText(c.toString());

}

else if ((s).equals("Sub")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a - b;

t3.setText(c.toString());

}

else if ((s).equals("Mul")) {

int a = Integer.parseInt(t1.getText());

int b = Integer.parseInt(t2.getText());

Integer c = a \* b;

t3.setText(c.toString());

}

}

public static void main(String args[])

{

try {

UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

}

catch (Exception e) {

System.out.println("Look and Feel not set");

}

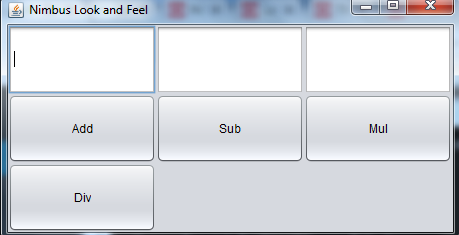
Awt a = UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");

a.awt1();

}

}

**This will give the following output:**

****

For Linux and Solaris, the System L&Fs are "GTK+" if GTK+ 2.2 or later is installed, "Motif" otherwise. For Windows, the System L&F is "Windows," which mimics the L&F of the particular Windows OS that is running - classic Windows, XP, or Vista. The GTK+, Motif, and Windows L&Fs are provided by Sun and shipped with the Java SDK and JRE, although they are not part of the Java API.

Apple provides its own JVM which includes their proprietary L&F.

|  |  |
| --- | --- |
| Platform | Look and Feel |
| Solaris, Linux with GTK+ 2.2 or later | GTK+ |
| Other Solaris, Linux | Motif |
| IBM UNIX | IBM\* |
| HP UX | HP\* |
| Classic Windows | Windows |
| Windows XP | Windows XP |
| Windows Vista | Windows Vista |
| Macintosh | Macintosh\* |

\* Supplied by the system vendor

## MVC Architecture of Swing Components

Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

In general a visual component is a composition of three distinct aspects:

* The way that a particular component looks when rendered on screen
* The way that a particular component reacts to an user
* The state information which is associated with a particular component

No matter what the architecture used to implement a particular component it must implicitly contain the three parts i.e. Model-View-Controller or MVC for short.

In the MVC terminology we have:

**Model:**

The Model corresponds to the state information which is associated with a component. For example-in case of a check box, the model contains a field which indicates whether the box is checked or unchecked.

**View:**

The view determines how a component has displayed on the screen, including any aspects of view that are affected by the current state of the model.

**Controller:**

The controller determines how the component will react to the user.

For example, when user clicks a check box, the controller will reacts by changing model to reflect the user’s choice (checked or unchecked) which then results in the view being updated.

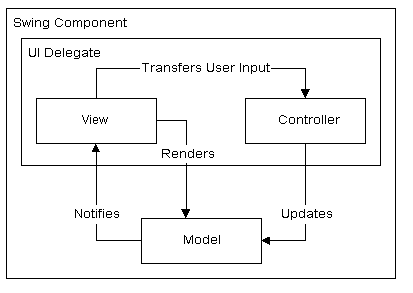


Image shows how the model, view, and controller work together to create a scrollbar component.

Swing actually makes use of a simplified variant of the MVC design called the model-delegate . This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the UI delegate . Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street.

## Working with Image

Swings allow us to perform simple image processing operation such as loading and editing an image and drawing a shape. Let’s understand image operations with Swing.

**Loading an Image**

The first thing is to create a BufferedImage object from a picture saved on our disk drive:

String imagePath = "path/to/your/image.jpg";

BufferedImage myPicture = ImageIO.read(newFile(imagePath));

**Displaying an Image**

Now that we have drawn something on our image, we would like to display it. We can do it using Swing library objects. First, we create JLabel object which is representing a display area for text or/and image:

JLabel picLabel = newJLabel(newImageIcon(myPicture));

Then add our JLabel to JPanel, which we can treat as <div></div> of Java-based GUI:

JPanel jPanel = newJPanel();

jPanel.add(picLabel);

In the end, we add everything to JFrame which is window displayed on a screen. We have to set size so that we don’t have to expand this window every time we run our program:

JFrame f = newJFrame();

f.setSize(newDimension(myPicture.getWidth(), myPicture.getHeight()));

f.add(jPanel);

f.setVisible(true);

**Example of displaying image in swing:**

import javax.swing.\*;

import java.awt.\*;

public class ImageTest

{

public static void main(String[] args)

{

JFrame f = new JFrame();

f.setLayout(new FlowLayout());

f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

ImageIcon img=new ImageIcon("E:/java.png");

JButton b=new JButton(img);

JLabel l=new JLabel(img);

f.add(l);

f.add(b);

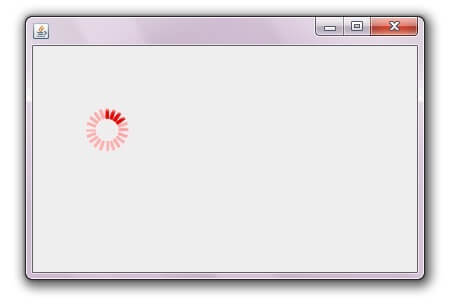
f.setSize(400,300);

f.setVisible(true);

}

}

**This will give following Output:**



## Advanced Swing Components

The Advanced Java Swing Programming focuses on the more complex JFC components, like the JTree component and the JTable component. Below are few advanced swing components:

**JOptionPane**: A component which provides standard methods to pop up a standard dialog box for a value or informs the user of something.

**Let us see an example of JOptionPane below:**

import javax.swing.\*;

public class OptionPaneExample {

JFrame f;

OptionPaneExample(){

f=new JFrame();

JOptionPane.showMessageDialog(f,"Successfully Updated.","Alert",JOptionPane.WARNING\_MESSAGE);

}

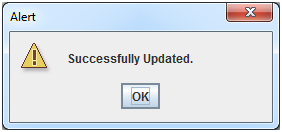
public static void main(String[] args) {

new OptionPaneExample();

}

}

**This will give following Output:**

****

**JTree:** A control that displays a set of hierarchical data as an outline.

**Let us see an example of JTree below:**

import javax.swing.\*;

import javax.swing.tree.DefaultMutableTreeNode;

public class TreeExample {

JFrame f;

TreeExample(){

f=new JFrame();

DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");

DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");

DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");

style.add(color);

style.add(font);

DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");

DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");

DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");

DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");

color.add(red); color.add(blue); color.add(black); color.add(green);

JTree jt=new JTree(style);

f.add(jt);

f.setSize(200,200);

f.setVisible(true);

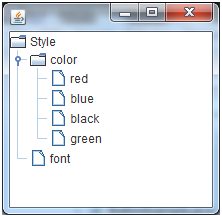
}

public static void main(String[] args) {

new TreeExample();

}}

**This will give following Output:**



**JTable:** The JTable is used to display and edit regular two-dimensional tables of cells.

**Let us see an example of JTable below:**

import javax.swing.\*;

public class TableExample {

JFrame f;

TableExample(){

f=new JFrame();

String data[][]={ {"101","Amit","670000"},

{"102","Jai","780000"},

{"101","Sachin","700000"}};

String column[]={"ID","NAME","SALARY"};

JTable jt=new JTable(data,column);

jt.setBounds(30,40,200,300);

JScrollPane sp=new JScrollPane(jt);

f.add(sp);

f.setSize(300,400);

f.setVisible(true);

}

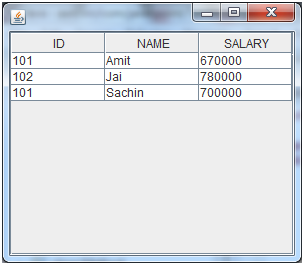
public static void main(String[] args) {

new TableExample();

}

}

**This will give following Output:**

****

**JTabbedPane:** A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon.

**Let us see an example of JTabbedPane below:**

import javax.swing.\*;

public class TabbedPaneExample {

JFrame f;

TabbedPaneExample(){

f=new JFrame();

JTextArea ta=new JTextArea(200,200);

JPanel p1=new JPanel();

p1.add(ta);

JPanel p2=new JPanel();

JPanel p3=new JPanel();

JTabbedPane tp=new JTabbedPane();

tp.setBounds(50,50,200,200);

tp.add("main",p1);

tp.add("visit",p2);

tp.add("help",p3);

f.add(tp);

f.setSize(400,400);

f.setLayout(null);

f.setVisible(true);

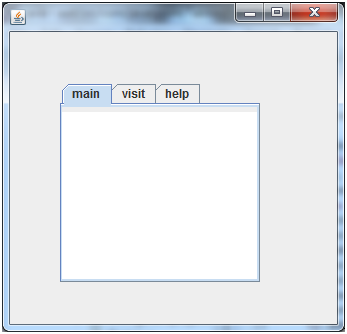
}

public static void main(String[] args) {

new TabbedPaneExample();

}}

**This will give following Output:**

****

**JFileChooser:** JFileChooser provides a simple mechanism for the user to choose a file.

**Let us see an example of JFileChooser below:**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

public class SwingControlDemo {

private JFrame mainFrame;

private JLabel headerLabel;

private JLabel statusLabel;

private JPanel controlPanel;

public SwingControlDemo(){

prepareGUI();

}

public static void main(String[] args){

SwingControlDemo swingControlDemo = new SwingControlDemo();

swingControlDemo.showFileChooserDemo();

}

private void prepareGUI(){

mainFrame = new JFrame("Java Swing Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(new GridLayout(3, 1));

mainFrame.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent windowEvent){

System.exit(0);

}

});

headerLabel = new JLabel("", JLabel.CENTER);

statusLabel = new JLabel("",JLabel.CENTER);

statusLabel.setSize(350,100);

controlPanel = new JPanel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);

}

private void showFileChooserDemo(){

headerLabel.setText("Control in action: JFileChooser");

final JFileChooser fileDialog = new JFileChooser();

JButton showFileDialogButton = new JButton("Open File");

showFileDialogButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

int returnVal = fileDialog.showOpenDialog(mainFrame);

if (returnVal == JFileChooser.APPROVE\_OPTION) {

java.io.File file = fileDialog.getSelectedFile();

statusLabel.setText("File Selected :" + file.getName());

} else {

statusLabel.setText("Open command cancelled by user." );

}

}

});

controlPanel.add(showFileDialogButton);

mainFrame.setVisible(true);

}

}

**This will give following Output:**



**JColorChooser:** JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

public class SwingControlDemo {

private JFrame mainFrame;

private JLabel headerLabel;

private JLabel statusLabel;

private JPanel controlPanel;

public SwingControlDemo(){

prepareGUI();

}

public static void main(String[] args){

SwingControlDemo swingControlDemo = new SwingControlDemo();

swingControlDemo.showColorChooserDemo();

}

private void prepareGUI(){

mainFrame = new JFrame("Java Swing Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(new GridLayout(3, 1));

mainFrame.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent windowEvent){

System.exit(0);

}

});

headerLabel = new JLabel("", JLabel.CENTER);

statusLabel = new JLabel("",JLabel.CENTER);

statusLabel.setSize(350,100);

controlPanel = new JPanel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);

}

private void showColorChooserDemo(){

headerLabel.setText("Control in action: JColorChooser");

JButton chooseButton = new JButton("Choose Background");

chooseButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

Color backgroundColor = JColorChooser.showDialog(mainFrame,

"Choose background color", Color.white);

if(backgroundColor != null){

controlPanel.setBackground(backgroundColor);

mainFrame.getContentPane().setBackground(backgroundColor);

}

}

});

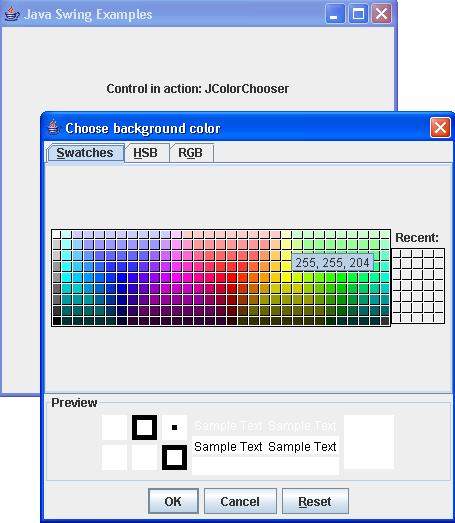
controlPanel.add(chooseButton);

mainFrame.setVisible(true);

}

}

**This will give following Output:**

****

**JMenu:** An implementation of a menu -- a popup window containing JMenuItems that is displayed when the user selects an item on the JMenuBar. In addition to JMenuItems, a JMenu can also contain JSeparators.

In essence, a menu is a button with an associated JPopupMenu. When the "button" is pressed, the JPopupMenu appears. If the "button" is on the JMenuBar, the menu is a top-level window. If the "button" is another menu item, then the JPopupMenu is "pull-right" menu.

**JMenuItem:** An implementation of an item in a menu. A menu item is essentially a button sitting in a list. When the user selects the "button", the action associated with the menu item is performed. A JMenuItem contained in a JPopupMenu performs exactly that function.

**JMenuBar:** An implementation of a menu bar. You add JMenu objects to the menu bar to construct a menu. When the user selects a JMenu object, its associated JPopupMenu is displayed, allowing the user to select one of the JMenuItems on it.

**Program showing example of JMenu, JMenuItem and JMenuBar**

import javax.swing.\*;

class MenuExample

{

JMenu menu, submenu;

JMenuItem i1, i2, i3, i4, i5;

MenuExample(){

JFrame f= new JFrame("Menu and MenuItem Example");

JMenuBar mb=new JMenuBar();

menu=new JMenu("Menu");

submenu=new JMenu("Sub Menu");

i1=new JMenuItem("Item 1");

i2=new JMenuItem("Item 2");

i3=new JMenuItem("Item 3");

i4=new JMenuItem("Item 4");

i5=new JMenuItem("Item 5");

menu.add(i1); menu.add(i2); menu.add(i3);

submenu.add(i4); submenu.add(i5);

menu.add(submenu);

mb.add(menu);

f.setJMenuBar(mb);

f.setSize(400,400);

f.setLayout(null);

f.setVisible(true);

}

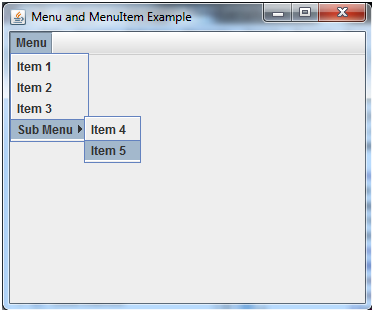
public static void main(String args[])

{

new MenuExample();

}}

**This will give following Output:**



## Summary

* **JFC** include five APIs that include **Accessibility, 2D API, Drag and Drop, AWT**and**Swing**
* **2D API** helps to display GUI components in 2D and 3D elevations
* **Java Swing**is a part of JFC that is used to create window-based applications
* It is used to design platform-independent and lightweight components
* The Frame is a type of a container that contains the title bar and menu bars
* EventListner interface is a marker interface that every listener interface has to extend
* The event listener represents the interfaces responsible to handle events
* The JTable is used to display and edit regular two-dimensional tables of cells

# Chapter 10 – Multithreaded Programming

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. Multitasking is used to utilize the CPU. It can be achieved by two ways:

* Process-based Multitasking(Multiprocessing)
* Thread-based Multitasking(Multithreading)

## Multiprocessing vs. Multithreading

**1) Process-based Multitasking (Multiprocessing)**

* Each process have its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the processes is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

**2) Thread-based Multitasking (Multithreading)**

* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the threads is low.

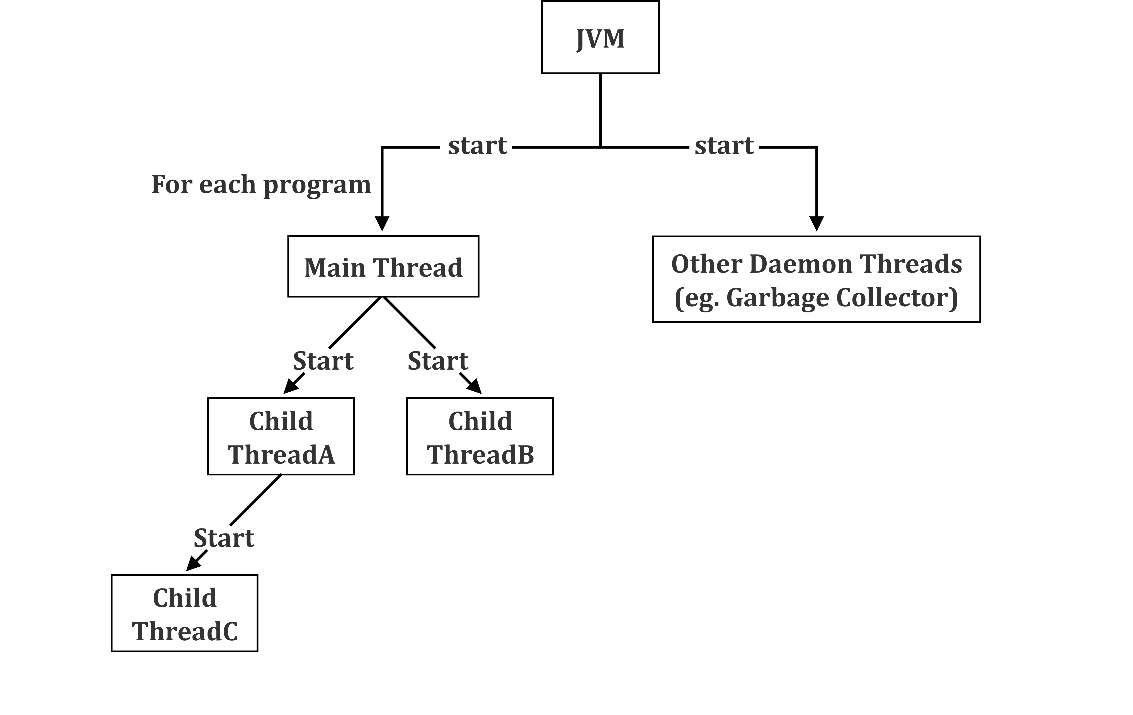
## Main Thread (Default Java Thread)

When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins.

**Properties:**

* It is the thread from which other “child” threads will be spawned.
* Often, it must be the last thread to finish execution because it performs various shutdown actions

**Flow diagram:**



## Creating Child Thread

In Java, threads are objects and can be created in two ways:

* By extending the class Thread
* By implementing the interface Runnable

In the first approach, a user-specified thread class is created by extending the class Thread and overriding its run () method. In the second approach, a thread is created by implementing the Runnable interface and overriding its run () method. In both approaches the run () method has to be overridden. Usually, the code that is to be executed by a thread is written in its run () method. The thread terminates when its run () method returns.

In Java, methods and variables are inherited by a child class from a parent class by extending the parent. By extending the class Thread, however, one can only extend or inherit from a single parent class (in this case, the class Thread is the parent class). This limitation of using extends within Java can be overcome by implementing interfaces. This is the most common way to create threads. A thread that has been created can create and start other threads.

**Creating a new thread by extending Thread**

The first method of creating a thread is simply by extending the Thread class. The Thread class is defined in the package java.lang. The class that inherits overrides the run () method of the parent.

Thread for its implementation. This is done as shown in the code fragment given below. By its side a representation of the inheritance that is being implemented.

public class SampleThread extends Thread

{

public SampleThread (String name)

{

Super (name);

}

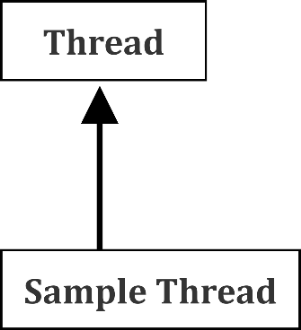
Public void run()

{

//code to be run when this thread is started

}

}



A thread can be started by applying the start () method on the thread object. The following code segment creates an object of the thread class and starts the thread object:

class Start Threadclass

{

  public static void main (String args [ ])

  {

      ……..

      ……..

      SampleThread st = new SampleThread ();

      st.start ();

      …….

  }

}

Here, the thread object st of the thread class SampleThread is created as

SampleThread st = new SampleThread ();

To start the thread object st, the start () method can be applied on this object as

st.start ();

When the above statement is executed, the run () method of the *SampleThread* class is invoked. The start () method implicitly calls the run () method. Note that the run () method can never be called directly.

**Creating a thread by implementing Runnable interface**

The interface Runnable is defined in the java.lang package. It has a single method-*run ().*

public interface Runnable

{

  public abstract void run();

}

If we want multithreading to be supported in a class that is already derived from a class other than Thread, we must implement the Runnable interface in that class.

The majority of classes created that need to be run as a thread will implement Runnable since they may be extending some other functionality from another class. Whenever the class defining *run ()* method needs to be the sub-class of classes other than Thread, using Runnable interface is the best way of creating threads. The syntax and the inheritance structure are given below.

public class SampleThread extends

OtherClass implements Runnable

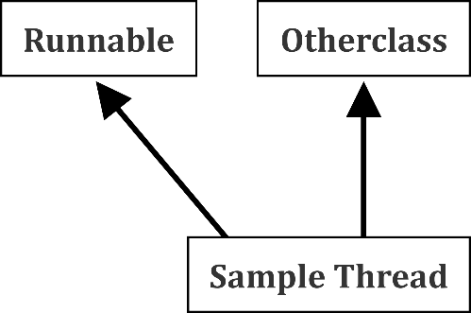
{

.....

public void run()

{

//code to be run when this thread is started}}



The class Thread itself implements the Runnable interface (package java.lang) as expressed in the class header:

public class Thread extends Object implements Runnable

As the Thread class implements Runnable interface, the code that controls the thread is placed in the run () method. In order to create a new thread with Runnable interface, we need to instantiate the Thread class. This thread class will have the following constructors:

public Thread (Runnable obj);

public Thread (Runnable obj, String threadname);

public Thread (ThreadGroup tg, Runnable obj, String threadname);

Here, obj is the object of the class which implements the Runnable interface, threadname is the name given to the thread and tg is the name of the ThreadGroup.

Following program illustrates the creation of threads using Runnable interface.

class ThreadExample implements Runnable

{

    Thread t;

    public ThreadExample (String threadname)

    {

        t = new Thread (this, threadname);

    }

    public void run ()

    {

        System.out.println (Thread.currentThread () );

        for (int i =0; i <=5; i++)

        System.out.println (i);

    }

}

public class ExampleT2

{

   public static void main (String args [ ])

    {

        ThreadExample obj = new ThreadExample ("First");

        Obj.t.start ( );

        System.out.println ("This is:" + Thread.currentThread ());

    }

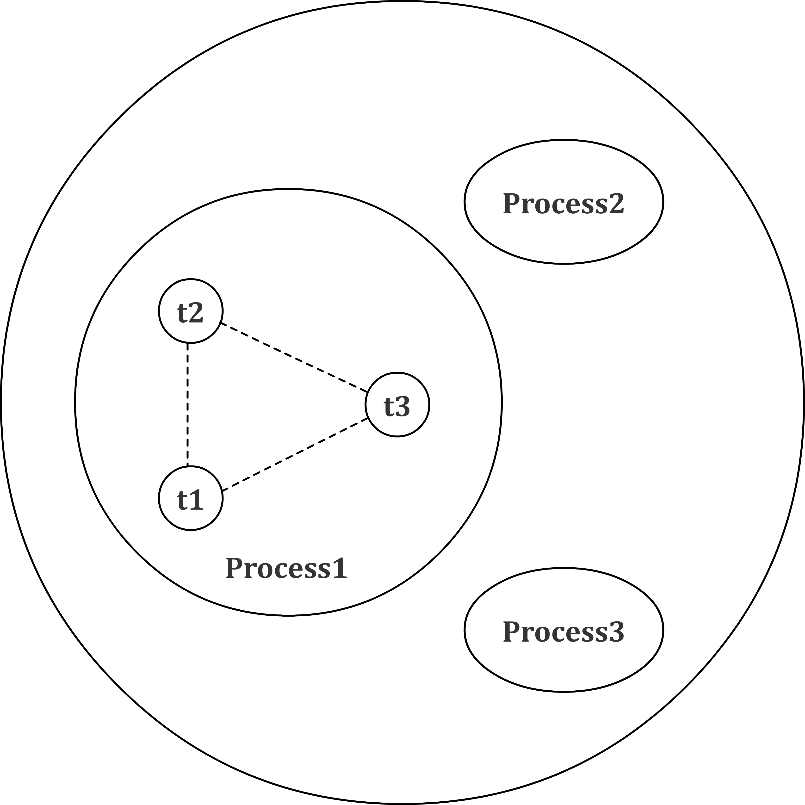
}

## Context Switching

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Context switching isn't cheap. You don't want to switch between threads more than necessary.

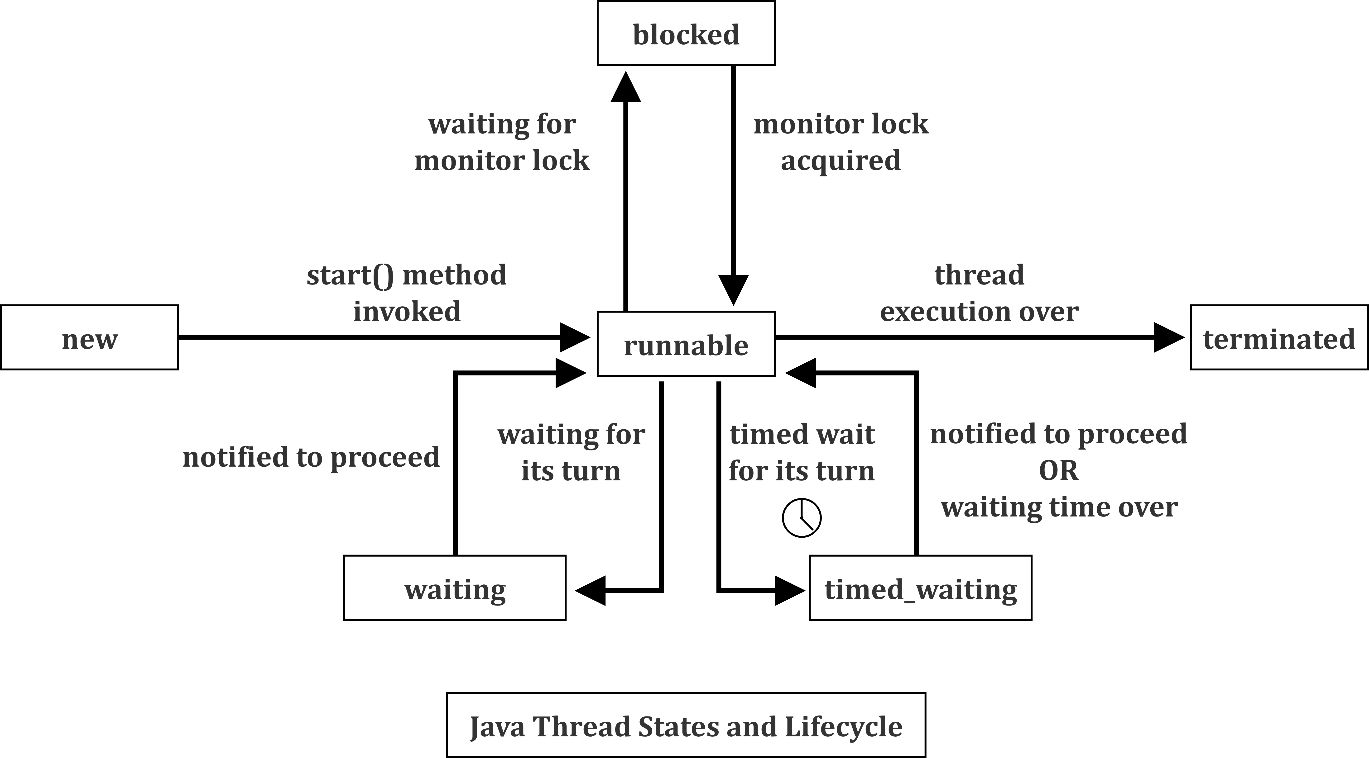


As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

## Thread States

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated



**New State:**

A thread is said to be in new state when we created the thread instance, but we have not yet called start() on the thread newly created thread instance. Even though it is a live thread object, it is not a thread of execution. At this state, thread is not active.

**Runnable State:**

A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

**Blocked/Waiting state:**

When a thread is temporarily inactive, then it’s in one of the following states:

* Blocked
* Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

**Timed Waiting:**

A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to the *timed waiting state*.

**Terminated State:**

A thread terminates because of either of the following reasons:

* Because it exists normally. This happens when the code of thread has entirely executed by the program.
* Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

## Commonly Used Methods

We have various methods which can be called on Thread class object. These methods are very useful when writing a multithreaded application. Thread class has following important methods.

**static Thread currentThread()**

This method returns a reference to the currently executing thread object.

**static void sleep()**

This method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

[**void setName(String name)**](https://www.tutorialspoint.com/java/lang/thread_setname.htm)

This method changes the name of this thread to be equal to the argument name.

**void setPriority(int newPriority)**

This method changes the priority of this thread.

**String getName()**

This method returns this thread's name.

**int getPriority()**

This method Returns this thread's priority.

## ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Java thread group is implemented by java.lang.ThreadGroup class.

**Constructors of ThreadGroup class**

There are only two constructors of ThreadGroup class.

|  |  |  |
| --- | --- | --- |
| No. | Constructor | Description |
| 1) | *ThreadGroup(String name)* | Creates a thread group with given name. |
| 2) | *ThreadGroup(ThreadGroup parent, String name)* | Creates a thread group with given parent group and name. |

## Important methods of ThreadGroup class

There are many methods in *ThreadGroup class*. A list of important methods are given below.

**int activeCount():** This method returns the number of threads in the group plus any group for which this thread is parent.

**Syntax:**

public int activeCount()

**Example of activeCount() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 1000; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Exception encounterted");

}

}

}

}

public class ThreadGroupDemo

{

public static void main(String arg[])

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("parent thread group");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting one");

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting two");

// checking the number of active thread

System.out.println("number of active thread: "

+ gfg.activeCount());

}

}

Output

Starting one

Starting two

number of active thread: 2

**int activeGroupCount():** This method returns an estimate of the number of active groups in this thread group.

**Syntax:**

public int activeGroupCount().

**Example of int activeGroupCount() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 1000; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Exception encounterted");

}

}

System.out.println(Thread.currentThread().getName() +

" finished executing");

}

}

public class ThreadGroupDemo

{

public static void main(String arg[]) throws InterruptedException

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("gfg");

ThreadGroup gfg\_child = new ThreadGroup(gfg, "child");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting one");

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting two");

// checking the number of active thread

System.out.println("number of active thread group: "

+ gfg.activeGroupCount());

}

}

Output

Starting one

Starting two

number of active thread group: 2

one finished executing

two finished executing

**void destroy():** Destroys the thread group and any child groups on which it is called.

**Syntax:**

public void destroy().

**Example of void destroy() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Exception encounterted");

}

}

}

}

public class ThreadGroupDemo

{

public static void main(String arg[]) throws InterruptedException,

SecurityException

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("Parent thread");

ThreadGroup gfg\_child = new ThreadGroup(gfg, "child thread");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting one");

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting two");

// block until other thread is finished

t1.join();

t2.join();

// destroying child thread

gfg\_child.destroy();

System.out.println(gfg\_child.getName() + " destroyed");

// destroying parent thread

gfg.destroy();

System.out.println(gfg.getName() + " destroyed");

}

}

Output

Starting one

Starting two

child thread destroyed

Parent thread destroyed

**String getName():** This method returns the name of the group.

**Syntax:**

final String getName().

**Example of void destroy() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Exception encounterted");

}

}

System.out.println(Thread.currentThread().getName() +

" finished executing");

}

}

public class ThreadGroupDemo

{

public static void main(String arg[]) throws InterruptedException,

SecurityException

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("Parent thread");

ThreadGroup gfg\_child = new ThreadGroup(gfg, "child thread");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting " + t1.getName());

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting " + t2.getName());

}

}

**Output**

Starting one

Starting two

two finished executing

one finished executing

**void interrupt():** Invokes the ***interrupt()*** methods of all threads in the group.

**Syntax:**

public final void interrupt().

**Example of void destroy() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Thread " + Thread.currentThread().getName()

+ " interrupted");

}

}

System.out.println(Thread.currentThread().getName() +

" finished executing");

}

}

public class ThreadGroupDemo

{

public static void main(String arg[]) throws InterruptedException,

SecurityException

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("Parent thread");

ThreadGroup gfg\_child = new ThreadGroup(gfg, "child thread");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting " + t1.getName());

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting " + t2.getName());

// interrupting thread group

gfg.interrupt();

}

}

Output

Starting one

Starting two

Thread two interrupted

Thread one interrupted

one finished executing

two finished executing

**void list():** Displays information about the group.

**Syntax:**

public void list().

**Example of void destroy() method:**

import java.lang.\*;

class NewThread extends Thread

{

NewThread(String threadname, ThreadGroup tgob)

{

super(tgob, threadname);

start();

}

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

Thread.sleep(10);

}

catch (InterruptedException ex)

{

System.out.println("Thread " + Thread.currentThread().getName()

+ " interrupted");

}

}

System.out.println(Thread.currentThread().getName() +

" finished executing");

}

}

public class ThreadGroupDemo

{

public static void main(String arg[]) throws InterruptedException,

SecurityException, Exception

{

// creating the thread group

ThreadGroup gfg = new ThreadGroup("Parent thread");

ThreadGroup gfg\_child = new ThreadGroup(gfg, "child thread");

NewThread t1 = new NewThread("one", gfg);

System.out.println("Starting " + t1.getName());

NewThread t2 = new NewThread("two", gfg);

System.out.println("Starting " + t2.getName());

// listing contents of parent ThreadGroup

System.out.println("\nListing parentThreadGroup: " + gfg.getName()

+ ":");

// prints information about this thread group

// to the standard output

gfg.list();

}

}

Output:

Starting one

Starting two

Listing parentThreadGroup: Parent thread:

java.lang.ThreadGroup[name=Parent thread, maxpri=10]

Thread[one, 5, Parent thread]

Thread[two, 5, Parent thread]

java.lang.ThreadGroup[name=child thread, maxpri=10]

one finished executing

two finished executing

## Thread Synchronization: Methods and Blocks

**Synchronized Methods:**

Synchronized methods are methods that are used to control access to an object. A thread only executes a synchronized method after it has acquired the lock for the method's object or class.   
  
Two invocations of synchronized methods cannot interleave on the same object.  
  
When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

To make a method synchronized, simply add the *synchronized* keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

**Let us see an example of synchronized method below**:

class Line

{

    synchronized public void getLine()

    {

        for (int i = 0; i < 3; i++)

        {

            System.out.println(i);

            try

            {

                Thread.sleep(400);

            }

            catch (Exception e)

            {

                System.out.println(e);

            }

        }

    }

}

class Train extends Thread

{

    // Reference variable of type Line.

    Line line;

    Train(Line line)

    {

        this.line = line;

    }

    // Override

    public void run()

    {

        line.getLine();

    }

}

class GFG

{

    public static void main(String[] args)

    {

        Line obj = new Line();

        // we are creating two threads which share

        // same Object.

        Train train1 = new Train(obj);

        Train train2 = new Train(obj);

        // both threads start executing.

        train1.start();

        train2.start();

    }

}

**Output**:

0

1

2

0

1

2

**Synchronized Blocks:**

Another way to create synchronized code is with synchronized blocks. Synchronized blocks are similar to synchronized methods. A synchronized block can only be executed after a thread has acquired the lock for the object or class referenced in the synchronized block. Synchronized blocks must specify the object that provides the intrinsic lock:

public void add(String str)  
{  
      synchronized(this)   
      {  
      str1 = str;  
      cnt++;  
      }  
     strList.add(str);  
}

**Let us see an example of synchronized block below**:

import java.io.\*;

import java.util.\*;

public class Geek

{

    String name = "";

    public int count = 0;

    public void geekName(String geek, List<String> list)

    {

        // Only one thread is permitted

        // to change geek's name at a time.

        synchronized(this)

        {

            name = geek;

            count++;  // how many threads change geek's name.

        }

        // All other threads are permitted

        // to add geek name into list.

        list.add(geek);

    }

}

class GFG

{

    public static void main (String[] args)

    {

        Geek gk = new Geek();

        List<String> list = new ArrayList<String>();

        gk.geekName("mohit", list);

        System.out.println(gk.name);

    }

}

Output

1

## ReentrantLock

On class level, ReentrantLock is a concrete implementation of *Lock* interface provided in Java concurrency package from Java 1.5 onwards.  As per Javadoc, ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by Thread until a call to unlock() method. Fairness parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

Let’s understand ReentrantLock with a simple example.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {

private static class Counter {

private ReentrantLock lock = new ReentrantLock();

private int number;

public Counter(int number) {

this.number = number;

}

public boolean dec() {

// the same as "synchronized(lockObject)":

lock.lock();

try {

if (number > 0) {

Thread.yield();

--number;

return true;

}

return false;

} finally {

// have to unlock explicitely:

lock.unlock();

}

}

}

private static class Worker implements Runnable {

private static int nth = 0;

private final int id = ++nth;

private Counter counter;

public Worker(Counter counter) {

this.counter = counter;

}

// Override

public void run() {

System.out.println("Starting worker: " + id);

while (counter.dec()) {

System.out.printf("Decreased to %2d by worker: %2d%n",

counter.number, id);

}

}

}

public static void main(String[] args) {

Counter counter = new Counter(10);

ExecutorService executor = Executors.newCachedThreadPool();

for (int i = 1; i <= 10; ++i) {

executor.execute(new Worker(counter));

}

executor.shutdown();

System.out.println("Number actually is: " + counter.number);

}

}

Output

Starting worker: 1

Starting worker: 2

Starting worker: 3

Starting worker: 4

Starting worker: 5

Starting worker: 6

Starting worker: 7

Starting worker: 8

Starting worker: 9

Starting worker: 10

Decreased to 9 by worker: 2

Decreased to 8 by worker: 10

Decreased to 7 by worker: 2

Decreased to 6 by worker: 10

Decreased to 5 by worker: 8

Decreased to 4 by worker: 1

Decreased to 3 by worker: 7

Decreased to 2 by worker: 5

Decreased to 1 by worker: 6

Decreased to 0 by worker: 4

Number actually is: 0

**Benefits of ReentrantLock in Java**

Most of the benefits offered by ReentrantLock over synchronized in Java are:

* 1. Ability to *lock interruptibly*.
  2. Ability to timeout while waiting for lock.
  3. Power to create fair lock.
  4. API to get list of waiting thread for lock.
  5. Flexibility to try for lock withAAout blocking.

**Disadvantages of Re-entrantLock in Java**

Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block, which makes code unreadable and hides business logic. It’s really cluttered and I hate it most, though IDE like Eclipse and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

Synchronized block are reentrant in nature i.e if a thread has lock on the monitor object and if another synchronized block requires to have the lock on the same monitor object then thread can enter that code block. This is the reason for the class name to be ReentrantLock.

## Inter Thread Communication

Inter-thread communication or co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of object class:

* wait()
* notify()
* notifyAll()

**1) wait() method**

Causes current thread to release the lock and wait until either another thread invokes the *notify()* method or the *notifyAll()* method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

|  |  |
| --- | --- |
| Method | Description |
| *public final void wait()throws InterruptedException* | It waits until object is notified. |
| *public final void wait(long timeout)throws InterruptedException* | It waits for the specified amount of time. |

**2) notify() method**

This method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax:**

public final void notify()

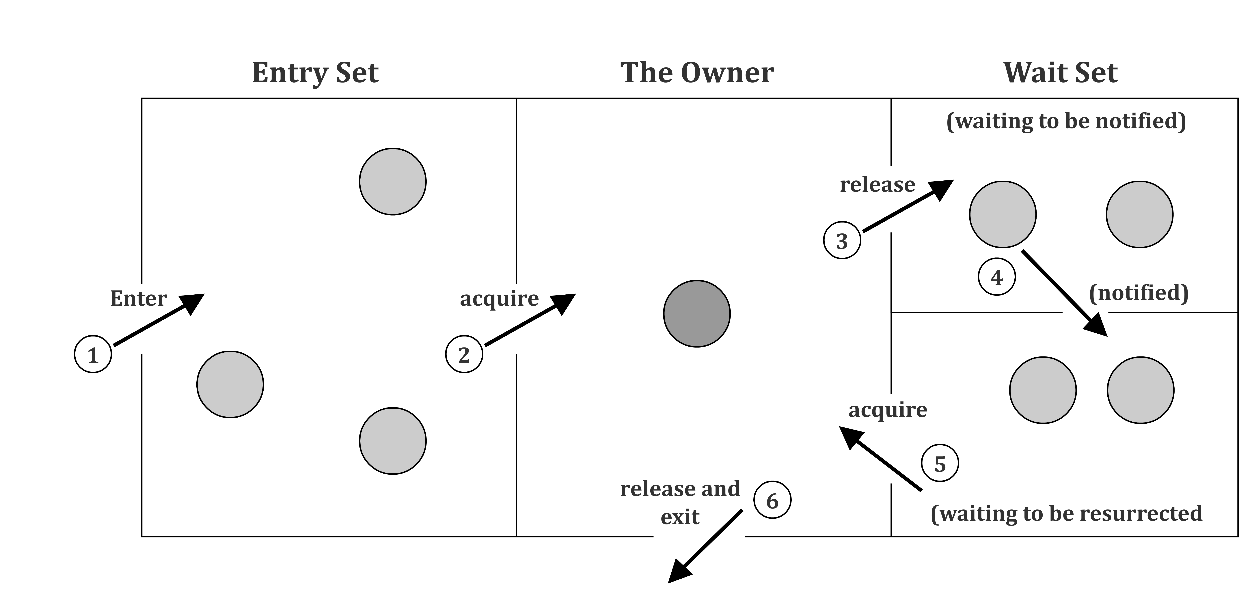
**3) notifyAll() method**

This method wakes up all threads that are waiting on this object's monitor.

**Syntax:**

public final void notifyAll()

**Understanding the process of inter-thread communication**



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

**Example of inter thread communication in java**

Let's see the simple example of inter thread communication.

class Customer{

int amount=10000;

synchronized void withdraw(int amount){

System.out.println("going to withdraw...");

if(this.amount<amount){

System.out.println("Less balance; waiting for deposit...");

try{wait();}catch(Exception e){}

}

this.amount-=amount;

System.out.println("withdraw completed...");

}

synchronized void deposit(int amount){

System.out.println("going to deposit...");

this.amount+=amount;

System.out.println("deposit completed... ");

notify();

}

}

class Test{

public static void main(String args[]){

final Customer c=new Customer();

new Thread(){

public void run(){c.withdraw(15000);}

}.start();

new Thread(){

public void run(){c.deposit(10000);}

}.start();

 }}

Output

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

## Daemon thread

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

**Properties:**

* They cannot prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
* It is an utmost low priority thread.

**Methods:**

1. **void setDaemon(boolean status):**

This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread *tU* then *tU.setDaemon(true)* would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling *tD.setDaemon(false)* would make it user thread.

**Syntax:**

public final void setDaemon(boolean on)

parameters:

on : if true, marks this thread as a daemon thread.

exceptions:

IllegalThreadStateException: if only this thread is active.

SecurityException: if the current thread cannot modify this thread.

1. **boolean isDaemon():**  
   This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.

**Syntax:**

public final boolean isDaemon()

returns:

This method returns true if this thread is a daemon thread, else returns false.

This example is to demonstrate the usage of *setDaemon()* and *isDaemon()* method.

// Java program to demonstrate the usage of

// setDaemon() and isDaemon() method.

public class DaemonThread extends Thread

{

public void run()

{

// Checking whether the thread is Daemon or not

if(Thread.currentThread().isDaemon())

{

System.out.println("This is Daemon thread");

}

else

{

System.out.println("This is User thread");

}

}

public static void main(String[] args)

{

DaemonThread t1 = new DaemonThread();

DaemonThread t2 = new DaemonThread();

DaemonThread t3 = new DaemonThread();

// Setting user thread t1 to Daemon

t1.setDaemon(true);

// starting all the threads

t1.start();

t2.start();

t3.start();

// Setting user thread t3 to Daemon

t3.setDaemon(true);

}

}

**This will give following output**:

This is Daemon thread

This is User thread

This is Daemon threaduser(normal) thread executing

## Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object

// Java program to illustrate Deadlock

// in multithreading.

class Util

{

// Util class to sleep a thread

static void sleep(long millis)

{

try

{

Thread.sleep(millis);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

// This class is shared by both threads

class Shared

{

// first synchronized method

synchronized void test1(Shared s2)

{

System.out.println("test1-begin");

Util.sleep(1000);

// taking object lock of s2 enters

// into test2 method

s2.test2(this);

System.out.println("test1-end");

}

// second synchronized method

synchronized void test2(Shared s1)

{

System.out.println("test2-begin");

Util.sleep(1000);

// taking object lock of s1 enters

// into test1 method

s1.test1(this);

System.out.println("test2-end");

}

}

class Thread1 extends Thread

{

private Shared s1;

private Shared s2;

// constructor to initialize fields

public Thread1(Shared s1, Shared s2)

{

this.s1 = s1;

this.s2 = s2;

}

// run method to start a thread

@Override

public void run()

{

// taking object lock of s1 enters

// into test1 method

s1.test1(s2);

}

}

class Thread2 extends Thread

{

private Shared s1;

private Shared s2;

// constructor to initialize fields

public Thread2(Shared s1, Shared s2)

{

this.s1 = s1;

this.s2 = s2;

}

// run method to start a thread

@Override

public void run()

{

// taking object lock of s2

// enters into test2 method

s2.test2(s1);

}

}

public class GFG

{

public static void main(String[] args)

{

// creating one object

Shared s1 = new Shared();

// creating second object

Shared s2 = new Shared();

// creating first thread and starting it

Thread1 t1 = new Thread1(s1, s2);

t1.start();

// creating second thread and starting it

Thread2 t2 = new Thread2(s1, s2);

t2.start();

// sleeping main thread

Util.sleep(2000);

}

}

**This will give following output:**

test1-begin

test2-begin

Now let’s see step by step what is happening there.

* Thread t1 starts and calls test1 method by taking the object lock of s1.
* Thread t2 starts and calls test2 method by taking the object lock of s2.
* t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
* t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
* Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
* Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
* As none of them is ready to release lock, so this is the Dead Lock condition.
* When you will run this program, it will be look like execution is paused.

**More Complicated Deadlocks**

Deadlock can also include more than two threads. This makes it harder to detect. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

## Java Shutdown Hook

The shutdown hook can be used to perform clean-up resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

**When does the JVM shut down?**

The JVM shuts down when:

* User presses ctrl+c on the command prompt
* System.exit(int) method is invoked
* User logoff
* User shutdown etc.

**The addShutdownHook(Thread hook) method**

The *addShutdownHook()* method of *Runtime* class is used to register the thread with the Virtual Machine.

**Syntax:**

public void addShutdownHook(Thread hook){}

The object of Runtime class can be obtained by calling the static factory method *getRuntime()*. For example:

Runtime r = Runtime.getRuntime();

**Factory Method**

The method that returns the instance of a class is known as **factory method**.

**Simple example of Shutdown Hook**

class MyThread extends Thread{

    public void run(){

        System.out.println("shut down hook task completed..");

    }

}

public class TestShutdown1{

public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new MyThread());

System.out.println("Now main sleeping... press ctrl+c to exit");

try{Thread.sleep(3000);}catch (Exception e) {}

}

}

**Output**:

Now main sleeping... press ctrl+c to exit

shut down hook task completed..

**Note that:**

1. Shutdown Hooks may not be executed in some cases!
2. Once started, Shutdown Hooks can be forcibly stopped before completion.
3. We can have more than one Shutdown Hooks, but their execution order is not guaranteed.
4. We cannot register / unregister Shutdown Hooks with in Shutdown Hooks
5. Once shutdown sequence starts, it can be stopped by Runtime.halt() only.
6. Using shutdown hooks require security permissions.

## Summary

* Shutdown Hooks may not be executed in some cases
* Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block
* In Java, threads are objects and can be created in two ways: by extending the class Thread and by implementing the interface Runnable
* The method that returns the instance of a class is known as factory method
* Inter-thread communication is all about allowing synchronized threads to communicate with each other
* Synchronized methods are methods that are used to control access to an object
* Java supports messaging which provides methods to all objects for inter-thread communication
* Fairness parameter is provided while creating instance of ReentrantLock in constructor

Chapter 11 – I/O Streams

What is I/O

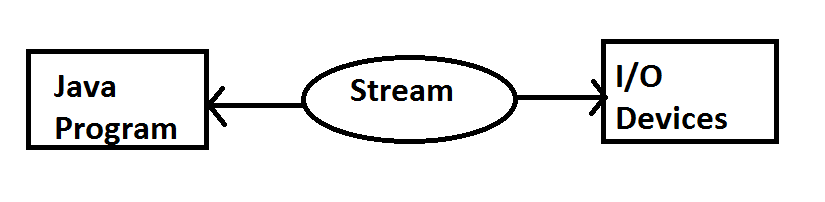
Input and Output (I/O) is used to process the input and produce the output.

In Java I/O, a program reads inputs from data sources (e.g., keyboard, file, network, memory buffer etc.) and writes outputs to data sinks (e.g., display console, file etc.).

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Why Need Streams

Java performs I/O through streams to improve efficiency of read/write operations. A stream is linked to a physical layer by Java I/O system to make input and output operation in java. Generally, a stream means a continuous flow of data. Streams are a clean way to deal with input/output without having every part of the code understand the hardware.

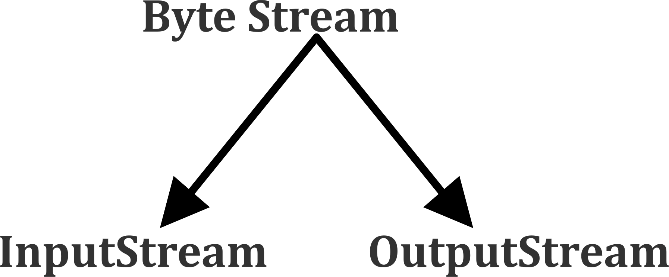


Java encapsulates stream using*java.io* package. Java defines two types of streams. They are:

1. **Byte Stream:** It provides a convenient means to handle input and output of byte.
2. **Character Stream:** It offers a convenient means to handle input and output of characters. The character stream uses Unicode and thus can be internationalized.

Byte Stream Classes

The byte stream is defined by using two abstract class at the top of the hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc. This classes read and write file byte by byte.

**Some important Byte stream classes are shown in the table below:**

|  |  |
| --- | --- |
| Stream class | Description |
| *BufferedInputStream* | Used for Buffered Input Stream |
| *BufferedOutputStream* | Used for Buffered Output Stream |
| *DataInputStream* | Contains method for reading java standard datatype |
| *DataOutputStream* | Contains a method for writing java standard data type |
| *FileInputStream* | An input stream that reads from a file |
| *FileOutputStream* | An output stream that writes to a file |
| *InputStream* | Abstract class that describes stream input |
| *OutputStream* | Abstract class that describes stream output |
| *PrintStream* | Output stream that contains print() and println() method |

These classes define several key methods. Two most important key methods are:

1. *read()* : reads byte of data
2. *write()* : Writes byte of data

A program of byte stream that shows reading and writing file byte by byte is given below:

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyBytes {

public static void main(String[] args) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("inputfile.txt");

out = new FileOutputStream("outputfile.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

Character Stream Classes

Character stream is also defined by using two abstract class at the top of a hierarchy, they are Reader and Writer. It reads and writes character by character instead of each byte.



These two abstract classes have several concrete classes that handle Unicode character.

A program of character stream that shows reading and writing of a file character by character is shown below:

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CopyCharacters {

public static void main(String[] args) throws IOException {

FileReader inputStream = null;

FileWriter outputStream = null;

try {

inputStream = new FileReader("xanadu.txt");

outputStream = new FileWriter("characteroutput.txt");

int c;

while ((c = inputStream.read()) != -1) {

outputStream.write(c);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

**Some important Character stream classes are:**

|  |  |
| --- | --- |
| *Stream class* | Description |
| *BufferedReader* | Handles buffered input stream |
| *BufferedWriter* | Handles buffered output stream |
| *FileReader* | Input stream that reads from file |
| *FileWriter* | Output stream that writes to file |
| *InputStreamReader* | An input stream that translates byte to character |
| *OutputStreamReader* | An output stream that translates character to byte |
| *PrintWriter* | Output stream that contains print() and println() method |
| *Reader* | Abstract class that defines character stream input |
| *Writer* | Abstract class that defines character stream output |

Reader/Write Operations with File

Java FileWriter and FileReader classes are used to write and read data from text files (they are Character Stream classes). It is recommended not to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information since these are Byte stream classes.

**FileWriter**

FileWriter is suitable to create a file writing characters into it.

* This class inherits object from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values, you can construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant to write streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.

**Constructors:**

* **FileWriter (File file) –** Constructs a FileWriter object given a File object.
* **FileWriter (File file, boolean append) –** Constructs a FileWriter object given a File object with an option to open the file in append mode.
* **FileWriter (FileDescriptor fd) –** Constructs a FileWriter object associated with a file descriptor.
* **FileWriter (String fileName) –** Constructs a FileWriter object given a file name.
* **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

**Methods:**

* **public void write (int c) throws IOException –** Writes a single character.
* **public void write (char [] stir) throws IOException –** Writes an array of characters.
* **public void write(String str)throws IOException –** Writes a string.
* **public void writes (String str,int off,int len)throws IOException –** Writes a portion of a string. Here *off* is offset from which to start writing characters and *len* is a number of character to write.

Reading and writing take place character by character, which increases the number of I/O operations and affects the performance of the system. *BufferedWriter* can be used along with FileWriter to improve the speed of execution.

Following program depicts how to create a text file using FileWriter character by character.

|  |
| --- |
| // Creating a text File using FileWriter  import java.io.FileWriter;  import java.io.IOException;  class CreateFile  {      public static void main(String[] args) throws IOException      {          // Accept a string          String str = "File Handling in Java using "+                       " FileWriter and FileReader";            // attach a file to FileWriter          FileWriter fw=new FileWriter("text");            // read character wise from string and write          // into FileWriter          for (int i = 0; i < str.length(); i++)              fw.write(str.charAt(i));            //close the file          fw.close();      }  }  **BufferedWriter**  This is used to write file line by line. We use *BufferedWriter* to implement this. Below is program that shows writing of file line by line:    import java.io.BufferedWriter;  import java.io.FileWriter;  import java.io.IOException;    /\*\*   \* This program demonstrates how to write characters to a text file   \* using a BufferedReader for efficiency.   \*   \*   \*/  public class TextFileWritingExample2 {        public static void main(String[] args) {          try {              FileWriter writer = new FileWriter("MyFile.txt", true);              BufferedWriter bufferedWriter = new BufferedWriter(writer);                bufferedWriter.write("Hello World");              bufferedWriter.newLine();              bufferedWriter.write("See You Again!");                bufferedWriter.close();          } catch (IOException e) {              e.printStackTrace();          }        }    }  **FileOutputStream**  FileOutputStream to write binary data to a file. The following code converts a String int bytes and writes the bytes to file using a FileOutputStream:  import java.io.FileOutputStream;  public class FileOutputStreamExample {  public static void main(String args[]){  try{  FileOutputStream fout=new FileOutputStream("D:\\testfile.txt");  fout.write(65);  fout.close();  System.out.println("success...");  }catch(Exception e){System.out.println(e);}  }  }  The output will be:  Success  **DataOutputStream**  DataOutputStream is used to write a String to file. Below is the program of DataOutputStream.  import java.io.\*;  public class OutputExample {  public static void main(String[] args) throws IOException {  FileOutputStream file = new FileOutputStream(D:\\testfile.txt);  DataOutputStream data = new DataOutputStream(file);  data.writeInt(65);  data.flush();  data.close();  System.out.println("Successful");  }  }  The output will be:  Successful |
|  |

**FileReader**

FileReader is useful to read data in the form of characters from a ‘text’ file.

* This class inherit from the InputStreamReader Class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

**Constructors:**

* **FileReader(File file) –**Creates a FileReader , given the File to read from
* **FileReader(FileDescripter fd) –** Creates a new FileReader , given the FileDescripter to read from
* **FileReader(String fileName) –**Creates a new FileReader , given the name of the file to read from

**Methods:**

* **public int read () throws IOException –** Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
* **public int read(char[] cbuff) throws IOException –** Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
* **public abstract int read(char[] buff, int off, int len) throws IOException –**Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.  
  Parameters:  
  cbuf – Destination buffer  
  off – Offset at which to start storing characters  
  len – Maximum number of characters to read
* **public long skip(long n) throws IOException –**Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.

Following program depicts how to read from the ‘text’ file using FileReader character by character.

|  |  |
| --- | --- |
| // Reading data from a file using FileReader  import java.io.FileNotFoundException;  import java.io.FileReader;  import java.io.IOException;  class ReadFile  {      public static void main(String[] args) throws IOException      {          // variable declaration          int ch;            // check if File exists or not          FileReader fr=null;          try          {              fr = new FileReader("text");          }          catch (FileNotFoundException fe)          {              System.out.println("File not found");          }            // read from FileReader till the end of file          while ((ch=fr.read())!=-1)              System.out.print((char)ch);            // close the file          fr.close();      }  }  **BufferedReader**  This is used to write file line by line. We use *BufferedReader* to implement this. Below is program that shows reading of file line by line:    import java.io.BufferedReader;  import java.io.FileReader;  import java.io.IOException;    /\*\*   \* This program demonstrates how to read characters from a text file   \* using a BufferedReader for efficiency.   \*   \*   \*/  public class TextFileReadingExample3 {        public static void main(String[] args) {          try {              FileReader reader = new FileReader("MyFile.txt");              BufferedReader bufferedReader = new BufferedReader(reader);                String line;                while ((line = bufferedReader.readLine()) != null) {                  System.out.println(line);              }              reader.close();            } catch (IOException e) {              e.printStackTrace();          }      }    }  **Scanner class**  A simple text scanner which can parse primitive types and strings using regular expressions. A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.   |  | | --- | | // Java Program to illustrate reading from Text File  // using Scanner Class  import java.io.File;  import java.util.Scanner;  public class ReadFromFileUsingScanner  {    public static void main(String[] args) throws Exception    {      // pass the path to the file as a parameter      File file =        new File("C:\\Users\\pankaj\\Desktop\\test.txt");      Scanner sc = new Scanner(file);        while (sc.hasNextLine())        System.out.println(sc.nextLine());    }  } | |

Scanner Class in Java

The scanner is a class in *java.util* package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

* To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
* To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
* To read strings, we use nextLine().
* To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

**Commonly used methods of scanner class**

There is a list of commonly used scanner class methods:

|  |  |
| --- | --- |
| Method | Description |
| *public String next()* | It returns the next token from the scanner. |
| *public String nextLine()* | It moves the scanner position to the next line and returns the value as a string. |
| *public byte nextByte()* | It scans the next token as a byte. |
| *public short nextShort()* | It scans the next token as a short value. |
| *public int nextInt()* | It scans the next token as an int value. |
| *public long nextLong()* | It scans the next token as a long value. |
| *public float nextFloat()* | It scans the next token as a float value. |
| *public double nextDouble()* | It scans the next token as a double value. |

Let us look at the code snippet to read data of various data types.

|  |
| --- |
| // Java program to read data of various types using Scanner class.  importjava.util.Scanner;  publicclassScannerDemo1  {      publicstaticvoidmain(String[] args)      {          // Declare the object and initialize with          // predefined standard input object          Scanner sc = newScanner(System.in);           // String input          String name = sc.nextLine();           // Character input          chargender = sc.next().charAt(0);           // Numerical data input          // byte, short and float can be read          // using similar-named functions.          intage = sc.nextInt();          longmobileNo = sc.nextLong();          doublecgpa = sc.nextDouble();           // Print the values to check if input was correctly obtained.          System.out.println("Name: "+name);          System.out.println("Gender: "+gender);          System.out.println("Age: "+age);          System.out.println("Mobile Number: "+mobileNo);          System.out.println("CGPA: "+cgpa);      }  } |

**Input:**

Geeta

F

40

9876543210

9.9

**Output:**

Name: Geeta

Gender: F

Age: 40

Mobile Number: 9876543210

CGPA: 9.9

Object Serialization and Deserialization

Serialization in Java is a process in which the object’s state is converted into a byte stream while deserialization is just the reverse of serialization in which we use the byte stream to convert into the original state of a Java object.



Object Serialization and Deserialization

**Implementing Serialization and Deserialization in Java**

In Java Object Serialization is implemented with the help of ObjectInput and ObjectOutput interfaces, which extends DataInput and DataOutput interfaces, respectively. ObjectOutputStream class and the ObjectInputStream class implement the ObjectOutput interface and the ObjectInput interface, respectively. These two classes provides some set of methods which helps in writing and reading binary form of Object and Primitive values (int,long,double etc.). Below are the set of methods which are provided by below classes:

**1. ObjectOutputStream:**

* writeInt(i)
* writeLong(l)
* writeShort(s)
* writeChars(str)
* writeUTF(str)
* writeBoolean(b)
* writeByte(i)
* writeChar(i)
* writeDouble(d)
* writeFloat(f)
* writeObject(o)

ObjectOutputStream class can write object to any stream which extends Outputstream class. ObjectOutputStream class is used in serialization process.

**2. ObjectInputStream:**

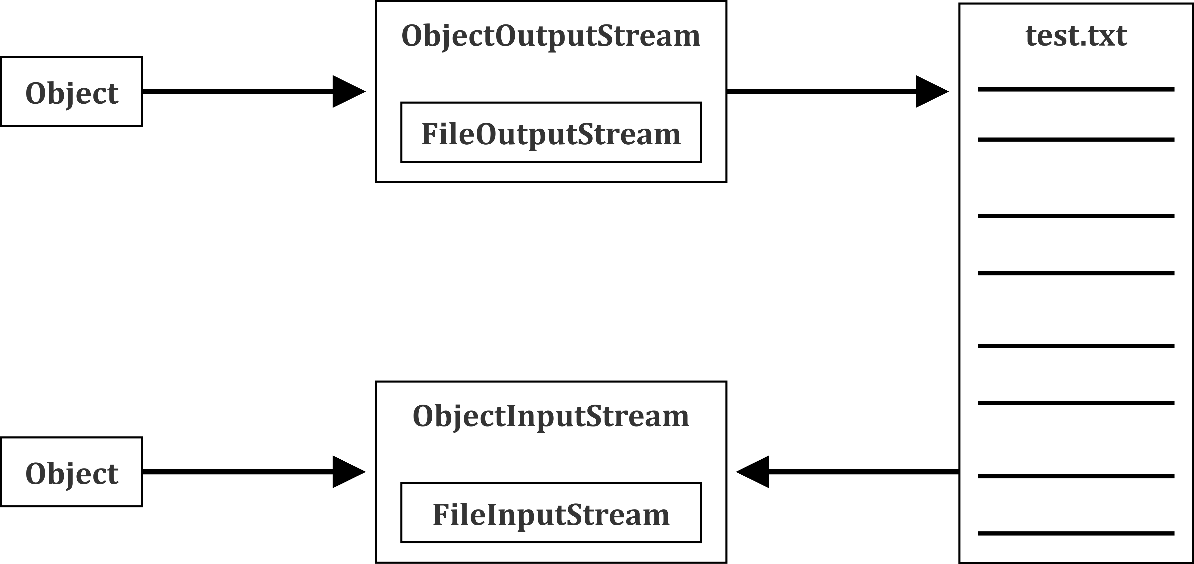
* readInt()
* readLong()
* readShort()
* readLine()
* readUTF()
* readBoolean()
* readByte()
* readChar()
* readDouble()
* readFloat()
* readObject()

ObjectInputStream class can read bytes and converts them in to object so it plays role in Deserialization process.

Above read and write methods are the method of Abstract Classes OutputStream and InputStream classes. These methods can throw IOException and EOFException. EOFEception is thrown if the end of the stream is reached.

**Serialization and Deserialization in a File**

Let’s understand Serialization and Deserialization with the help of a file. Suppose we have a file test.txt. We will store an Object in this file in form of bytes and then we recreate that object from the same file.



Serialization and Deserialization with the help of a file

**Example of Serialization**

import java.io.\*;

public class ObjectSerialization

{

public static void main(String[] args)

{

try

{

Student s=new Student();

s.name="sonu";

s.age=20;

FileOutputStream fout=new FileOutputStream("e:/obj.txt");

ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s);

System.out.println("serialization done");

out.close();

fout.close();

}

catch(Exception e)

{

System.out.println(e);

}

}

}

**Example of Deserialization**

import java.io.\*;

public class ObjectDeserialization

{

public static void main(String[] args)

{

try

{

FileInputStream fin=new FileInputStream("e:/obj.txt");

ObjectInputStream oin=new ObjectInputStream(fin);

Student o=(Student)oin.readObject();

System.out.println(o.name);

System.out.println(o.age);

oin.close();

fin.close();

}

catch(Exception e)

{

System.out.println(e);

}

}

}

*transient* Keyword

Transient is a variables modifier used in serialization. At the time of serialization, if we don’t wish to save the value of a particular variable in a file, we use a *transient* keyword. During execution, when JVM comes across *transient*keyword, it ignores the original value of the variable and saves default value of that variable data type.

The *transient* keyword plays an important role to meet security constraints. There are various real-life examples where we don't want to save private data in the file. Another use of *transient*keyword is not to serialize the variable whose value can be calculated or derived using other serialized objects such as the age of a person, current date, etc.

Practically we can serialize only those fields which represent a state of the instance after all serialization is all about to save the state of an object to a file. It is a good habit to use the *transient* keyword with private confidential fields of a class during serialization.

Let’s check a sample class that uses transient keyword

|  |
| --- |
| // A sample class that uses transient keyword to  // skip their serialization.  classTest implementsSerializable  {      // Making password transient for security      privatetransientString password;        // Making age transient as age is auto-  // computable from DOB and the current date.      transientintage;        // serialize other fields      privateString username, email;      Date dob;        // other code  } |

Externalization

Externalization is nothing but the serialization by implementing an Externalizable interface to persist and restore the object. To externalize your object, you need to implement an Externalizable interface that extends the Serializable interface. Here just the class identity is written in the serialization stream and the class is responsible to save and restore the contents of its instances. This means that you have complete control over what to serialize and what not to serialize.

**The Externalizable Interface**

Unlike Serializable interface, Externalizable interface is not a marker interface. It provides two methods i.e. writeExternal and readExternal. These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of the writeObject and readObject methods.

**Implementing writeExternal() method**

As the writeExternal() method takes an ObjectOutput,we can use its method to write object’s states into the underlying stream.

This is implemented by following these rules:

* For primitive types, use the writeXXX() methods of the DataOutput interface, such as writeBoolean(), writeByte(), writeInt(), writeLong(), etc.
* For object types (Strings, arrays or custom classes), use the writeObject() method.

Following the above rules, we implement the writeExternal() method of the user class like the following code:

publicvoidwriteExternal(ObjectOutput out) throwsIOException {

    out.writeInt(code);

    out.writeObject(name);

    // write empty password:

    out.writeObject("");

    out.writeObject(birthday);

}

In the above code, we have serialized the attributes: code, name, password and birthday. For security purpose, the password is cleared. This gives you the ideas of how we can control the process of serialization by implementing the Externalizable interface.

**Implementing readExternal() method**

As the readExternal() method takes an ObjectInput, we can use its method to read object’s states from the underlying stream by following these rules:

* For primitive types, use the readXXX() methods of the DataInput interface, such as readBoolean(), readByte(), readInt(), readLong(), etc.
* For object types (Strings, arrays, or custom classes), use the readObject() method

Following the above rules, we implement the readExternal()method of the user class like the following code:

publicvoidreadExternal(ObjectInput in) throwsClassNotFoundException, IOException {

    this.code = in.readInt();

    this.name = (String) in.readObject();

    this.password = (String) in.readObject();

    this.birthday = (Date) in.readObject();

}

In the above code, we de-serialized the attributes: code, name, password and birthday. This gives you the ideas of how we can control the process of de-serialization by implementing the Externalizable interface.

File Class and Its Methods

The File class is Java’s representation of a file or directory path name. Since the file and directory names have different formats on different platforms, a simple string is not sufficient to name them. The File class contains numerous methods to work with the path name, delete and rename files, create new directories, list the contents of a directory, and determine various common attributes of files and directories.

**How to create a File Object?**

A File object is created by passing in a String, which represents the name of a file, or a String or another File object. For example,

File a = new File ("/usr/local/bin/Ducat");

defines an abstract file name for the Ducat file in directory /usr/local/bin. This is an absolute abstract file name.

**Constructors**

* **File(File parent, String child):**Creates a new File instance from a parent abstract pathname and a child pathname string.
* **File(String pathname):**Creates a new File instance by converting the given pathname string into an abstract pathname.
* **File(String parent, String child):**Creates a new File instance from a parent pathname string and a child pathname string.
* **File(URI uri):**Creates a new File instance by converting the given file: URI into an abstract pathname.

**Methods**

* **boolean canExecute():** Tests whether the application can execute the file denoted by this abstract pathname.
* **boolean canRead()**: Tests whether the application can read the file denoted by this abstract pathname.
* **boolean canWrite():** Tests whether the application can modify the file denoted by this abstract pathname.
* **int compareTo(File pathname):** Compares two abstract pathnames alphabetically.
* **boolean createNewFile() :**Creates a new, empty file named by this abstract pathname .
* **static File createTempFile(String prefix, String suffix):** Creates an empty file in the default temporary-file directory.
* **boolean delete():** Deletes the file or directory denoted by this abstract pathname.
* **boolean equals(Object obj):** Tests this abstract pathname for equality with the given object.
* **boolean exists()**: Tests whether the file or directory denoted by this abstract pathname exists.
* **String getAbsolutePath() :**Returns the absolute pathname string of this abstract pathname.
* **long getFreeSpace() :**Returns the number of unallocated bytes in the partition.
* **String getName():** Returns the name of the file or directory denoted by this abstract pathname.
* **String getParent():** Returns the pathname string of this abstract pathname’s parent.
* **File getParentFile() :**Returns the abstract pathname of this abstract pathname’s parent.
* **String getPath():** Converts this abstract pathname into a pathname string.
* **boolean isDirectory():** Tests whether the file denoted by this pathname is a directory.
* **boolean isFile():** Tests whether the file denoted by this abstract pathname is a normal file.
* **boolean isHidden():** Tests whether the file named by this abstract pathname is a hidden file.
* **long length():** Returns the length of the file denoted by this abstract pathname.
* **String[] list():** Returns an array of strings naming the files and directories in the directory.
* **File[] listFiles() :**Returns an array of abstract pathnames denoting the files in the directory.
* **boolean mkdir():** Creates the directory named by this abstract pathname.
* **boolean renameTo(File dest):** Renames the file denoted by this abstract pathname.
* **boolean setExecutable(boolean executable):** A convenience method to set the owner’s execute permission.
* **boolean setReadable(boolean readable):** A convenience method to set the owner’s read permission.
* **boolean setReadable(boolean readable, boolean ownerOnly):** Sets the owner’s or everybody’s read permission.
* **boolean setReadOnly():** Marks the file or directory named so that only read operations are allowed.
* **boolean setWritable(boolean writable)**: A convenience method to set the owner’s write permission.
* **String toString():** Returns the pathname string of this abstract pathname.
* **URI toURI():** Constructs a file URI that represents this abstract pathname.

The following programs are the different examples of File class:

**Get Details of a File**

We can get details of file such as file size, file modified, can read status, can write status, etc. Below program demonstrate this:

import java.io.\*;

class FileDetails

{

public static void main(String[] args)

{

File file = new File("C:\\Java\\abc.txt");

System.out.println("Size of File is : "+file.length()+"bytes");

System.out.println("Can Read File : "+file.canRead());

System.out.println("Can Write to File : "+file.canWrite());

System.out.println("File name is : "+file.getName());

System.out.println("File last modified on : "+ new java.util.Date(file.lastModified()));

}

}

The output of the program will be:

Size of File is : 5 bytes

Can Read File : true

Can Write to File : true

File name is : abc.txt

File last modified on : Thu Jun 28 13:01:43 IST 2018

**Get all files and folders of a directory**

We can get all the files of directory as in given example:

import java.io.File;

public class Filedirect

{

     static void RecursivePrint(File[] arr,int index,int level)

     {

         // terminate condition

         if(index == arr.length)

             return;

         // tabs for internal levels

         for (int i = 0; i < level; i++)

             System.out.print("\t");

         // for files

         if(arr[index].isFile())

             System.out.println(arr[index].getName());

         // for sub-directories

         else if(arr[index].isDirectory())

         {

             System.out.println("[" + arr[index].getName() + "]");

             // recursion for sub-directories

             RecursivePrint(arr[index].listFiles(), 0, level + 1);

         }

         // recursion for main directory

         RecursivePrint(arr,++index, level);

    }

    // Driver Method

    public static void main(String[] args)

    {

        // Provide full path for directory(change accordingly)

        String maindirpath = "C:\\Java";

        // File object

        File maindir = new File(maindirpath);

        if(maindir.exists() && maindir.isDirectory())

        {

            // array for files and sub-directories

            // of directory pointed by maindir

            File arr[] = maindir.listFiles();

            System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

            System.out.println("Files from main directory : " + maindir);

            System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

            // Calling recursive method

            RecursivePrint(arr,0,0);

       }

    }

}

The output of this program will be

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Files from main directory : C:\Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Abc.txt

Extra-Items.pdf

Hello.class

Hello.java

[Xtra files]

A.docx

C.docx

**Get all Details about Drive**

We can get the details of Drive like free space, available space, etc as shown in below program:

import java.io.File;

import javax.swing.filechooser.FileSystemView;

public class DrivesListingExample {

    public static void main(String[] args) {

        FileSystemView fsv = FileSystemView.getFileSystemView();

        File[] drives = File.listRoots();

        if (drives != null && drives.length > 0) {

            for (File aDrive : drives) {

                System.out.println("Drive Letter: " + aDrive);

                System.out.println("\tType: " + fsv.getSystemTypeDescription(aDrive));

                System.out.println("\tTotal space: " + aDrive.getTotalSpace());

                System.out.println("\tFree space: " + aDrive.getFreeSpace());

                System.out.println();

            }

        }

    }

}

The output will be:

Drive Letter: C:\

    Type: Local Disk

    Total space: 73402363904

    Free space: 11994337280

Drive Letter: D:\

    Type: Local Disk

    Total space: 106151542272

    Free space: 84617833472

Drive Letter: E:\

    Type: Local Disk

    Total space: 106232282624

    Free space: 70415875072

Drive Letter: F:\

    Type: CD Drive

    Total space: 0

    Free space: 0

**Delete a File from a Folder**

We can delete a file from a folder. The following example demonstrate the deleting of a file:

import java.io.File;

public class DeleteFileJava {

public static void main(String[] args) {

//absolute file name with path

File file = new File("/Users/Ducat/file.txt");

if(file.delete()){

System.out.println("/Users/Ducat/file.txt File deleted");

}else System.out.println("File /Users/Ducat /file.txt doesn't exists");

//file name only

file = new File("file.txt");

if(file.delete()){

System.out.println("file.txt File deleted from Project root directory");

}else System.out.println("File file.txt doesn't exists in project root directory");

//relative path

file = new File("tmp/file.txt");

if(file.delete()){

System.out.println("tmp/file.txt File deleted from Project root directory");

}else System.out.println("File tmp/file.txt doesn't exists in project root directory");

//delete empty directory

file = new File("tmp");

if(file.delete()){

System.out.println("tmp directory deleted from Project root directory");

}else System.out.println("tmp directory doesn't exists or not empty in project root directory");

//try to delete directory with files

file = new File("/Users/Ducat /project");

if(file.delete()){

System.out.println("/Users/Ducat/project directory deleted from Project root directory");

}else System.out.println("/Users/Ducat/project directory doesn't exists or not empty");

}

}

The output of the program will be:

/Users/Ducat/file.txt File deleted

file.txt File deleted from Project root directory

tmp/file.txt File deleted from Project root directory

tmp directory deleted from Project root directory

/Users/Ducat/project directory doesn't exists or not empty

Summary

* Java FileWriter and FileReader classes are used to write and read data from text files
* The byte stream is defined by using two abstract classes InputStream and OutputStream
* A stream is linked to a physical layer by Java I/O system to make input and output operation in java
* Unlike Serializable interface, Externalizable interface is not a marker interface
* The scanner is a class in *java.util* package used for obtaining the input of the primitive types like int, double, and strings
* The *transient* keyword plays an important role to meet security constraints
* A File object is created by passing in a String that represents the name of a file, or a String or another File object
* FileWriter is suitable to create a file writing characters into it

Chapter 12 – Socket Programming

Fundamentals of a Network

A network is a group of two or more computer systems or other devices that are linked together to exchange data. Networks share resources, exchange files and electronic communications. For example, networked multiple computers can share the files and printers on the network.

**Java Networking Terminology**

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket
7. **IP Address**: IP address is a unique number assigned to a node of a network example 192.168.0.1 . It is composed of octets that range from 0 to 255. It is a logical address that can be changed.
8. **Protocol**: The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications. The port number is associated with the IP address for communication between two applications.
9. **Port Number**: The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications. The port number is associated with the IP address for communication between two applications.
10. **MAC Address**: MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.
11. **Connection-oriented and connection-less protocol**: In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP (Transmission Control Protocol). But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP (User Datagram Protocol).
12. **Socket**: A socket is an endpoint between two way communication.

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE. Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket Classes

*Socket* and *ServerSocket* classes are used for connection-oriented socket programming and *DatagramSocket* and *DatagramPacket* classes are used for connection-less socket programming.

**Socket Class**

A socket is simply an endpoint for communications between the machines. The socket class can be used to create a socket.

**Class Methods**

Below are the methods used in the socket class:

|  |  |
| --- | --- |
| Method | Description |
| *public InputStream getInputStream()* | Returns the *inputstream* attached with this socket |
| *public OutputStream getOutputStream()* | Returns the *outputstream* attached with this socket |
| *public synchronized void close()* | Closes this socket |

**ServerSocket Class**

The *ServerSocket* class can be used to create a server socket. This object is used to establish communication with the clients.

**Class Methods**

Below are the methods used in ServerSocket class:

|  |  |
| --- | --- |
| Method | Description |
| *public Socket accept()* | Returns the socket and establish a connection between server and client |
| *public synchronized void close()* | Closes the server socket |

**Example of Java Socket Programming**

Let's see an example of java socket programming in which client sends a text and server receives it.

*File: MySocketServer.java*

import java.io.\*;

import java.net.\*;

public class MyServer {

public static void main(String[] args){

try{

ServerSocket ss=new ServerSocket(6666);

Socket s=ss.accept();//establishes connection

DataInputStream dis=new DataInputStream(s.getInputStream());

String  str=(String)dis.readUTF();

System.out.println("message= "+str);

ss.close();

}catch(Exception e){System.out.println(e);}

}

}

*File: MySocketClient.java*

import java.io.\*;

import java.net.\*;

public class MyClient {

public static void main(String[] args) {

try{

Socket s=new Socket("localhost",6666);

DataOutputStream dout=new DataOutputStream(s.getOutputStream());

dout.writeUTF("Hello Server");

dout.flush();

dout.close();

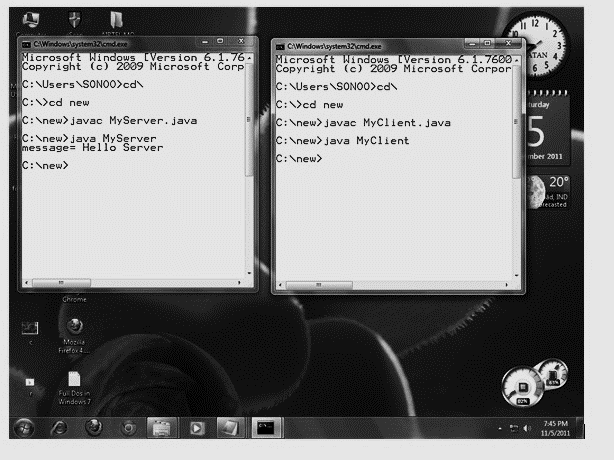
s.close();

}catch(Exception e){System.out.println(e);}

}

}

To execute this program, open two command prompts and execute each program at each command prompt separately as displayed in the below figure.



After running the client application, a message will be displayed on the server console.

Java InetAddress class

Java *InetAddress* class represents an internet protocol address (IP address). The *java.net.InetAddress* class provides methods to get the IP of any host name, for example www.ducat.com, www.google.com, www.facebook.com etc.

**Class Methods**

Below are the methods used in *InetAddress* class:

|  |  |
| --- | --- |
| Method | Description |
| *public static InetAddress getByName(String host) throws UnknownHostException* | Returns the instance of *inetaddress* containing localhost IP and name |
| *public static InetAddress getLocalHost() throws UnknownHostException* | Returns the instance of *inetadddress* containing local host name and address |
| *public String getHostName()* | Returns the host name of the IP address |
| *public String getHostAddress()* | Returns the IP address in string format |

**Example of Java InetAddress class**

Let's see a simple example of InetAddress class to get IP address of www.ducatindia.com website.

import java.io.\*;

import java.net.\*;

public class InetDemo{

public static void main(String[] args){

try{

InetAddress ip=InetAddress.getByName("www.ducatindia.com");

System.out.println("Host Name: "+ip.getHostName());

System.out.println("IP Address: "+ip.getHostAddress());

}catch(Exception e){System.out.println(e);}

}

}

Output

Host Name: www.ducatindia.com

IP Address: 162.251.85.191

Java DatagramSocket and DatagramPacket Classes

Java *DatagramSocket* and *DatagramPacket* classes are used for connection-less socket programming.

**Java DatagramSocket Class**

Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets.

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

**Class Constructors**

The commonly used constructors of *DatagramSocket* class are:

* **DatagramSocket() throws SocketEeption:**It creates a datagram socket and binds it with the available port number on the *localhost* machine.
* **DatagramSocket(int port) throws SocketEeption:**It creates a datagram socket and binds it with the given port number.
* **DatagramSocket(int port, InetAddress address) throws SocketEeption:**It creates a datagram socket and binds it with the specified port number and host address.

**Java DatagramPacket Class**

Java DatagramPacket is a message that can be sent or received. If you send multiple packets, they can arrive in any order. Moreover, the packet delivery is not guaranteed.

**Class Constructors**

The commonly used constructors of*DatagramPacket* class are:

* **DatagramPacket(byte[] barr, int length):**It creates a datagram packet. This constructor is used to receive the packets.
* **DatagramPacket(byte[] barr, int length, InetAddress address, int port):**It creates a datagram packet. This constructor is used to send the packets.

**Example**

Now take a look at code that is used to send DatagramPacket by DatagramSocket.

//DSender.java

import java.net.\*;

public class DSender{

  public static void main(String[] args) throws Exception {

    DatagramSocket ds = new DatagramSocket();

    String str = "Hello Java";

    InetAddress ip = InetAddress.getByName("127.0.0.1");

    DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);

    ds.send(dp);

    ds.close();

  }

}

**Example of Receiving DatagramPacket by DatagramSocket**

//DReceiver.java

import java.net.\*;

public class DReceiver{

  public static void main(String[] args) throws Exception {

    DatagramSocket ds = new DatagramSocket(3000);

    byte[] buf = new byte[1024];

    DatagramPacket dp = new DatagramPacket(buf, 1024);

    ds.receive(dp);

    String str = new String(dp.getData(), 0, dp.getLength());

    System.out.println(str);

    ds.close();

  }

}

Java URL Class

The Java URL class represents a URL. It points to a resource on the World Wide Web.

For example: http://www.ducat.com:80/index.jsp

A URL contains the following types of information:

* **Protocol:** In the example (stated above), *http* is the protocol.
* **Server name or IP address:** In the above example, *www.ducat.com* is the server name.
* **Port number:** It is an optional attribute. If we write *http://www.ducat.com:80/index.jsp*, 80 is the port number. If port number is not mentioned in the URL, it returns -1.
* **File name or directory name:** In the example*, index.jsp* is the file name.

**Class Methods**

The java.net.URL class provides many methods. The important methods of URL class are given below.

|  |  |
| --- | --- |
| Method | Description |
| *public String getProtocol()* | Returns the protocol of the URL |
| *public String getHost()* | Returns the host name of the URL |
| *public String getPort()* | Returns the Port Number of the URL |
| *public String getFile()* | Returns the file name of the URL |
| *public URLConnection openConnection()* | Returns the instance of *URLConnection* that is associated with this URL |

**Example of Java URL Class**

//URLDemo.java

import java.io.\*;

import java.net.\*;

public class URLDemo{

public static void main(String[] args){

try{

URL url=new URL("http://www.ducatindia.com/javaanduitraining/ ");

System.out.println("Protocol: "+url.getProtocol());

System.out.println("Host Name: "+url.getHost());

System.out.println("Port Number: "+url.getPort());

System.out.println("File Name: "+url.getFile());

}catch(Exception e){System.out.println(e);}

}

}

Output

Protocol: http

Host Name: www.ducatindia.com

Port Number: -1

File Name: /javaanduitraining

Java URLConnection class

The Java URLConnection class represents a communication link between the URL and the application. This class can be used to read and write the data to the specified resource referred by the URL.

**Getting an Object of URLConnection Class**

To get the object of the URL class, we use the *openConnection()* method. It returns the object of URLConnection class.

**Syntax:**

public URLConnection openConnection()throws IOException{}

**Displaying Source Code of a Webpage by URLConnection Class**

The URLConnection class provides various methods, such as getInputStream() that can be used to display the data of a webpage. This method returns all the data of the specified URL in the stream, which can be read and displayed.

**Example of Java URLConnection Class:**

import java.io.\*;

import java.net.\*;

public class URLConnectionExample {

public static void main(String[] args){

try{

URL url=new URL("http://www.ducatindia.com/javaanduitraining/");

URLConnection urlcon=url.openConnection();

InputStream stream=urlcon.getInputStream();

int i;

while((i=stream.read())!=-1){

System.out.print((char)i);

}

}catch(Exception e){System.out.println(e);}

}

}

Java HttpURLConnection Class

The Java *HttpURLConnection* class is an ‘*HTTPspecificURLConnection’*. It works for HTTP protocol only.

By the help of HttpURLConnection class, we can get information of any HTTP URL such as header information, status code, response code etc.

The java.net.HttpURLConnection is a subclass of the URLConnection class.

**Getting the object of HttpURLConnection class**

The openConnection() method of URL class returns the object of URLConnection class.

**Syntax:**

public URLConnection openConnection()throws IOException{}

**Example**

You can typecast it to HttpURLConnection type as given below:

URL url=new URL("http://www.ducatindia.com/javaanduitraining/");

HttpURLConnection huc=(HttpURLConnection)url.openConnection();

Java HttpURLConnecton Example

import java.io.\*;

import java.net.\*;

public class HttpURLConnectionDemo{

public static void main(String[] args){

try{

URL url=new URL("http://www.ducatindia.com/javaanduitraining/");

HttpURLConnection huc=(HttpURLConnection)url.openConnection();

for(int i=1;i<=8;i++){

System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));

}

huc.disconnect();

}catch(Exception e){System.out.println(e);}

}

}

Summary

* Networks share resources, exchange files and support electronic communications
* The *java.net.HttpURLConnection* is a subclass of the *URLConnection* class
* *Socket* and *ServerSocket* classes are used for connection-oriented socket programming
* Java *DatagramSocket*class represents a connection-less socket for sending and receiving datagram packets
* The *java.net.InetAddress* class provides methods to get the IP of any host name
* Java Socket programming is used for communication between the applications running on different JRE
* The Java *URLConnection* class represents a communication link between the URL and the application

# Chapter 13 – Reflection

Understanding the Need of Reflection

Reflection is an API which is used to examine or modify the behaviour of *methods, classes, interfaces* at runtime. Reflection is a key mechanism to allow an application or framework to work with code that might not have even been written yet.

Reflection has the ability to "reflect" on the structure of your program. Reflection allows instantiation of new objects, invocation of methods, and get/set operations on class variables dynamically at run time without having prior knowledge of its implementation.

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

An example to show use of reflection in Java:

Method[] methods = MyObject.class.getMethods();

for(Method method : methods){

System.out.println("method = " + method.getName());

}

This example obtains the Class object from the class called MyObject. Using the class object the example gets a list of the methods in that class, iterates the methods and print out their names.

Reflection can be used to get information about:

* **Classes:** The getClass() method is used to get the name of the class to which an object belongs.
* **Constructors:** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
* **Methods:** The getMethods() method is used to get the public methods of the class to which an objects belongs.

The Reflection API is mainly used in IDEs (e.g. Eclipse, MyEclipse, NetBeans etc.), Debugger and Test Tools etc. Debuggers use the property of reflection to examine private members on classes.

Class loading

The *Java Classloader* is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine. Usually classes are only loaded on demand. There are two types of classloading in Java.

* Static Class Loading
* Dynamic Class Loading

**Static Class Loading**: Creating objects and instance using new keyword is known as static class loading. The retrieval of class definition and instantiation of the object is done at compile time.

**Example**

class Student {

int id;

String name;

}

class TestStudent2 {

public static void main(String args[]) {

Student s1 = new Student();

s1.id = 101;

s1.name = "Ducat";

System.out.println(s1.id + " " + s1.name); //printing members with a white space

}

}

Output

101 Ducat

**Dynamic Class Loading**: It allows the loading of java code that is not known about before a program starts. Many classes rely on other classes and resources such as icons which make loading a single class unfeasible. For this reason the ClassLoader (*java.lang.ClassLoader*) is used to manage all the inner dependencies of a collection of classes.

**Example**

class MyClass {

}

public class ClassLoading {

public static void main(String[] args) {

ClassLoader cl = ClassLoading.class.getClassLoader();

try {

Class c = cl.loadClass("MyClass");

System.out.println("c.getName() = " + c.getName());

} catch (Exception e) {

System.out.println("Exception: " + e.toString());

}

}

}

Output

$ java ClassLoading

c.getName() = MyClass

Getting Information about Class’s Modifiers, Fields, Methods, Constructors and Super Classes

**Class Modifiers**

You can access the modifiers of a class via the Class object. The class modifiers are the keywords "public", "private", "static" etc. You obtain the class modifiers like this:

Class aClass = ... //obtain Class object. See prev. section

int modifiers = aClass.getModifiers();

The modifiers are packed into an int where each modifier is a flag bit that is either set or cleared. You can check the modifiers using these methods in the class java.lang.reflect.Modifier:

Modifier.isAbstract(int modifiers)

Modifier.isFinal(int modifiers)

Modifier.isInterface(int modifiers)

Modifier.isNative(int modifiers)

Modifier.isPrivate(int modifiers)

Modifier.isProtected(int modifiers)

Modifier.isPublic(int modifiers)

Modifier.isStatic(int modifiers)

Modifier.isStrict(int modifiers)

Modifier.isSynchronized(int modifiers)

Modifier.isTransient(int modifiers)

Modifier.isVolatile(int modifiers)

**Fields**

You can access the fields (member variables) of a class like this:

Field[] method = aClass.getFields();

Fields are covered in more detail in the text on Fields.

**Methods**

You can access the methods of a class like this:

Method[] method = aClass.getMethods();

Methods are covered in more detail in the text on Methods.

**Constructors**

You can access the constructors of a class like this:

Constructor[] constructors = aClass.getConstructors();

Constructors are covered in more detail in the text on Constructors.

**Superclass**

From the Class object you can access the superclass of the class. Here is how:

Class superclass = aClass.getSuperclass();

The superclass class object is a Class object like any other, so you can continue doing class reflection on that too.

Finding out Constant and Method Declaration Belong to an Interface

**Finding out constant**

Some java classes have some predefined constants. For example java.lang.Math has the values of *E* and *PI*stored in fields.   
  
This java example shows how to get the value of these fields from a class using reflection.

Source: (ConstantValues.java)

import java.lang.reflect.Field;

public class ConstantValues {

public static void main(String... args) {

try {

Class c = Class.forName(args[0]);

Field[] fields = c.getFields();

System.out.println("Constants for: " + c.getName());

for (Field f: fields) {

f.setAccessible(true);

System.out.print(" " + f.getName());

System.out.println(" : " + f.get(null));

}

} catch (Exception e) {

System.out.println("Exception: " + e.toString());

}

}

}

Output

$ java ConstantValues java.lang.Math

E : 2.718281828459045

PI : 3.141592653589793

$ java ConstantValues java.lang.Byte

MIN\_VALUE : -128

MAX\_VALUE : 127

TYPE : byte

SIZE : 8

$ java ConstantValues java.lang.Float

POSITIVE\_INFINITY : Infinity

NEGATIVE\_INFINITY : -Infinity

NaN : NaN

MAX\_VALUE : 3.4028235E38

MIN\_NORMAL : 1.17549435E-38

MIN\_VALUE : 1.4E-45

MAX\_EXPONENT : 127

MIN\_EXPONENT : -126

SIZE : 32

TYPE : float

**Finding out method**

A method declaration may override a super class method and implement multiple unrelated interface methods at the same time, e.g.

class A {

public void foo() {}

}

interface I1 {

void foo();

}

interface I2 {

void foo();

}

class B extends A implements I1, I2 {

@Override

public void foo() {}

}

Then, when you use this naive approach:

public static void printCandidates(Method m) {

Class<?> cl=m.getDeclaringClass();

String name=m.getName(); Class<?>[] param=m.getParameterTypes();

for(Class<?> i: cl.getInterfaces()) try {

Method ifM=i.getMethod(name, param);

System.out.println("candidate "+ifM.getDeclaringClass().getName()+'.'+name);

} catch(NoSuchMethodException ex) { ex.printStackTrace();}

for(;;) {

System.out.println("candidate "+cl.getName()+'.'+name);

cl=cl.getSuperclass();

if(cl==null) break;

try { m=cl.getMethod(name, param); }

catch(NoSuchMethodException ex) { break; }

}

}

and invoke printCandidates(new B().getClass().getMethod("foo"));, it will print

candidate I1.foo

candidate I2.foo

candidate B.foo

candidate A.foo

Creating an instance of the class whose name is not known until runtime

There are two reflective methods for creating instances of classes: java.lang.reflect.Constructor.newInstance() and Class.newInstance(). The former is preferred and is thus used in these examples because:

* Class.newInstance() can only invoke the zero-argument constructor, while Constructor.newInstance() may invoke any constructor, regardless of the number of parameters.
* Class.newInstance() throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. Constructor.newInstance() always wraps the thrown exception with an InvocationTargetException.
* Class.newInstance() requires that the constructor be visible; Constructor.newInstance() may invoke private constructors under certain circumstances.

Sometimes it may be desirable to retrieve internal state from an object which is only set after construction. Consider a scenario where it is necessary to obtain the internal character set used by java.io.Console.(The Console character set is stored in an private field and is not necessarily the same as the Java virtual machine default character set returned byjava.nio.charset.Charset.defaultCharset()). The ConsoleCharset example shows how this might be achieved:

import java.io.Console;

import java.nio.charset.Charset;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.InvocationTargetException;

import static java.lang.System.out;

public class ConsoleCharset {

public static void main(String... args) {

Constructor[] ctors = Console.class.getDeclaredConstructors();

Constructor ctor = null;

for (int i = 0; i < ctors.length; i++) {

ctor = ctors[i];

if (ctor.getGenericParameterTypes().length == 0)

break;

}

try {

ctor.setAccessible(true);

Console c = (Console)ctor.newInstance();

Field f = c.getClass().getDeclaredField("cs");

f.setAccessible(true);

out.format("Console charset : %s%n", f.get(c));

out.format("Charset.defaultCharset(): %s%n",

Charset.defaultCharset());

// production code should handle these exceptions more gracefully

} catch (InstantiationException x) {

x.printStackTrace();

} catch (InvocationTargetException x) {

x.printStackTrace();

} catch (IllegalAccessException x) {

x.printStackTrace();

} catch (NoSuchFieldException x) {

x.printStackTrace();

}

}

}

Getting and setting values of an object’s field if field name is unknown until runtime

Using Java Reflection you can inspect the methods of classes and invoke them at runtime. This can be used to detect what getters and setters a given class has. You cannot ask for getters and setters explicitly, so you will have to scan through all the methods of a class and check if each method is a getter or setter.

First let's establish the rules that characterizes getters and setters:

* **Getter**: A getter method have its name start with "get", take 0 parameters, and returns a value.
* **Setter:** A setter method have its name start with "set", and takes 1 parameter.

Setters may or may not return a value. Some setters return void, some the value set, others the object the setter were called on for use in method chaining. Therefore you should make no assumptions about the return type of a setter.

Here is a code example that finds getter and setters of a class:

public static void printGettersSetters(Class aClass){

Method[] methods = aClass.getMethods();

for(Method method : methods){

if(isGetter(method)) System.out.println("getter: " + method);

if(isSetter(method)) System.out.println("setter: " + method);

}

}

public static boolean isGetter(Method method){

if(!method.getName().startsWith("get")) return false;

if(method.getParameterTypes().length != 0) return false;

if(void.class.equals(method.getReturnType()) return false;

return true;

}

public static boolean isSetter(Method method){

if(!method.getName().startsWith("set")) return false;

if(method.getParameterTypes().length != 1) return false;

return true;

}

Invoking a Method on an Object If the Method is Unknown until Runtime

You can invoke a method like this:

//get method that takes a String as argument

Method method = MyObject.class.getMethod("doSomething", String.class);

Object returnValue = method.invoke(null, "parameter-value1");

The null parameter is the object you want to invoke the method on. If the method is static you supply nullinstead of an object instance. In this example, if doSomething(String.class) is not static, you need to supply a valid MyObject instance instead of null;

The Method.invoke(Object target, Object ... parameters) method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the method you are invoking. In this case it was a method taking a String,so one String must be supplied.

Invoking Private Methods

You can call the private method from outside the class by changing the runtime behaviour of the class.

By the help of *java.lang.Class* class and *java.lang.reflect.Method*class, we can call private method from any other class.

**Required methods of Method class**

1)*public void setAccessible(boolean status) throws SecurityException:* sets the accessibility of the method.

2) *public Object invoke(Object method, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException***:** is used to invoke the method.

**Required method of Class class**

*public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException***:** returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.

**Example of calling private method from another class**

Let's see the simple example to call private method from another class.

*File: A.java*

public class A {

  private void message(){System.out.println("hello java"); }

}

*File: MethodCall.java*

import java.lang.reflect.Method;

public class MethodCall{

public static void main(String[] args)throws Exception{

    Class c = Class.forName("A");

    Object o= c.newInstance();

    Method m =c.getDeclaredMethod("message", null);

    m.setAccessible(true);

    m.invoke(o, null);

}

}

Output:

hello java

Summary

* Reflection is an API which is used to examine or modify the behaviour of *methods, classes, interfaces* at runtime
* The class modifiers are the keywords "public", "private", "static" etc.
* A method declaration may override a super class method and implement multiple unrelated interface methods at the same time
* Using Java Reflection you can inspect the methods of classes and invoke them at runtime
* A getter method have its name start with "get", take 0 parameters, and returns a value.
* By the help of *java.lang.Class* class and *java.lang.reflect.Method*class, we can call private method from any other class

Chapter 14 – Additional Concepts

var-args (Variable Arguments)

The *var-args* allows the method to accept zero or multiple arguments. Before the addition of *var-args* to the package, the developers had to use the overloaded method or take an array as the method parameter, which led to the maintenance problem. Hence, they were not considered as good options.

Therefore, in a situation when we don't know how many arguments we will have to pass in the method, *var-args* proves to be a better approach.

**Advantage of var-args**

* It offers a much simpler way to have a method with variable number of arguments.
* It reduces code length as there is no need to write overloaded methods.

**Syntax of var args**

Note the syntax to use var-args. It uses ellipsis i.e. three dots after the data type.

return\_type method\_name(data\_type... variableName){}

**Example of var args in Java**

class VarargsExample1{

  static void display(String... values){

  System.out.println("display method invoked ");

 }

 public static void main(String args[]){

 display();//zero argument

 display("my","name","is","varargs");//four arguments

 }

}

Output

display method invoked

display method invoked

**Note:** Variable and argument must be the last argument of method.

Static Import

Static import, as a concept was introduced in Java 1.5. With the help of this concept, we can access the static members of a class directly without referring to their class name or any object. For example, while we always have to use the *Math* class (*Math.sqrt())* to call its *sqrt()* method***,*** now with the static import feature we can directly access the *sqrt()* method.

According to SUN Microsystem, static import improves the code readability and enhance coding. On the contrary, the programming experts say that it might lead to confusion in code readability. Therefore, unless it is really required, we should notgo for the static import.

**Advantage of static import**

The amount of the code, we are required to write, reduces, when we access any static member of a class.

**Disadvantage of static import**

In situations wherein the static import feature is overused, the program becomes unreadable and unmaintainable.

**Example of static import**

import static java.lang.System.\*;

class StaticImportExample{

  public static void main(String args[]){

   out.println("Hello");//Now no need of System.out

   out.println("Java");

   }

}

For-each loop

The *for-each loop* was introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

**Advantage of for-each loop:**

* It makes the code more readable.
* It eliminates the possibility of programming errors.

**Syntax of for-each loop:**

for(data\_type variable : array | collection){}

**Example of for-each loop for traversing the array elements:**

class ForEachExample1{

  public static void main(String args[]){

   int arr[]={12,13,14,44};

   for(int i:arr){

     System.out.println(i);

   }

 }

}

Output

12

13

14

44

Enum (Enumerate)

A *Java Enum* is a special Java type used to define collections of constants. More precisely, a Java *enum* type is a special kind of Java class. An *enum* can contain constants, methods etc. Java *enums* were added in Java 5.

In Java, enums are represented using enum data type. Java enums are more powerful than C/ enums . In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types (i.e. enumerated data types).

**Declaration of enum in java:**

Enum declaration can be done outside a Class or inside a Class but not inside a Method.

**Example of enum**

// A simple enum example where enum is declared

// outside any class

enumColor

{

    RED, GREEN, BLUE;

}

publicclassTest

{

    // Driver method

    publicstaticvoidmain(String[] args)

    {

        Color c1 = Color.RED;

        System.out.println(c1);

    }

}

Output

RED

**Example of enum declaration inside a class**

// enum declaration inside a class.

public class Test

{

enum Color

{

RED, GREEN, BLUE;

}

// Driver method

public static void main(String[] args)

{

Color c1 = Color.RED;

System.out.println(c1);

}

}

Output

RED

Notice the enum keyword which is used in place of class or interface. The Java enum keyword signals to the Java compiler that this type definition is an enum.

printf( ) Method

Earlier the print and println methods were used for printing strings to standard output(System.out).Nowadays, in addition to the print()andprintln()methods, theprintf()method is also quite frequently used for displaying text that needs to be formatted.

The printf () method requires a format String and a set of other arguments whose number depends on the format String. The format String is similar to a normal String with the exception that it contains placeholders where appropriate data items may be placed. These place holders state the type of data that is going to be placed and also include any other necessary formatting details such as the number of decimal digits to be displayed, the alignment of the text and so on. For example, consider the following two equivalent statements assuming that the variable str holds the String "Java":

System.out.println ("Welcome to "+str);

System.out.printf ("Welcome to %s", str);

Both of the above statements print the String "Welcome to Java". However, the approach taken varies. In the first statement, we have used concatenation while in the second case, we have achieved the same result by inserting a placeholder for a String (%s) and then stating the String that needs to be placed in that defined area.

We may specify as many placeholders as we wish and then state their values as arguments in the same order in which they need to be placed in the empty placeholders. The first parameter in the printf() method call is known as the format string. And the '%s' and any other similar entities are known as format specifiers.

The digit that follows the % sign ( in this case, an s) is known as the conversion character. Format specifiers are provided with additional information depending on the formatting that we wish to apply. Different format specifiers are used to print different types of text. The table below summarises the most commonly used.

|  |  |
| --- | --- |
| Format specifier | Description |
| *%d* | Displays a decimal (base 10 ) integer |
| *%f* | Display a floating point value in decimal format |
| *%e or %E* | Display a floating point number in exponential notation |
| *%c or %C* | Display characters |
| *%s or %S* | Display Strings |
| *%b or %B* | Display boolean values |
| *%%* | Display a % sign |

Therefore, to print a short, byte, int or long data item, we use the %d format specifier. To print float and double values, we use either %f or %e depending on the requirement of output. To print boolean, char and String data items, we use %b, %c and %s specifiers respectively.

Static, Default, and Private Methods

In earlier versions of Java, all methods of an interface had to be abstract, i.e. without a body. Nowadays you can add three kinds of methods with a concrete implementation: static, default, and private methods. The following sections describe these methods.

**Static Methods**

There was never a technical reason why an interface could not have static methods, but they did not fit into the view of interfaces as abstract specifications. That thinking has now evolved. In particular, factory methods make a lot of sense in interfaces. For example, the IntSequence interface can have a static method digitsOf that generates a sequence of digits of a given integer:

IntSequence digits = IntSequence.digitsOf(1729);

The method yields an instance of some class implementing the IntSequence interface, but the caller need not care which one it is.

public interface IntSequence {  
    ...  
    static IntSequence digitsOf(int n) {  
        return new DigitSequence(n);  
    }  
}

**Default Methods**

You can supply a *default* implementation for any interface method. You must tag such a method with the default modifier.

public interface IntSequence {  
    default boolean hasNext() { return true; }  
        // By default, sequences are infinite  
    int next();  
}

A class implementing this interface can choose to override the hasNext method or to inherit the default implementation.

Default methods put an end to the classic pattern of providing an interface and a companion class that implements most or all of its methods, suchas Collection/AbstractCollection or WindowListener/WindowAdapter in the Java API. Nowadays you should just implement the methods in the interface.

**Private Methods**

After Java 9, methods in an interface can be private. A private method can be static or an instance method, but it cannot be a default method since that can be overridden. As private methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.

**Example of static, default and private method inside interface is given below:**

public interface CustomInterface {

public abstract void method1();

public default void method2() {

method4(); //private method inside default method

method5(); //static method inside other non-static method

System.out.println("default method");

}

public static void method3() {

method5(); //static method inside other static method

System.out.println("static method");

}

private void method4(){

System.out.println("private method");

}

private static void method5(){

System.out.println("private static method");

}

}

public class CustomClass implements CustomInterface {

@Override

public void method1() {

System.out.println("abstract method");

}

public static void main(String[] args){

CustomInterface instance = new CustomClass();

instance.method1();

instance.method2();

CustomInterface.method3();

}

}

Output

abstract method

private method

private static method

default method

private static method

static method

Date-Time API - Time Package to Work with Local Date & Time

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API.

* **Not thread safe** − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
* **Poor design** – In the API, default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
* **Difficult time zone handling** − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.

* **Local** − Simplified date-time API with no complexity of timezone handling.
* **Zoned** − Specialized date-time API to deal with various timezones.

Let’s discuss local classes of date-time API:

**Local Date-Time API**

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required. Let's see them in action.

**LocalDate**

LocalDate is an immutable class that represents Date with default format of yyyy-MM-dd. We can use now() method to get the current date. We can also provide input arguments for year, month and date to create LocalDate instance. This class provides overloaded method for now() where we can pass ZoneId for getting date in specific time zone. This class provides the same functionality as java.sql.Date.

**Example**

import java.time.LocalDate;

import java.time.Month;

import java.time.ZoneId;

public class LocalDateExample {

public static void main(String[] args) {

//Current Date

LocalDate today = LocalDate.now();

System.out.println("Current Date="+today);

//Creating LocalDate by providing input arguments

LocalDate firstDay\_2014 = LocalDate.of(2014, Month.JANUARY, 1);

System.out.println("Specific Date="+firstDay\_2014);

//Try creating date by providing invalid inputs

//LocalDate feb29\_2014 = LocalDate.of(2014, Month.FEBRUARY, 29);

//Exception in thread "main" java.time.DateTimeException:

//Invalid date 'February 29' as '2014' is not a leap year

//Current date in "Asia/Kolkata", you can get it from ZoneId javadoc

LocalDate todayKolkata = LocalDate.now(ZoneId.of("Asia/Kolkata"));

System.out.println("Current Date in IST="+todayKolkata);

//java.time.zone.ZoneRulesException: Unknown time-zone ID: IST

//LocalDate todayIST = LocalDate.now(ZoneId.of("IST"));

//Getting date from the base date i.e 01/01/1970

LocalDate dateFromBase = LocalDate.ofEpochDay(365);

System.out.println("365th day from base date= "+dateFromBase);

LocalDate hundredDay2014 = LocalDate.ofYearDay(2014, 100);

System.out.println("100th day of 2014="+hundredDay2014);

}

}

*LocalDate* methods explanation is provided in comments, when we run this program, we get following output.

Current Date=2014-04-28

Specific Date=2014-01-01

Current Date in IST=2014-04-29

365th day from base date= 1971-01-01

100th day of 2014=2014-04-10

**LocalTime**

LocalTime is an immutable class whose instance represents a time in the human readable format. It’s default format is hh:mm:ss.zzz. Just like LocalDate, this class provides time zone support and creating instance by passing hour, minute and second as input arguments. Let’s look at its usage with a simple program.

**Example**

import java.time.LocalDate;

import java.time.LocalTime;

import java.time.ZoneId;

public class LocalTimeExample {

public static void main(String[] args) {

//Current Time

LocalTime time = LocalTime.now();

System.out.println("Current Time="+time);

//Creating LocalTime by providing input arguments

LocalTime specificTime = LocalTime.of(12,20,25,40);

System.out.println("Specific Time of Day="+specificTime);

//Try creating time by providing invalid inputs

//LocalTime invalidTime = LocalTime.of(25,20);

//Exception in thread "main" java.time.DateTimeException:

//Invalid value for HourOfDay (valid values 0 - 23): 25

//Current date in "Asia/Kolkata", you can get it from ZoneId javadoc

LocalTime timeKolkata = LocalTime.now(ZoneId.of("Asia/Kolkata"));

System.out.println("Current Time in IST="+timeKolkata);

//java.time.zone.ZoneRulesException: Unknown time-zone ID: IST

//LocalTime todayIST = LocalTime.now(ZoneId.of("IST"));

//Getting date from the base date i.e 01/01/1970

LocalTime specificSecondTime = LocalTime.ofSecondOfDay(10000);

System.out.println("10000th second time= "+specificSecondTime);

}

}

When we run above program for LocalTime examples, we get following output.

Current Time=15:51:45.240

Specific Time of Day=12:20:25.000000040

Current Time in IST=04:21:45.276

10000th second time= 02:46:40

Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

**Functional Interface**

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation @*FunctionalInterface*, which is used to declare an interface as functional interface.

**Why use Lambda Expression**

1. To provide the implementation of Functional interface.
2. Reduce length of codes.

**Syntax of Lambda Expression**

(argument-list) -> {body}

Java lambda expression consists of three components:

* **Argument-list:** It can be empty or non-empty as well.
* **Arrow-token:** It is used to link arguments-list and body of expression.
* **Body:** It contains expressions and statements for lambda expression.

**Example of lambda expression is given below:**

  public class Java8Tester {

public static void main(String args[]) {

Java8Tester tester = new Java8Tester();

//with type declaration

MathOperation addition = (int a, int b) -> a + b;

//with out type declaration

MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces

MathOperation multiplication = (int a, int b) -> { return a \* b; };

//without return statement and without curly braces

MathOperation division = (int a, int b) -> a / b;

System.out.println("10 + 5 = " + tester.operate(10, 5, addition));

System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));

System.out.println("10 / 5 = " + tester.operate(10, 5, division));

//without parenthesis

GreetingService greetService1 = message ->

System.out.println("Hello " + message);

//with parenthesis

GreetingService greetService2 = (message) ->

System.out.println("Hello " + message);

greetService1.sayMessage("Mahesh");

greetService2.sayMessage("Suresh");

}

interface MathOperation {

int operation(int a, int b);

}

interface GreetingService {

void sayMessage(String message);

}

private int operate(int a, int b, MathOperation mathOperation) {

return mathOperation.operation(a, b);

}

}

**Output**

10 + 5 = 15

10 - 5 = 5

10 x 5 = 50

10 / 5 = 2

Hello Mahesh

Hello Suresh

HTTPClient API

HTTP Client provides synchronous and asynchronous request mechanisms.

The API consists of 3 core classes:

* *HttpRequest***–**Itrepresents the request to be sent via the *HttpClient*
* *HttpClient***–**Itbehaves as a container for configuration information common to multiple requests
* *HttpResponse***–**Itrepresents the result of an *HttpRequest* call

**HttpRequest**

*HttpRequest,*as the name*suggests,* is an object which represents request we want to send. New instances can be created using *HttpRequest.Builder.*

We can get it by calling *HttpRequest.newBuilder()*. *Builder* class provides a bunch of methods which we can use to configure our request.

**HttpClient**

All requests are sent using *HttpClient* which can be instantiated using the *HttpClient.newBuilder()*method or by calling *HttpClient.newHttpClient()*.

It provides a lot of useful and self-describing methods we can use to handle our request/response.

**HttpResponse Object**

The *HttpResponse* class represents the response from the server. It provides a number of useful methods – but two the most important are:

* *statusCode()*– It returns status code (type *int*) for a response (*HttpURLConnection* class contains possible values)
* *body()* – It returns a body for a response (return type depends on the response *BodyHandler*parameter passed to the *send()* method)

The response object has other useful method which we’ll cover like *uri()*, *headers()*, *trailers()* and *version()*.

**Example**

import java.io.\*;

import org.apache.http.HttpResponse;

import org.apache.http.client.HttpClient;

import org.apache.http.client.methods.CloseableHttpResponse;

import org.apache.http.client.methods.HttpGet;

import org.apache.http.impl.client.CloseableHttpClient;

import org.apache.http.impl.client.DefaultHttpClient;

import org.apache.http.impl.client.HttpClients;

public class CloseableHttpClientExmpl {

public static void main(String[] args) {

CloseableHttpClient client = HttpClients.createDefault();

HttpGet request = new HttpGet("http://localhost/httpclient/letsstart.php");

CloseableHttpResponse response = null;

try {

response = client.execute(request);

int status = response.getStatusLine().getStatusCode();

if (status >= 200 && status < 300) {

BufferedReader br;

br = new BufferedReader(new InputStreamReader(response.getEntity().getContent()));

String line = "";

while ((line = br.readLine()) != null) {

System.out.println(line);

}

} else {

System.out.println("Unexpected response status: " + status);

}

} catch (IOException | UnsupportedOperationException e) {

e.printStackTrace();

} finally {

if(null != response){

try {

response.close();

client.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

## Java Generics

The The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

**Advantage of Java Generics**

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** Before Generics, we always needed to type cast. With Generics, there is no need to typecast the object.

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

**Example of java generics:**

import java.util.\*;

class TestGenerics1{

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>();

list.add("rahul");

list.add("jai");

//list.add(32);//compile time error

String s=list.get(1);//type casting is not required

System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

**Output**

element is: jai

rahul

jai

Annotations in Java

Annotations are used to provide supplement information about a program.

* Annotations start with ‘**@**’.
* Annotations do not change action of a compiled program.
* Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
* Annotations are not pure comments as they can change the way a program is treated by compiler.

**Categories of Annotations**

There are 3 categories of Annotations:

**1. Marker Annotations:**  
The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. *@Override* is an example of Marker Annotation.

**Example:**

@TestAnnotation()

**2. Single value Annotations:**  
These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don’t need to specify the name of the member. However in order to use this shorthand, the name of the member must be value.

**Example**

@TestAnnotation(“testing”);

**3. Full Annotations:**  
These annotations consist of multiple data members/ name, value, pairs.

**Example**

@TestAnnotation(owner=”Rahul”, value=”Class Geeks”)

Let’s see a example showing use of built-in annotations in java:

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.List;

public class AnnotationExample {

public static void main(String[] args) {

}

@Override

@MethodInfo(author = "Pankaj", comments = "Main method", date = "Nov 17 2012", revision = 1)

public String toString() {

return "Overriden toString method";

}

@Deprecated

@MethodInfo(comments = "deprecated method", date = "Nov 17 2012")

public static void oldMethod() {

System.out.println("old method, don't use it.");

}

@SuppressWarnings({ "unchecked", "deprecation" })

@MethodInfo(author = "Pankaj", comments = "Main method", date = "Nov 17 2012", revision = 10)

public static void genericsTest() throws FileNotFoundException {

List l = new ArrayList();

l.add("abc");

oldMethod();

}

}

VAR

The VAR type allows you to declare a variable without specifying its type. For example, instead of using *string str = "Java"*, you can now just type *var str = "Java".*

This may not sound like much when declaring *strings* or an *int* variable. However, it helps reduce the size of code, enhancing its readability when dealing with the complex data types. This is especially true in case of writing boilerplate code and ceremonies. Tasks that can be executed in 5 minutes when performed in other languages (such as, Python, Groovy, or JavaScript) take more than 30 minutes in Java (mainly due to verbosity of the code).

Though the *type* inference was improved significantly in Java 8 with the introduction of lambda expressions, method references and streams, the local variables still needed to be declared with proper types — which has been improved now.

**Java 10 VAR Example:**

var i = 10;

var j = new student();

At this point, you may not fully appreciate what var is doing for you, but look at the next example:

var list = List.of(1, 2.0, "3")

Here, list will be inferred into List<? extends serializable & comparable <..>>, which is an intersection type.

The use of var also makes your code concise by reducing duplication, for e.g. the name of the class that comes in both right and left-hand side of assignments, as shown in the following example:

ByteArrayOutputStream bos = new ByteArrayOutputStream();

Here, ByteArrayOutputStream repeats twice. You can eliminate that by using the var feature of Java 10 as shown below:

var bos = new ByteArrayOutputStream();

Java Module

A Java Module is a mechanism to package up your Java application and Java packages into Java modules. A Java module can specify which of the Java packages it contains that should be visible to other Java modules using this module. A Java module must also specify which other Java modules is requires to do its job. This will be explained in more detail later in this Java modules tutorial.

Java modules is a new feature in Java 9 via the Java Platform Module System (JPMS). The Java Platform Module System is also sometimes referred to as Java Jigsaw or Project Jigsaw depending on where you read. Jigsaw was the internally used project name during development. Later Jigsaw changed name to Java Platform Module System.

**Why modules?**

In a few words, a module is generally a JAR file that contains a set of packages. Just like fields and methods are grouped into classes, and they in turn into packages, packages are grouped into modules.

Modularizing an application means decomposing it into multiple modules that work together.

The important thing is that with modules, you have to explicitly require other modules your module depends on, and you have to export packages from your module to be used by other modules. This brings some benefits and solves three problems.

First of all, the direct consequence of the module system is that public no longer means everyone can access a type. If a class is marked as public, it can be public only within a module (if its package is not exported) or only to specific modules that require it.

This promotes stronger encapsulation (the ability to hide parts of the code), which is good because we will no longer have access to internal (private) classes or private members (through reflection) that we are not supposed to use but occasionally change between versions and break our code.

Then, we have the problem of classloaders. Traditionally, the Java Virtual Machine (JVM) has used classloaders to load classes specified in the classpath.

Summary

* The *var-args* allows the method to accept zero or multiple arguments
* According to SUN Microsystem, static import improves the code readability and enhance coding
* The *for-each loop* was introduced in Java5 which is mainly used to traverse array or collection elements
* The main objective of enum is to define our own data types
* Since, marker interface contains no members, simply determining whether it is present or absent is sufficient
* Java has VAR keyword to declare variables that allows us to declare a variable without specifying its type