Programs Offered

Post Graduate Programmes (PG)

- Master of Business Administration
- Master of Computer Applications
- Master of Commerce (Financial Management / Financial Technology)
- Master of Arts (Journalism and Mass Communication)
- Master of Arts (Economics)
- Master of Arts (Public Policy and Governance)
- Master of Social Work
- Master of Arts (English)
- Master of Science (Information Technology) (ODL)
- Master of Science (Environmental Science) (ODL)

Diploma Programmes

- Post Graduate Diploma (Management)
- Post Graduate Diploma (Logistics)
- Post Graduate Diploma (Machine Learning and Artificial Intelligence)
- Post Graduate Diploma (Data Science)

Undergraduate Programmes (UG)

- Bachelor of Business Administration
- Bachelor of Computer Applications
- Bachelor of Commerce
- Bachelor of Arts (Journalism and Mass Communication)
- Bachelor of Arts (General / Political Science / Economics / English / Sociology)
- Bachelor of Social Work
- Bachelor of Science (Information Technology) (ODL)

AMITY UNIVERSITY ONLINE
Amity Helpline: (Toll free) 18001023434
For Student Support: +91 - 8826334455
Support Email id: studentsupport@amityonline.com | https://amityonline.com

Product code
81030112000585

Core Java

AMITY UNIVERSITY ONLINE

Core Java

Published by Amity University Press for exclusive use of Amity Directorate of Distance and Online Education, Amity University, Noida-201313

Contents

Module - I: Introduction to Java

Notes

Learning Objectives

At the end of this module, you will be able to:

- Understand the basics of Java
- Define features, identifiers and byte code in Java
- Understand JVM and JDK Tools
- Define various control flow statements
- Analyse command line arguments

Introduction

Java is a object oriented programming language created by James Gosling in 991. As, it is mentioned object oriented so we need to know what is object.

Object is nothing but real-world entity that we can touch like pen, pencil, chair, computer, dog etc. Object has state (which represents data) and behaviour (which represents functionality or method). For example dog has state like its name, colour, breed etc and behaviour like eating, barking etc.

1.1 Introduction of Java

Java runs on a variety of platforms, hence it is known as platform independent. Now, the question will come now what is platform. Here, platform means the operating system on which you will run your Java program such as Windows, Mac OS, UNIX etc. Java programming is little bit matured now, near about 23 years old. The target of Java language was to write a program once and run this program on multiple operating systems. The first publicly available version of Java (Java 1.0) was released in 1995. So, let us know about object first.

The developer claimed that the Java programming language is very simple, it is portable, very secure, high performed, multi threaded, interpreted, platform independent, dynamic architecturally neural, object oriented and finally, it is robust.

Advantages of learning Java Programming:

- Simple: Java is easy to learn if you understand the concepts of OOP (Object Oriented Programming).
- Object Oriented: In Java, everything is considered as an Object.
- Platform Independent: This term means Java is not compiled into platform specific machine. You can write the code once in one platform and can run it in other platforms without writing it several times for different platforms.
- Secure: Java enables to develop virus-free systems as all the pointers work internally and user cannot handle pointer externally.
- Architecture-neutral: Java compiler creates an architecture-neutral object, which makes the compiled code executable on many processors, ith the presence of Java runtime system.
- Portable: Java is portable as it gives us opportunity to carry the Java bytecode to any platform. It doesn't require any implementation.

Notes

- Robust: It uses strong memory management and has automatic garbage collection methods etc. That's why Java is called Robust which means strong.
- Multi threaded: Multi threaded because it can perform many tasks at the same time.
- Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- High Performance: Java enables high performance.
- Distributed: Java is distributed as it helps users to create distributed applications in Java. RMI(Remote Method Invocation) and EJB(Enterprise Java Beans) are used for creating distributed applications.
- Dynamic: Java is considered to be more dynamic as Java programs can carry extensive amount of run-time information which can be used to verify and resolve accesses to objects on run-time.

1.1.1 Overview of Java: Skeleton, History of Java

First time Java was introduced the 1991, there was a team named as the Green Team from the sun micro system lab, Sun system lab was very famous for developing hardware and software. The pioneer of this Green Team was James Gosling and his colleagues Mike Sheridan and Patrick Naughton.

They first time introduce the concept of object oriented programming. They give the name of the programming has Greentalk as it is from the green team. Later, they developed a more improved version of the concept and they gave the name as Oak. In 1995, Gosling introduced the name Java. Java is an island of Indonesia, where best coffee of the world is produced and Gosling was very fond of coffee and use to consume coffee while working on this project, that is why he chose the name Java for this programming language.

Then gradually Java became very popular and with this popularity in 1996, the Sun Micro system introduced a full set of programming environment, and they named it as JDK (Java Development Kit). So, this gives you a brief history of the Java and why the name of the programming language is Java.

History of Java

There is a lot of fascinating history to Java. Although Java was initially intended for interactive television, the digital cable television market at the time could not use such sophisticated technology. The Green Team is where Java's history began. The goal of this project, which was started by the Java team (also known as the Green Team), is to create a language for digital devices like televisions and set-top boxes. It worked well, though, for online programming. Later, Netscape integrated Java technology.

"Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic" were the guiding concepts behind the creation of Java programming. The man credited as the father of Java, James Gosling, created the language in 1995. The initiative was initiated in the early nineties by James Gosling and his colleagues.

Jason Gosling is the creator of Java.

Java is currently utilized in e-business solutions, games, mobile apps, and internet development. Important points describing Java's history are listed below.

Notes

1) In June 1991, James Gosling, Mike Sheridan, and Patrick Naughton started work on the Java language project. The Green Team is a tiny group of solar engineers.

2) It was first created for tiny embedded devices found in set-top boxes and other electronic appliances.

   The file extension was.gt, and James Gosling dubbed it "Greentalk" to start.

4) Following that, it was created as a component of the Green project and given the name Oak.

Why was Java given the name "Oak"?

Java's Origins from Oak to Java

5) Why Oak? An emblem of power, the oak tree is the national tree of numerous nations, including the United States, France, Germany, Romania, and others.

6) Because Oak Technologies had already registered the term as a trademark, Oak was rebranded as "Java" in 1995.

Why is the term "Java" for Java Programming?

7) What made them decide to call the Java language Java? To decide on a new name, the team got together. Words like "dynamic," "revolutionary," "Silk," "jolt," "DNA," and so on were suggested. They desired a name that captured the spirit of the technology: something innovative, energetic, dynamic, cool, one-of-a-kind, simple to spell, and enjoyable to pronounce.

   "Java was one of the top choices along with Silk," James Gosling claims. The majority of the team members chose Java over other names since it was so distinctive.

8) The first coffee was grown in the Indonesian island of Java, which is known as Java coffee. It is an espresso bean variety. James Gosling came up with the term Java while enjoying a cup of coffee close to his office.

9) Take note that Java is not an abbreviation; it is just a name.

10) Originally created in 1995 by James Gosling at Sun Microsystems, a division of Oracle Corporation, today.

11) Java was listed as one of the Ten Best Products of 1995 by Time magazine.

12) On January 23, 1996, JDK 1.0 was made available. Numerous new features have been introduced to Java since its first release. These days, Java is utilized in a variety of applications, including cards, Web, mobile, and Windows programs. Java features are added with each new version.

Version	Release Date	Major changes
JDK Beta	1995	
JDK 1.0	Jan-96	The Very first version was released on January 23, 1996. The principal stable variant, JDK 1.0.2, is called Java 1.
JDK 1.1	Feb-1997	Was released on February 19, 1997. There were many additions in JDK 1.1 as compared to version 1.0 such as: 1. A broad retooling of the AWT occasion show 2. Inner classes added to the language 3. JavaBeans 4. JDBC 5. RMI

Notes

J2SE 1.2	Dec-1998	"Play area" was the codename which was given to this form and was released on 8th December 1998. Its real expansion included: strictfp keyword 1. the Swing graphical API was coordinated into the centre classes 2. Sun's JVM was outfitted with a JIT compiler out of the blue 3. Java module 4. Java IDL, an IDL usage for CORBA interoperability 5. Collections system
J2SE 1.3	May-2000	Codename- "KESTREL" Release Date- 8th May 2000 Additions: 1. HotSpot JVM included 2. Java Naming and Directory Interface 3. JPDA 4. JavaSound 5. Synthetic proxy classes
J2SE 1.4	Feb-2002	Codename- "Merlin" Release Date- 6th February 2002 Additions: Library improvements 1. Regular expressions modelled after Perl regular expressions 2. The image I/O API for reading and writing images in formats like JPEG and PNG 3. Integrated XML parser and XSLT processor (JAXP) (specified in JSR 5 and JSR 63) 4. Preferences API (java.util.prefs) Public Support and security updates for this version ended in October 2008.
J2SE 5.0	Sep-2004	Codename- "Tiger" Release Date- "30th September 2004" Originally numbered as 1.5 which is still used as its internal version. Added several new language features such as: 1. for-each loop 2. Generics 3. Autoboxing 4. Var-args
JAVA SE 6	Dec-2006	Codename- "Mustang" Released Date- 11th December 2006 Packaged with a database supervisor and encourages the utilization of scripting languages with the JVM. Replaced the name J2SE with java SE and dropped the .0 from the version number. Additions: 1. Upgrade of JAXB to version 2.0: Including integration of a StAX parser. 2. Support for pluggable annotations (JSR 269). 3. JDBC 4.0 support (JSR 221)
JAVA SE 7	Jul-2011	Codename- "Dolphin" Release Date- 7th July 2011 Added small language changes including strings in the switch. The JVM was extended with support for dynamic languages. Additions: 1. Compressed 64-bit pointers. 2. Binary Integer Literals. 3. Upstream updates to XML and Unicode.
JAVA SE 8	Mar-2014	Released Date- 18th March 2014 Language level support for lambda expressions and default methods and a new date and time API inspired by Joda Time.
JAVA SE 9	Sep-2017	Release Date: 21st September 2017 Project Jigsaw: designing and implementing a standard, a module system for the Java SE platform, and to apply that system to the platform itself and the JDK.
JAVA SE 10	Mar-2018	Released Date- 20th March Addition: 1. Additional Unicode language-tag extensions

Notes

		2. Root certificates 3. Thread-local handshakes 4. Heap allocation on alternative memory devices 5. Remove the native-header generation tool – javah. 6. Consolidate the JDK forest into a single repository.
JAVA SE 11	Sep-2018	Released Date- 25th September, 2018 Additions- 1. Dynamic class-file constants 2. Epsilon: a no-op garbage collector 3. The local-variable syntax for lambda parameters 4. Low-overhead heap profiling 5. HTTP client (standard) 6. Transport Layer Security (TLS) 1.3 7. Flight recorder
JAVA SE 12	Mar-2019	Released Date- 19th March 2019 Additions- 1. Shenandoah: A Low-Pause-Time Garbage Collector (Experimental) 2. Microbenchmark Suite 3. Switch Expressions (Preview) 4. JVM Constants API 5. One AArch64 Port, Not Two 6. Default CDS Archives
JAVA SE 13	Sep-2019	Released Date – 17th September 2019 Additions- 1. Text Blocks (Multiline strings). 2. Switch Expressions. 3. Enhanced Thread-local handshakes.
JAVA SE 14	Mar-2020	Released Date – 17th March 2020 Additions- 1. Records (new class type for data modeling). 2. Pattern Matching for instanceof. 3. Helpful NullPointerExceptions.
JAVA SE 15	Sep-2020	Released Date – 15th September 2020 Additions- 1. Sealed Classes. 2. Hidden Classes. 3. Foreign Function and Memory API (Incubator).
JAVA SE 16	Mar-2021	Released Date – 16th March 2021 Additions- 1. Records (preview feature). 2. Pattern Matching for switch (preview feature). 3. Unix Domain Socket Channel (Incubator).
JAVA SE 17	Sep-2021	Released Date – 14th September 2021 Additions- 1. Sealed Classes (finalized). 2. Pattern Matching for instanceof (finalized). 3. Strong encapsulation of JDK internals by default. 4. New macOS rendering pipeline.

1.1.2 Features, Identifiers and Bytecode in Java

In programming languages, identifiers are used to identify through symbolic names. In Java, an identifier can be a class name, variable name, method name, package name, interface name or a label. Identifiers are names of variables, methods, classes, packages and interfaces. In the FirstProgram.java program, FirstProgram, String, args, main and println are all examples of identifiers.

Notes

Example:

```
class IdentifierExample{ void myMethod(){ System.out.println("Hello Java");
}
public static void main(String[] args){
int a = 95;
}
}
```

In the above java code, we have 5 identifiers namely :

- IdentifierExample: class name.
- myMethod, main : method name.
- System, String : predefined class name.
- args, a : variable name.

Rules for defining Java Identifiers

To define a valid java identifier you need to follow few rules. If rules are not followed then you will get compile time error.

- Alphanumeric characters are only allowed characters which includes ([A-Z], [a-z], [0-9]), '$'(dollar sign) and '_' (underscore).
- Example: "abc@" is not a valid java identifier as it contain '@' which is a special character.
- Identifiers should not start with digits ([0-9]). Example: "12abc" is not a valid java identifier.
- Java identifiers are case-sensitive i.e. myVariable, MyVariable and Myvariable are different from each other.
- Though no limit is there to specify the length of the identifier but it is always better if you use an optimum length of 4 to 15 letters only.
- You cannot use Reserved Words as an identifier.
- Example: "int while = 20;" is an invalid statement as "while" is a reserved word.

There are 53 reserved words in Java.

Examples of valid identifiers

Below are listed few valid identifiers.

MyVariable

MYVARIABLE

myvariable

a

i x

p1

i1

_myvariable

$myvariable

total_salary

Notes

salary123

Examples of invalid identifiers

My Variable: as it contains space between My and Variable

123 salary: as it begins with a digit

var-5: as hyphen is not an alphanumeric character

p+q: as plus sign is not an alphanumeric character

a&b: as ampersand is not an alphanumeric character

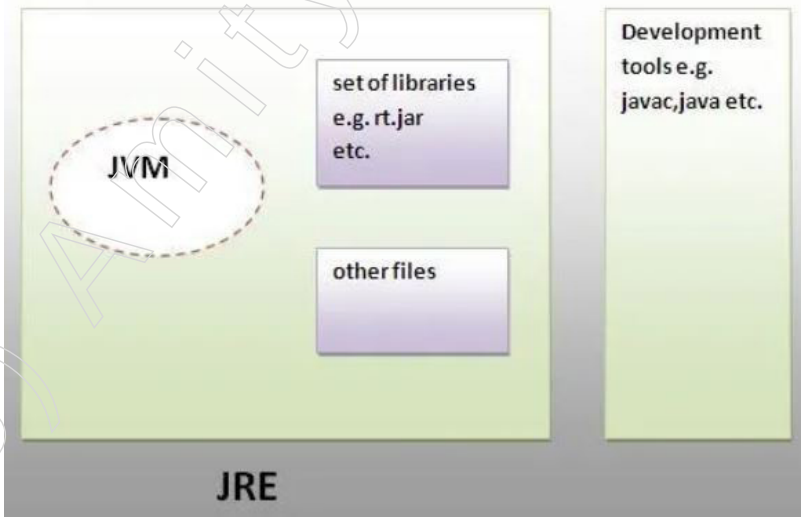
Java bytecode is the result of the compilation of a Java program which the Java Virtual Machine (JVM) understands and which is machine independent. On compilation of a Java program, a byte code has to be produced only once and after that it can run on any platform wherever a Java Virtual Machine exists. Bytecode files have a .class extension.

In above example, you can see we have used javac which is the compiler. In command prompt, we have to write javac then file name with extension to compile your Java program. So, we have to give the name of the file in the same way as the name of the class you have given and it is a case sensitive so, be careful about that. Now, let us compile it. So, as there is no error, no message in the command prompt, this means that this program has successfully compiled.

Once on the successful compilation, you can see in the same directory one file is created, the name of the file is the same as the name of the Java file except the extension is .class. So, here you can see the byte code file which has been created is FirstProgram.class. So, once this program is successful on compilation now we are ready to run it, to run this program the command that we said to use it java. So, Java first program FirstProgram and then .class you can use the .class or even if you use the class also no issue so it will run.

So, here, for example, the class file name is FirstProgram. Simply type java and the name of the class file, namely FirstProgram here. So, this is the program that has been executed and as you see this program is basically used only one statement namely System. out.println() and within this println() is basically print the statement “This is my first Java program”.

1.1.3 JVM and JDK Tools



JVM: JVM (Java Virtual Machine) is an machine which does not exist physically, we can say it is an abstract machine or a specification that provides runtime environment in

Notes

which java bytecode can be executed. As we already know that Java language is platform independent but JVM is platform dependent. JVM calls the main method present in a java code. JVM is a part of Java Runtime Environment (JRE).

The JVM has different tasks. It loads code, verifies code, executing code and it provides runtime environment. As you can see in the picture JVM lies inside JRE (Java Runtime Environment) which provides runtime environment and is the implementation of JVM. It physically exists. It contains a set of libraries and other files that JVM uses at runtime.

We can see in the picture that JRE lies inside JDK (Java Development Kit). JDK is a software package and you need to download it to develop Java applications and applets. It contains JRE and other development tools. The JDK includes the JRE, an archiver (jar), a compiler (javac), an interpreter (java), a documentation generator (javadoc), and few more development tools.

Standard JDK Tools and Utilities

- Basic Tools: applet viewer, extcheck, jar, java, javac, javadoc, javah, javap, jdb

Tool Name	Brief Description
appletview-er	Run and debug applets without a web browser.
Apt	Annotation processing tool.
extcheck	Utility to detect Jar conflicts.
Jar	Create and manage Java Archive (JAR) files.
Java	The launcher for Java applications. In this release, a single launcher is used both for development and deployment. The old deployment launcher, jre, is no longer provided.
javac	The compiler for the Java programming language.
javadoc	API documentation generator.
javah	C header and stub generator. Used to write native methods.
javap	Class file disassembler
Jdb	The Java Debugger.

- Security Tools: keytool, jarsigner, policytool, kinit, klist, ktab

Tool Name	Brief Description
keytool	Manage keystores and certificates.
jarsigner	Generate and verify JAR signatures.
policytool	GUI tool for managing policy files.

These security tools help you obtain, list, and manage Kerberos tickets.

Tool Name	Brief Description
Kinit	Tool for obtaining Kerberos v5 tickets. Equivalent functionality is available on the Solaris operating system via the kinit tool.
Klist	Command-line tool to list entries in credential cache and key tab. Equivalent functionality is available on the Solaris operating system via the klist tool.

Ktab	Command-line tool to help the user manage entries in the key ta-ble. Equivalent functionality is available on the Solaris operating system via the kadmin tool.

Notes

- Internationalization Tools: native2ascii

Tool Name	Brief Description
native2ascii	Convert text to Unicode Latin-1.

- **Remote Method Invocation (RMI) Tools:** rmic, rmiregistry, rmid, serialver

Tool Name	Brief Description
Rmic	Generate stubs and skeletons for remote objects.
rmiregistry	Remote object registry service.
Rmid	RMI activation system daemon.
serialver	Return class serialVersionUID.

- **Java IDL and RMI-IIOP Tools:** tnameserv, idlj, orbd, servertool

Tool Name	Brief Description
tnameserv	Provides access to the naming service.
Idlj	Generates .java files that map an OMG IDL interface and enable an application writ-ten in the Java programming language to use CORBA functionality.
Orbd	Provides support for clients to transparently locate and invoke persistent objects on servers in the CORBA environment. ORBD is used instead of the Transient Nam-ing Service, tnameserv. ORBD includes both a Transient Naming Service and a Per-sistent Naming Service. The orbd tool incorporates the functionality of a Server Manager, an Interoperable Naming Service, and a Bootstrap Name Server. When used in conjunction with the servertool, the Server Manager locates, registers, and activates a server when a client wants to access the server.
servertool	Provides ease-of-use interface for the application programmers to register, un-register, startup, and shutdown a server.

- **Java Deployment Tools:** javafxpackager, pack200, unpack200. Utilities for use in conjunction with deployment of java applications and applets on the web.

Tool Name	Brief Description
javafxpackager	Packages JavaFX applications for deployment.
pack200	Transforms a JAR file into a compressed pack 200 file using the Java gzip com-pressor. The compressed packed files are highly compressed JARs, which can be directly deployed, saving bandwidth and reducing download time.
unpack200	Transforms a packed file produced by pack200 into a JAR file.

- **Java Web Start Tools:** javaws

Tool Name	Brief Description
javaws	Command line tool for launching Java Web Start and setting various options.

- **Java Troubleshooting, Profiling, Monitoring and Management Tools:** jcmd, jconsole, jmc, jvisualvm

Notes

Tool Name	Brief Description
Jcmd	JVM Diagnostic Commands tool - sends diagnostic command requests to a running Java Virtual Machine.
jconsole	A JMX-compliant graphical tool for monitoring a Java virtual machine. It can monitor both local and remote JVMs. It can also monitor and manage an application.
Jmc	The Java Mission Control (JMC) client includes tools to monitor and manage your Java application without introducing the performance overhead normally associat-ed with these types of tools.
jvisualvm	A graphical tool that provides detailed information about the Java technology-based applications (Java applications) while they are running in a Java Virtual Ma-chine. Java VisualVM provides memory and CPU profiling, heap dump analysis, memory leak detection, access to MBeans, and garbage collection

- **Java Web Services Tools:** schemagen, wsgen, wsimport, xjc

Tool Name	Brief Description
schemagen	Schema generator for Java Architecture for XML Binding.
wsgen	Tool to generate JAX-WS portable artifacts.
wsimport	Tool to generate JAX-WS portable artifacts.
xjc	Binding compiler for Java Architecture for XML Binding.

1.2 Basic Concept of Java Programming

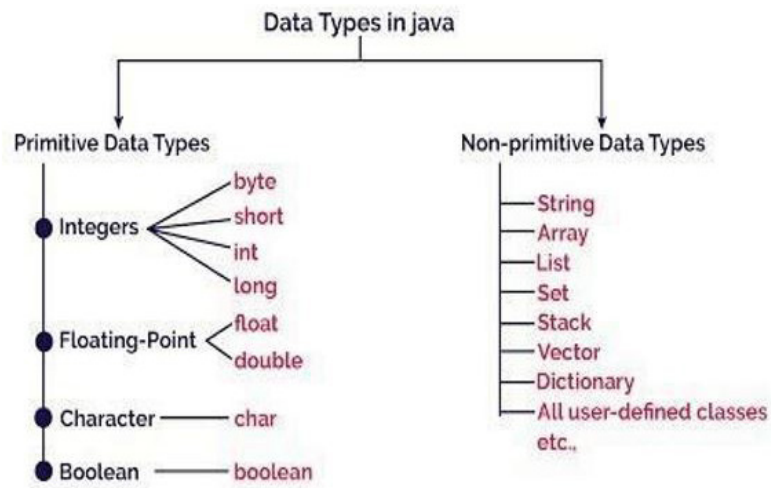
Java is a flexible object-oriented programming language that was created to be platform-neutral and enable programmers to create apps that can function across a range of devices without the need for customisation.

1.2.1 Data Types of Java

Data types of the Java programming language are divided into two categories: primitive types and non-primitive or reference or user defined types. A primitive data type means the data types which are predefined and provided to us by the Java programming language.

The primitive are the numeric types, boolean type and character type.

The numeric types contain integral types and floating-point types. Integral types contain byte, short, int, long and the floating-point types contain float and double.



Notes

The non-primitive or reference types are class, interface, array etc. which user creates as their own. String literals are represented by String objects. Picture 2 depicts the data types available in Java.

Integral Types and Values

Integral types have values in the below ranges:

- The byte data type has values ranging from -128 (-27)to 127 (27– 1), inclusive. Default value is 0. For example, byte a = 10 or byte b = -20
- For short data type, range from -32768 (216) to 32767(216 – 1), inclusive. Default value is 0. For example; short s = 20000, short r = -6500
- For int data type, range from -2147483648 (-231) to 2147483647 to (231– 1), inclusive. Default value is 0. int a = 100000, int b = -200000
- For long data type, range from -9223372036854775808 (-263) to 9223372036854775807 (263 – 1), inclusive. Default value is 0L. long a = 100000L, long b = -150000L

Example:

- int myNum = 5;// Integer (whole number)

Wrapper Class: Byte

- Minimum value: -128 (-2^7)
- Maximum value: 127 (2^7 -1)
- Default value: 0
- Example: byte a = 10 , byte b = -50;

Wrapper Class: Short

- Minimum value: -32,768 (-2^15)
- Maximum value: 32,767 (2^15 -1)
- Default value: 0.
- Example: short s = 10, short r = -1000;

Wrapper Class: Integer

- Minimum value: (-2^31)
- Maximum value: (2^31 -1)
- The default value: 0

Example: int a = 50000, int b = -20

Wrapper Class: Long

- Minimum value: (-2^63)
- Maximum value: (2^63 -1)
- Default value: 0L.
- Example: long a = 100000L, long b = -600000L;

Floating-Point Type

There are two floating-point data types which are double and float. The float data type is characterized by single-precision 32-bit and double data type is double precision 64-bit IEEE 754 floating point values. Precision means how many digits the value can have after

Notes

the decimal point. It is recommended to use a float if you want to save memory area in large arrays of floating point numbers. The float data type stores fractional numbers which is capable to store 6 to 7 decimal digits whereas the double data type stores fractional numbers which is capable to store 15 decimal digits. Default value of float data type is 0.0f and for double it is 0.0d. The float data type can store fractional numbers ranging from 3.4e−038 to 3.4e+038 and double data type can store fractional numbers ranging from 1.7e−308 to 1.7e+308.

Example of float and double:

float f1 = 234.5f

double d1 = 12.3

Wrapper Class: Float

- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value: 0.0f.
- Example: float f1 = 24.5f

Character Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (65,535 inclusive).The char data type is used to store characters.

Example: char ex = 'A

Wrapper Class: Character

- Minimum value: '\u0000' (or 0).
- Maximum value: '\uffff' (or 65,535).
- Default value: null ('\u0000').
- Example: char letterA ='a';

Boolean Type

The boolean data type is used to store two possible values: true and false. This data type specifies one bit of information, but its "size" can't be defined precisely.

Example: boolean a = false

Wrapper Class: Boolean

- This data type is used for simple flags that track true/false conditions.
- Default value is false

1.2.2 Control Flow Statements: Part 1

The statements inside your source files are generally executed from top to bottom. Control flow statements break up the flow of execution for decision making, looping, and branching and enable your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if, if-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue) supported by the Java programming language.

The if and if-else Statements

The most common control flow statement is "if statement". It tells a particular block of code to execute only if a condition checking produces true result. A car can allow the brakes

Notes

to decrease its speed only when the car is in motion. One possible implementation of the apply Brake method can be written as:

```
void applyBrake() {
 if (isMoving)    // we have to apply break when car is moving
 {
//if the car is moving then we can decrease its speed
 speed--;
 }
 }
```

If this test evaluates to true then only we can change the value of speed variable by 1 but if result is false that means that the car is not in motion then the control jumps to the end of the if statement.

The if-else Statement

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

```
void applyBrake() {
if (isMoving) {
speed--;
} else {
System.err.println("The car has already stopped!");
}
}
```

Example:

```
// A Java program to demonstrate the if-else statement
class IfElseDemo {
    public static void main(String args[])
    {
        int i = 20;
        if (i < 15)
            System.out.println("i is smaller than 15");
             else
            System.out.println("i is greater than 15");
        System.out.println("Outside if-else block");
    }
}
```

Output

Notes

```
i is greater than 15
Outside if-else block
```

1.2.3 Control Flow Statements: Part 2

The switch Statement

The Java switch statement executes one statement from multiple conditions. It is like if-else-if kind of statement. The switch statement works with primitive data types like byte, short, int, long. It can work with wrapper class types like Byte, Short, Integer and Long, it also works with enumerated types. It can also work with String class. In other words, the switch statement tests the equality of a variable against multiple values.

Example: SwitchTest.java

```
public class SwitchTest {
public static void main(String[] args) {
int n=35;
switch(n){
case 10: System.out.println(“ Value is 7”);
break;
case 20: System.out.println(“Value is 21”);
break;
case 30: System.out.println(“Value is 35”);
break;
default: System.out.println(“Value is not 7, 21 or 35”);
}
}
}
```

In the above example we have used break statement. When a break statement is encountered, flow is immediately terminated and the program control resumes at the next statement. The Java break statement is used to break switch or loop statement. It breaks the current flow of the program at specified condition.

Example:

```
public class SizePrinter {
    public static void main(String[] args) {
        int sizeNumber = 2; // Replace with your desired size (1, 2, 3, 4, or 5)
            switch (sizeNumber) {
                case 1:
                    System.out.println(“Extra Small”);
                    break;
                case 2:
                    System.out.println(“Small”);
                    break;
```

Notes

```
            case 3:
               System.out.println("Medium");
               break;
            case 4:
               System.out.println("Large");
               break;
            case 5:
               System.out.println("Extra Large");
               break;
            default:
               System.out.println("Invalid size number");
         }
      }
   }
```

Output:

Small

1.2.4 Loops in Java

In any programming language, we use loops to execute a set of instructions repeatedly when some conditions become true. The control flow goes out of the loop only when the condition will become false. Here, we are going to discuss three types of loops used in Java.

- for loop
- while loop
- do-while loop

The for Loop

The for loop is a control flow statement that iterates a part of the programs multiple times.

Example:

```
for(int a =0;  a<=10;          a++){
System.out.println(a);
 }
```

Example

```
// Java program to illustrate for loop.
class forLoopDemo {
    public static void main(String args[])
    {
    int sum = 0;
```

Notes

```
        // for loop begins
        // and runs till x <= 20
        for (int x = 1; x <= 20; x++) {
        sum = sum + x;
        }
        System.out.println("Sum: " + sum);
       }
}
```

Output

210

Nested for loop

When we have a for loop inside another for loop the we will call it a nested for loop.

Example:

```
 for(int a=1;a<=3;a++)  //outer for loop
 {
for(int b=1;b<=3;b++)   //inner for loop
 {
System.out.println "Value of a: " + a +"and value of b: "+b);
}//end of inner loop
}//end of outer loop
```

Example

```
// Java Program to implement
// Nested for loop
import java.io.*;
// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        // Printing a 1 to 5 (5 times)
        // first loop
        for (int i = 1; i <= 5; i++) {
            // second loop
            for (int j = 1; j <= 5; j++) {
            System.out.print(j + " ");
            }
            System.out.println();
```

```
        }
        }
}
```

Output

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

The for-each loop

The for-each loop is used to traverse array or collection in java. It is easier to use than in such cases because we don't need to increment value and use subscript notation. It works on elements basis and not index basis. It returns element one by one in the defined variable.

Example: LoopExample.java

```
//print the elements of a given  array
public class LoopExample {
public static void main(String[] args) {
int myArray[]={12,23,44,56,78};   //Declaring an array
//Printing the values of the array using for-each loop
for(int a : myArray){
System.out.println(a);
}
}
}
```

The while and do-while loop

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

Example of while loop:

```
int a =0; //initialisation
while (a<=10)  //condition checking
 {
 System.out.println(a);
 a++; //increment
 }
```

Note: The last line before the ending of brace, we have to increment the value of the variable

Example of do-while loop:

Notes

```
int a =0; //initialisation
do{
System.out.println(a);
a++;                    //increment
} while (a<=10); // condition checking
```

Note: In do-while loop, you must end while statement with a semicolon.

The while statement evaluates expression first, which actually return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false but when you are working with do-while loop then first time the loop will get executed without checking the condition and condition checking will be done afterwards.

Example:

```
// Java Program to Illustrate One Time Iteration
class abc {
   public static void main(String[] args)
   {
      // initial counter variable
      int i = 0;
      do {
         // Body of loop that will execute minimum
         // 1 time for sure no matter what
         System.out.println("Print statement");
         i++;
      }
      // Checking condition
      // Note: It is being checked after
      // minimum 1 iteration
      while (i < 0);
   }
}
```

Output

Print statement

1.2.5 Operators in Java

In Java language, operators perform some operations and which are nothing but symbols. +, -, *, / etc. are all examples of operators. Arithmetic operation, magnitude and tests for equality are common examples of expressions as after operation they return a value and you can assign that result to a variable or you can put the value to test in other Java statements. Arithmetic operator, assignment operator, increment and decrement operator and logical operator are commonly used operators in Java.

Notes

Arithmetic Operators

Java has five operators for basic arithmetic as described in below table.

Operator	Meaning	Example
+	Addition	10+12
-	Subtraction	20-4
*	Multiplication	8*7
/	Division	14/2
%	Modulus	100%8

Each operator takes two operands and one operator. To negate a single operand you can use the subtraction (-) operator. If you are performing integer division, it will return an integer value as integers don't have decimal fractions, thus is any remainder is coming that will be ignored. For example the expression 52/ 7 gives results as 7. The role of the modulus (%) operator is to return the remainder of two numbers. For example, 31% 9 gives results as 4 because modulus gives the result which is the remainder of your division operation.

Note that, the result of most operations involving integer values will produce an int or long values, regardless of the original type of the operands. If large values are coming as result then it is of type long; rest are int. Arithmetic operation involving one operand as integer and another as floating point will return a floating-point result.

Example: ArithmeticProgram.java

```
class ArithmeticProgram {
public static void main (String[] args) {
short x = 6;
int y = 4;
float a = 12.5f;
float b = 7f;
System.out.println("x is " + x + ", y is " + y);
System.out.println("x + y = " + (x + y));
System.out.println("x - y = " + (x - y));
System.out.println("x / y = " + (x / y));
System.out.println("x % y = " + (x % y));
System.out.println("a is " + a + ", b is " + b;
System.out.println("a / b = " + (a / b));
}
}
```

Output:

```
x is 6, y is 4
x + y = 10
```

Notes

x - y = 2

x / y = 1

x % y = 2

a is 12.5, b is 7

a / b = 1.78571

The System.out.println() method prints a message to the standard output of your system. This method takes a single argument—a string—but you can use + sign to concatenate values into a string, which you'll learn later on.

We can assign value to variable in the form of expression. If you write x = y = z = 10, you can tell that all three variables now have the value 10. Also, the right side of an assignment expression is always evaluated before the assignment operation takes place. This means that expressions such as x = x + 5 do the right thing i.e. 5 is added to the value of x, and then that new value is reassigned to x. This kind of operation is very common hence Java has several operators to do a shorthand version which is borrowed from C and C++. Below table shows these shorthand assignment operators.

Assignment Operators

Expression	Meaning
x += y	x = x + y
x –= y	x + x – y
x *= y	x = x * y
x ¸= y	x = x ¸ y

Incrementing and Decrementing

To increase or decrease a value by 1, we use + + and – – operators. For example, x++ increments the value of x by 1 which actually equivalent to x = x + 1 expression. Similarly, x– – decrements the value of x by 1. These increment and decrement operators can be prefixed i.e. ++ or -- can appear before the value or postfixed i.e. ++ or – – can appear after the value it increments or decrements.

For simple increment or decrement expressions, which form you use is not overly important but in case of complex assignments which form you use makes a difference because you are assigning the result of an increment or decrement expression.

Let us consider the following two expressions:

y = x++;

y = ++x;

These two expressions will produce different results because of the difference between prefix and postfix. The postfix operators (x++ or x--) first return the variable value, then increment or reduce the value of the variable. The prefix operators (++x or –x) first increment or decrease the value of a variable and then returns value of the variable.

Example of prefix and postfix increment operators:

```
class PrePostFixProgram {

public static void main (String[] args) {

int x = 0;
```

Notes

```
int y = 0;
System.out.println("x          and y are " + x   + " and  " + y );
x++;
System.out.println("x++ results in " + x);
++x;
System.out.println("++x results in " +          x);
System.out.println("Resetting x back to 0.");
x = 0;
System.out.println("————————");
y    =     x++;
 System.out.println("y          = x++ (postfix)   results   in:");
 System.out.println("x          is " + x);
 System.out.println("y          is " + y);
System.out.println("————————");
y = ++x;
System.out.println("y = ++x (prefix) results in:");
System.out.println("x is " + x);
System.out.println("y is " + y);
System.out.println("————————");
}
}
x and y are 0 and 0
 x++       results       in         1
++xresults    in        2
 Resetting x back to 0.
 ————————
 y   =     x++ (postfix)          results   in:
```

The postfix operator ++ adds one to its operand / variable and returns the value only after it is assigned to the variable. In other words, the assignment takes place first and the increment next.

```
x is 1
 y is 0
 ————————
 y   =     ++x (prefix)          results   in:
```

The prefix operator ++ adds one to its operand / variable and returns the value before it is assigned to the variable. In other words, the increment takes place first and the assignment next

```
x is 2
```

Notes

y is 2

_____

Java has several expressions for testing equality and magnitude. All of these expressions return a boolean value (that is, true or false). Below table shows the comparison operators:

Comparison operators

Operator	Meaning	Example
=	Equal	x == 3
!=	Not equal	x != 3
<	Less than	x < 3
>	Greater than	x>3
≤	Less than or equal	x ≤3
≥	Greater than or equal	x ≥ 3

Logical Operators

If there is a chance to get result in boolean values of Expressions (for example, the comparison operators) then it can be combined by using logical operators that represent the logical combinations AND, OR, XOR, and logical NOT.

AND expression

We use either & or && for AND combination. && is known as logical AND operator. In an expression it will return true if both the statements are true. If first condition is false then second condition will not be checked. Second condition will only be checked if the first condition is true.

& is known as bitwise operator and this operator always checks both conditions whether first condition is true or false.

Example:

```
int a=15;
int b=5;
int c=30;
System.out.println(a<b && a<c);
//here (a<b) is false and (a<c) is true hence resulting false
System.out.println(a<b & a<c);
//here (a<b) is false and (a<c) is true hence resulting false
```

NOT expression

For NOT, use the ! operator with a single expression argument. The value of the NOT expression is the negation of the expression; if x is true, !x is false.

Example:

```
!1 = 0
!0 = 1
```

Notes

Bitwise Operators

These are all inherited from C and C++ and are used to perform operations on individual bits in integers. This book does not go into bitwise operations; it's an advanced topic covered better in books on C or C++. Below table summarizes the bitwise operators

The expression will produce true result only if both operands true; if either expression is false, the entire expression will produce false result as you can see in the above example. You can find the difference between these two operators during expression evaluation.

Bitwise operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment (x = x << y)
>>=	Right shift assignment (x = x >> y)
>>>=	Zero fill right shift assignment (x = x >>> y)
x&=y	AND assignment (x = x & y)
x|=y	OR assignment (x + x | y)
x^=y	NOT assignment (x = x ^ y)

OR expression

For OR expressions, we use either | or ||. In an expression, if first condition is true then the logical || operator does not check second condition and if the first condition is false then only it checks second condition.

The bitwise | operator always checks both conditions regardless first condition is true or false.

In addition, there is the XOR operator (^), which returns true value only if its operands are different i.e. if there is one true value and one false value (or vice versa) and returns false if both are true or both are false. In general, only the && and || are commonly used as actual logical combinations. &, |, and ^ are more commonly used for bitwise logical operations.

Example:

```
 int a=40;
int b=25;
int c=50;
System.out.println(a>b||a<c);
 //(a>b) is true here and (a<c) is also true so, true || true = true
```

Notes

```
System.out.println(a>b|a<c);
 //(a>b) is true here and (a<c) is also true so, true | true = true
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//40  will be printed because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//41 will be printed because second condition is checked
// Java program to illustrate
// bitwise operators

public class operators {
    public static void main(String[] args)
    {
        // Initial values
        int a = 5;
        int b = 7;
        // bitwise and
        // 0101 & 0111=0101 = 5
        System.out.println("a&b = " + (a & b));
        // bitwise or
        // 0101 | 0111=0111 = 7
        System.out.println("a|b = " + (a | b));
        // bitwise xor
        // 0101 ^ 0111=0010 = 2
        System.out.println("a^b = " + (a ^ b));
        // bitwise not
    // ~00000000 00000000 00000000 00000101=11111111 11111111 11111111 11111010
        // will give 2's complement (32 bit) of 5 = -6
        System.out.println("~a = " + ~a);
        // can also be combined with
        // assignment operator to provide shorthand
        // assignment
        // a=a&b
        a &= b;
        System.out.println("a= " + a);
    }
}
```

Output

```
a&b = 5
a|b = 7
a^b = 2
```

~a = -6

a= 5

Notes

Operator Precedence

If an expression contains multiple operators then there are a number of rules to decide the order in which the operators will be evaluated. The most important rule is known as operator precedence. Operators which have higher precedence are executed before than the operators with lower precedence. For example, multiplication has a higher precedence than subtraction or addition. In the expression 5+2*3, the multiplication will be done before the addition which in turn will produce a result of 14. When operators of equal precedence appear in the same expression, binary operators except for the assignment operators are evaluated from left to right and assignment operators are evaluated right to left. In general, increment and decrement are evaluated before arithmetic expressions are evaluated, arithmetic expressions are evaluated before comparisons, and comparisons are evaluated before logical expressions. Assignment expressions are evaluated last.

[], () Parentheses group expressions; dot (.) is used for access to methods and variables within objects and classes. [] is used for arrays. ++ – – ! ~ instance of Returns true or false based on whether the object is an instance of the named class or any of that class's superclasses.

String Arithmetic

One special expression in Java is the use of the addition operator (+) which we will use to create and concatenate strings.

Example:

String name = "Tojo";

String colour ="white";

Int age = 10;

System.out.println(name + " is " + colour + " in colour whose age is "+ age +"years");

Output:

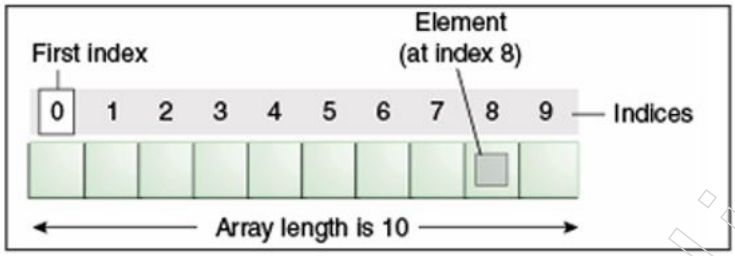
Tojo is white in colour whose age is 10 years

In this example you can see the output of this line has come in a single line as a string, with the values of the variables (name and colour) embedded on it. The concatenation (+) operator, when used with strings and other objects, creates a single string that contains the concatenation of all its operands. If any of the operands in string concatenation is not a string (age here is not String), it is automatically converted to a string, making it easy to create these sorts of output lines.

1.2.6 Arrays

A collection of similar type of elements when put in contiguous memory location is known as array. In Java, array is an object which contains elements of a similar data type. Array is a kind of data structure where we store similar elements. Array follows index-based structure, the first element of the array is stored at 0th index, and 2nd element is stored at 1st index and so on. In Java programming language, array is an object of a dynamically generated class. We can store primitive values or objects in an array in Java. You can also create one dimensional or multidimensional array in Java. In other words, we can say that array is a container object that holds a fixed number of values of same type.

Notes



Element is accessed by its numerical index. As shown in Figure above, indexing begins with 0. The 9th element, for example, would therefore be accessed at index 8.

When we want to store multiple values in a single variable then we will use array, rather than declaring separate variables for each value. To declare an array we need to define the variable type with square brackets. For example, String[] car; or you can write as String car[] (though it will work but not preferred). Here, car is the name of the variable that holds an array of strings..

Creating Arrays

We can create an array by using the new operator with the following syntax

Syntax: myArray = new dataType[arraySize];

The above statement fulfils two purposes:

- Creates an array using new dataType[arraySize].
- Assigns the reference of the newly created array to the variable myArray.

Declaring, creating and assigning the reference of the array to the variable can be combined in one statement as written below.

dataType[] myArray = new dataType[arraySize];

In the above case you are only creating an empty array with specified size.

Alternatively you can create arrays as follows,

dataType[] myAarray = {value 0, value 1, ..., value n};

In the above case you are creating an array as well as putting values to it.

The array elements are accessed through the index value. Array index starts from 0 to myArray.length-1.

Declaring an Array

An array declaration has two parts: First is type of array and the second is name of array. The way to write data type of array is written as type[]. The brackets are special symbols differentiating from other variable and indicating it as array variable. An array's name can be given as per user's choice maintaining the rules and conventions.

Example:

Below example allocates an array with memory for 10 integer elements and assigns the array to the myArray variable.

```
// create an array of integers
 myArray = new int[10];
```

Following statement declares an array variable, myArray and creates an array of 10 elements of double type and assigns its reference to myArray variable.

Notes

```
// create an array of integers
 double[] myArray = new double[10];
 Example: ArrayExample.java
 class ArrayExample {
public static void main(String[] args){
int arr[]={40,50,60,70}; //Array declaration, instantiation and initialization
//printing array
for(int i=0; i<arr.length; i++)
System.out.println(arr[i]);
}
 }
```

Note: length is the property of array. So, we can access length of array by mentioning it as arr.length.

1.2.7 Command Line Arguments

The java command-line argument is an argument that is passed at the time of running the java program. A Java application can accept any number of arguments from the command line. The command line argument is the argument that passed to a program during runtime. It is the way to pass argument to the main method in Java. The arguments passed from the console can be received in the java program and it can be used as an input. So, it provides a convenient way to check the behaviour of the program for the different values. You can pass any numbers of arguments from the command prompt.

Example: CmdExample.java

```
 class CmdExample{
public static void main(String[] args){
for(int i=0;i<args.length;i++)
System.out.println(args[i]);
}
}
compile by > javac CmdExample.java
run by > java CmdExample Tea Coffee Pepsi Fruti
```

Output:

```
 Tea
Coffee
Pepsi
Fruti
```

In the above example, Tea will be stores in args[0], Coffee will be stores in args[1], Pepsi will be stores in args[2], Fruti will be stores in args[3]

Note: This program will give output as Tea, Coffee, Pepsi, Fruti on a line by itself. This

Notes

is because the space character separates command-line arguments. To have Tea Coffee Pepsi Fruti as a single argument, the user would join them by enclosing them within quotation marks as java Echo " Tea Coffee Pepsi Fruti"

1.2.8 Inheritance in Java

The way we inherit properties from our parents, like the same way a class can acquire properties from another class. This mechanism of inheriting properties of one object from a parent object is called inheritance. A class that is inheriting properties from another class is called a subclass or a child class or a derived class. The class from which the subclass is derived is called a superclass, or a parent class or a base class. Inheritance represents IS-A relationship which is also known as a parent-child relationship.

Inheritance provides the reusability property. Reusability is a mechanism which facilitates us to reuse the properties (fields and methods). You can use the same fields and methods already defined in the previous class.

We have to use extends keyword to acquire the property of parent class to child class. The meaning of "extends" is to increase the functionality.

Example:

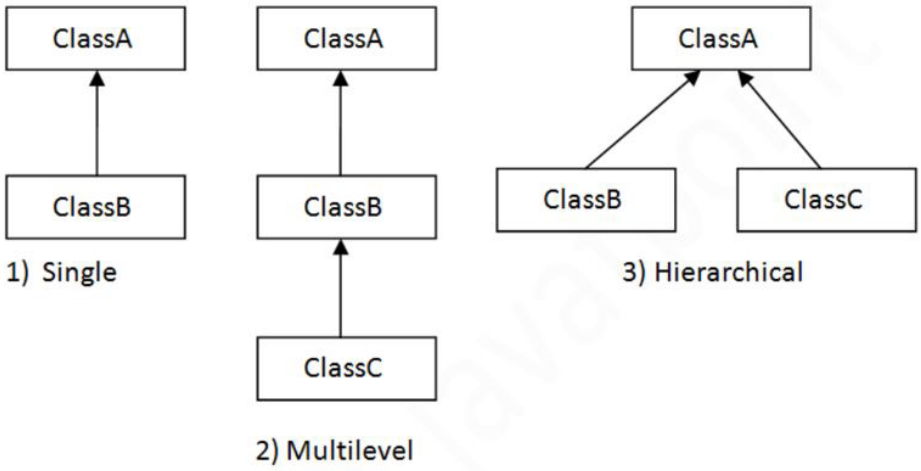
```
 class Employee{
 int salary=35000;
}
class Tester extends Employee{
 System.out.println("Tester's salary is:"+ salary);
}
 public class Programmer extends Employee{
int bonus=15000;
 public static void main(String[] args){
Programmer p=new Programmer();  //creating object of Programmer class
Int totalSalary =  salary +bonus;
 System.out.println("Programmer's total salary is:"+totalSalary);
}
}
```

In the above example, we have written three classes, Employee class, Tester class and Programmer class. We have defined salary (variable) inside the Employee class (parent class) but we are accessing the same salary variable from Tester class (child class of Employee class) and Programmer class (child class of Employee class). Hence, here we are using the inheritance property of Java. Also, note that we have mentioned about IS-A relationship and for this example we can say Tester IS-A Employee or Programmer IS-A Employee.

Types of Inheritance

In java programming, there are three types of inheritance in java: single, multilevel and hierarchical. There are also multiple and hybrid inheritance is supported through interface only. In Java, multiple inheritance is not supported through class.

Notes



Source: https://www.javatpoint.com/inheritance-in-java

Single Inheritance: Inheritance in Java When a class inherits properties from another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Example: InheritanceExample.java

```
class Animal{
void eat(){
 }
Example: InheritanceExample.java
class Animal{
void eat(){
System.out.println(“ Animals eat.”); }
}
class Horse extends Animal{
void run(){
System.out.println(“Horse gallops.”); }
System.out.println(“ Animals eat.”); }
class Horse extends Animal{
void run(){
System.out.println(“Horse gallops.”); }
}
class Lion extends Animal {
void roar(){
System.out.println(“Lions roar.”); }
}
public class InheritanceExample{
public static void main(String args[]){
Horse obj1 = new Horse();
Lion obj2 = new Lion();
```

Notes

```
obj1.run();
obj1.eat();
obj2.eat();
obj2.roar();
}
}
```

Multilevel Inheritance

When there is a chain of inheritance, it is known as multilevel inheritance. In the given example we can see BabyDog class inherits from the Dog class and Dog class inherits from the Animal class, so there is a multilevel inheritance.

Example: InheritanceTest.java

```
class Animal{
void eat(){
System.out.println("Animals eat.");}
}
class Dog extends Animal{
void bark(){
System.out.println("Dog barks.");}
}
class BabyDog extends Dog{
void wag(){
System.out.println("Baby dog wags");}
}
public class InheritanceTest{
public static void main(String args[]){
BabyDog obj=new BabyDog();
obj.wag();
obj.bark();
obj.eat();
}
}
```

Hierarchical Inheritance

When two or more classes inherits from a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Example: InheritanceExample.java

Notes

```
class Flower{
void petal(){
System.out.println("Flowers have petals.");}
```

```
}
class Rose extends Flower{
void smell(){
System.out.println("Rose has beautiful smell.");}
}
class Tulip extends Flower{
void colour(){
System.out.println("Tulip has beautiful colour.");}
}
public class InheritanceExample{
public static void main(String[] args){
Rose obj1 = new Rose();
obj1.petal();
obj1.smell();
Tulip obj2=new Tulip();
obj2.petal();
obj2.colour();
 }
 }
```

Note: You cannot write obj1.colour() as there is no inheritance relationship between Rose and Tulip class.

Summary

- Java is a simple language because its syntax is simple and easy to understand. Complex and ambiguous concepts of C++ are either eliminated or re-implemented in Java. Like pointer and operator overloading are not used in Java.
- Java is Object Oriented language and everything is in the form of the object here. It means it has data (state) and method (behaviour). Java contains a security manager that defines the access of Java classes. Platform-Independent: Java provides a guarantee that code writes once and run anywhere.
- This byte code is platform-independent and can be run on any machine. Java is a secure programming language because it has no explicit pointer and programs runs in the virtual machine.
- The Java Virtual Machine (JVM) and Java Development Kit (JDK) come with a suite of tools that are essential for developing, debugging, and monitoring Java applications. These tools are vital for a range of tasks from compiling and running Java applications to monitoring and debugging them. The JDK tools are particularly useful for developers, while JVM tools are more oriented towards performance monitoring and tuning.
- Control statements in Java direct the program's flow and include decision-making, looping, and branching statements.
- Command line arguments in Java allow users to pass information to a Java program at runtime via the command line. These arguments are stored in the args array passed to the main method.

Notes

Notes

- Inheritance in Java is a fundamental object-oriented programming concept where a new class (subclass) derives properties and behaviors (fields and methods) from an existing class (superclass). This mechanism promotes code reusability and establishes a natural hierarchical relationship between classes. Inheritance allows subclasses to inherit fields and methods from the superclass, and also to override or extend them to provide specific functionality.
- Java supports single inheritance, meaning a class can inherit from only one superclass, but it can implement multiple interfaces to achieve multiple inheritance-like behavior. This ensures a clear and maintainable class structure while enabling polymorphism and dynamic method dispatch.

Glossary

- JVM: Java Virtual Machine
- JRE: Java Runtime Environment
- JDK: Java Development Kit
- JAR: Java Archive
- JMC: Java Mission Control
- Arrays: A collection of similar type of elements when put in contiguous memory location is known as array. In Java, array is an object which contains elements of a similar data type. Array is a kind of data structure where we store similar elements.

Check Your Understanding

1. Why was the programming language initially named “Oak” changed to “Java”?
   a) Because the team wanted a more dynamic and cool name
   b) Because Oak was already trademarked by Oak Technologies
   c) Because the programming language was first created on the Indonesian island of Java
   d) Because the team members preferred Java over other names like Silk
2. Which of the following statements is true about Java bytecode and its compilation process?
   a) Java bytecode files have a .java extension and are compiled using the java command.
   b) Java bytecode is platform-dependent and needs to be recompiled for each platform.
   c) Java bytecode files have a .class extension and are compiled using the javac command.
   d) Java bytecode files are named differently from the class name in the Java source code.
3. Which of the following components is included in the Java Development Kit (JDK) but not directly part of the Java Runtime Environment (JRE)?
   a) JVM
   b) javac
   c) Libraries
   d) Runtime Environment

Notes

4. What is the purpose of the Java HotSpot VM?
   a) To provide a lightweight JVM for mobile devices
   b) To optimize the performance of Java applications at runtime
   c) To support the development of Java applets
   d) To facilitate the integration of Java with legacy systems
5. What will be the output of the following Java code?

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.print(i + " ");
        }
    }
}
```

   a) 0 1 2 3 4 5
   b) 1 2 3 4 5
   c) 0 1 2 3 4
   d) 1 2 3 4

Exercise

1. What is byte code?
2. Define JDK, JRE, JVM.
3. Define data types available in Java.
4. Differentiate between Bitwise AND operator and Logical And Operator.
5. How to find duplicate elements in an array?

Learning Activities

1. Write a program to check whether a number is prime number or not.
2. Write a Java program to calculate the factorial of a given number using a for loop.

Check Your Understanding- Answers

1. b) 2. c) 3. b) 4. b)
5. c)

Further Readings and Bibliography

1. R. Nageswara Rao. (2016). Core Java: An Integrated Approach. Dreamtech Press, 720 pages.
2. E. Balagurusamy. (2023). Programming with Java. 7th Edition. McGrawHill Publications, 592 pages.
3. Barry A. Burd. (2017). Beginning Programming with Java for Dummies. 5th Edition. Wiley Publications.
4. Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action. Dreamtech Press.

Notes

Module - II: Java with Object Orientated Features

Learning Objectives:

At the end of this module, you will be able to:

- Differentiate class and object
- Define Encapsulation
- Discuss types of polymorphism
- Define various types of Inheritance
- Define constructor and its types
- Differentiate between method overloading and method overriding
- Define final method, final variable, final class
- Explain how garbage collection runs in Java

Introduction

Object-Oriented Programming (OOP) define a programming concept based on objects and classes. OOP allows us to establish software as a collection of objects that consist of state and behaviour of object.

2.1 Introduction to Object Oriented Programming

OOP allows decomposition of a problem into a number of modules. The software is apportioned into a number of small units known as objects. Functions associated with that object helps to access the data. The functions of one object can access the functions of an alternative object. OOP does not permit data to flow spontaneously around the system rather ties data more closely to the function that operate on it, and safeguards it from other function.

Some of the features of Object Oriented programming are:

- ❖ OOP gives importance on data rather than procedure.
- ❖ Programs are divided into objects.
- ❖ Functions that operate on the data of an object are ties together in the data structure.
- ❖ Through function or method, objects can communicate with each other.
- ❖ New members (data and functions) can be easily be added.
- ❖ OOP follows bottom up approach.

Basic Concepts of Object Oriented Programming includes:

- ❖ Objects
- ❖ Classes
- ❖ Data abstraction and encapsulation
- ❖ Inheritance
- ❖ Polymorphism
- ❖ Dynamic binding
- ❖ Message passing

2.1.1 Overview of OOPs: Part 1

Notes

Classes and Objects:

An object is a software bundle of related state and behaviour. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behaviour are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Class:

A class is a design or pattern from which objects are created. In the real world, many individual objects of the same kind can be found. There are many kind of cars exists and each one is built from the same set of blueprints (i.e. four wheels, gear, break, engine etc) and contains the same components. In object-oriented terms, we say that Swift is an instance of the class Car. A class is the outline from which individual objects are created. In programming standpoint, we can pronounce class is a group of objects of having common properties. Class is logical entity whereas object is physical one.

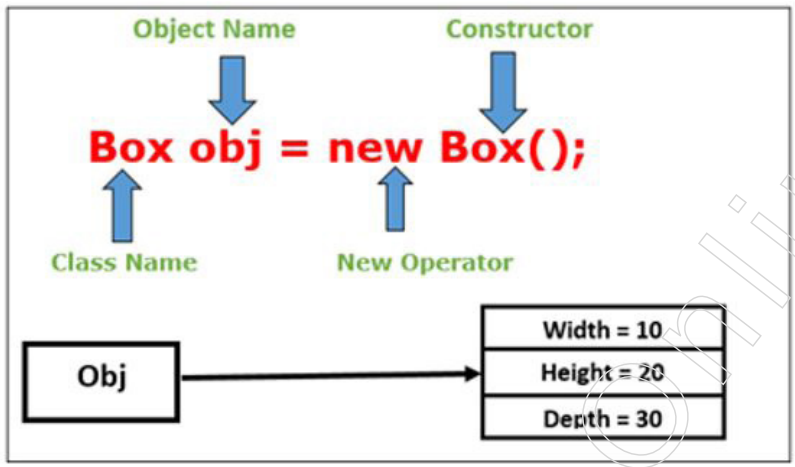
Example: Car.java

```
class Car {
int wheel = 4;
int speed = 0;
int gear = 5;
void changeGear(int value) {
gear = value;
}
void speedUp(int incr) {
speed = speed + incr;
}
void applyBrakes(int decr) {
speed = speed - decr;
}
void printState() {
System.out.println(“Speed is: “ + speed + “ gear is in:” + gear);
}
}
```

Objects:

Objects are the run time entities in object oriented system. Objects are real life entity like person, table, pen, pencil, car etc. Objects take up space in the memory. When a program is executed, the objects interact with each other by sending messages to one another. For example, if “customer” and “account” are two objects present in a program, then the customer object can send message to the account object requesting for the bank balance. Each object contains data and code to manipulate data. We already know that object has state that represents data and behaviour that represents functionality. Pencil is an example of object. Pencil has name (eg. Doms) and colour (eg. yellow) so these are the state of pencil. The functionality of Pencil is writing, drawing, sketching etc. So these are behaviours of Object.

Notes



The above picture is depicting how to create object in Java programming language.

Example: Box.java

```
class Box{
int width = 10;
int height = 20;
int depth = 30;
void display(){
System.out.println("Width is: " + width + "Height is: "+ height + "Depth is: " +depth);
}
public static void main(String args[]){
Box obj = new Box();
obj.display();
}
}
```

In above example we have used "new" keyword which is used to allocate memory.

2.1.2 Overview of OOPs: Part 2

Encapsulation

One of the most important object oriented concept is Encapsulation. As we know class is a base of any object oriented programming , so writing program in Java means to have a set of classes. All created classes can be used to create objects which basically can be achieved with the concept of object oriented principle called encapsulation, and the different elements that is possible in a class. As discussed , that class is a basic building blocks in any Java program that mean using it is feasible to write the entire Java program. So, a class in fact, is a template, basically gives a framework of an object. So, in order to describe an object many elements for that object are required.

The elements which are very important in any objects are called fields, the methods, constructor and sometimes special items called blocks and also class within a class called the nested class. One More important concept in an object is called the interface. So, mainly there are 6 different items to be incorporated in a class. Now fields and methods little

Notes

bit we have familiarity in our last few programs, but other 3 things like constructors and then interface and everything is not known to us at the moment. So, we shall on all this thing slowly. We will basically discuss or emphasized on field methods and constructors these 3 things.

So, any class can be defined with its own unique name. So, it is called the class name usually it is given by the user, so user defined name. So here so this is the name of the class that you have you are going to build, so these are name. And then it consists of 2 things which is mentioned here as you see a set of it is called the fields or member elements. All are member elements method also a member elements the particularly it is called fields and other is called the methods. So, fields and methods are 2 very important things in any class.

A class can consist of 1 or 0 or more methods and 0 or more fields. There is quite possible that a class does not have any fields also possible that a class does not have any methods, but which is really not useful is a useless thing without any fields are any methods anyway. So, logically a class have 0 or more member elements fields and methods. So, fields is basically simplest of his form is basically reference variables or some primitive variables that means objects and other name of the other variables that can be define and then methods are basically the operations which are possible to manipulate all the fields variables are there.

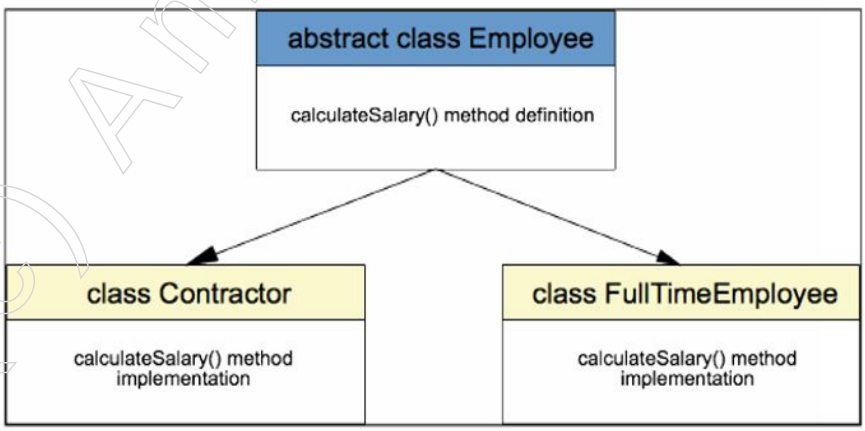
So, these are basically called data in a class and these are basically called operations in a class. So, these are the 2 things data and operations when they are punched together it is call encapsulation which you have a little bit learn in our previous lectures. So, here encapsulation means data and operation are to be put together into a single unit called class.

2.1.3 Overview of OOPs: Part 3

Abstraction:

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces.

An example of abstraction : create one superclass called Employee and two subclasses –Contractor and FullTimeEmployee. Both subclasses have common properties to share, like the name of the employee and the quantity of money the person will be paid per hour. There is one major divergence between contractors and full-time employees – the time they work for the company. Full-time employees work constantly 8 hours per day and the working time of contractors may vary.



Notes

Which should you use, abstract classes or interfaces?

Consider using abstract classes if any of these statements apply to the situation:

- Share code among several closely related classes.
- Anticipate that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- Affirm non-static or non-final fields. This empowers to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to the situation:

- Presume that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- Specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
- Take advantage of multiple inheritance of type.

Polymorphism:

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Polymorphism in Java is a concept by which we can perform a single action in different ways.

Polymorphism is derived from 2 Greek words: poly and morphs. The word “poly” means many and “morphs” means forms. So polymorphism means many forms. There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Runtime Polymorphism

Method Overriding is known as runtime polymorphism. During runtime it is decided which method has to call that’s why it is called runtime polymorphism.

Example:

1.

Animal.java

```
public class Animal{
public void sound(){
System.out.println(“Animal is making a sound”);
}
}
```

Horse.java

Notes

```
class Horse extends Animal{
public void sound(){ // Override
System.out.println("Neigh");
}
public static void main(String args[]){
Animal obj = new Horse();
obj.sound();
}
}
```

Output:

Neigh

2.

Cat.java

```
public class Cat extends Animal{
public void sound(){ // Override
System.out.println("Meow");
}
public static void main(String args[]){
Animal obj = new Cat();
obj.sound();
}
}
```

Output:

Meow

Compile time Polymorphism Example

Method Overloading on the other hand is known as compile time polymorphism. During compile time it is decided which method has to call that's why it is called compiler time polymorphism.

Example.

```
class Overload
{
void demo (int a)
{
System.out.println   ("a: " +  a);
}
void demo (int a, int b)
```

Notes

```
{
System.out.println (“a and   b: “ + a + “,” + b);
}
double demo(double a) {
System.out.println(“double a: “ + a);
return a*a;
}
}
class MethodOverloading
{
public static void main (String args [])
{
Overload Obj = new Overload();
double result;
Obj .demo(10);
Obj .demo(10, 20);
result = Obj .demo(5.5);
System.out.println(“O/P       : “ + result);
}
}
```

2.2 Other Important Aspects in Java

Access Specifier: Access type of method is described by access specifier or access modifier. It specifies the visibility of the method.

2.2.1 Access Specifier: Private, Public, Protected, and Default

Four types of access specifier:

- Public: When we use public specifier in method, it is accessible by all classes.
- Private: When we use a private access specifier, the method is accessible only in that class in which it is declared.
- Protected: When we use protected access specifier, the method is accessible within the same package or by subclasses in different package.
- Default: When we do not write any of the above three access specifier in the method declaration, Java uses default access specifier. It is visible only from the same package.

Return Type: Every method returns some value which is known as Return type. It may return a primitive data type, object etc. If the method does not return anything, we use void keyword.

Method Name: In your Java program you have to provide a unique name which will define the name of method. Name of method must be well connected to the functionality of the method. Suppose, you are creating a method for addition of two numbers, the method name can be add() or sum() or addition(). A method is invoked by its name.

Notes

Parameter List: It means list of parameters separated by a comma and enclosed in the pair of parentheses like void myMethod (int a, int b). Here, a and b are parameters which are of int type. It contains the data type (int here) and variable name(a and b here). If the method has no parameter, left the parentheses blank like void myMethod().

Method Body: It is a part of the method declaration which contains all the tasks to be performed. It is enclosed within the pair of curly braces.

Naming a Method: While defining a method we need to keep it in mind that the method name must be a verb and as per convention it should start with a lowercase letter. In the multi-word method name, the first letter of each word must be in uppercase except the first word.

Example:

- Single-word method name: sum(), add(), area() etc.
- Multi-word method name: areaOfTriangle(), perimeterOfCircle, stringComparision() etc.

Types of Method

There are two types of methods in Java. Predefined Method and User-defined Method.

Predefined Method

These are also known as the standard library method or built-in method. In Java, predefined methods are those methods which are already defined in the Java libraries. You can directly use these built-in methods by calling them in the program. Some pre-defined methods are equals(), length(), sqrt(), compareTo() etc. When we call any predefined methods in our program, a block of codes related to the particular method runs in the background. These built-in methods are also defined inside class. Like, print() method is defined in the java.io.PrintStream class. This method prints the statement that we write inside the method.

Example: Test.java

```
class Test
{
public static void main(String[] args)
{
System.out.print("The  output is: " + Math.sqrt(9));
}
}
```

Output:

The output is: 3.0

In this example, three built-in methods namely main(), print(), and sqrt() have been used. You can see these methods have been used directly without declaration because they are predefined. The sqrt() method is a method of the Math class which returns the square root of a number

User-defined Method

These types of methods are written by the user or programmer. You can modify these methods according to the requirement.

Notes

Example:

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
class Addition{
public void add(int a, int b) {
int c = a+b;
System.out.println(" The addition of two number is :" + c);
}
}
```

Calling a User-defined Method:

We create method to perform some specific task, for that that method should be called. The program control will be transferred to the called method when you will call or invoke a user-defined method,

```
public class AdditionTest {
public static void main (String args[])
{
Addition obj = new Addition();
obj.add(5,7);
}
}
```

Output:

The addition of two number is: 12

We have defined the above method named add(). It has a parameter a and b of type int. The method does not return any value that's why we have used void.

We have discussed that object has state and behaviour. The behaviour of object is represented by method in Java. The method is collection of instructions or you can say block of code that can do a specific task. We use method to achieve reusability of code. We write a method once and can use it many times. We do not require to write code again and again. The most important method in Java is the main() method. Execution of Java program starts from main() method. We write public static void main(String[] args)

Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

2.2.2 Access Specifiers: Final and Abstract

A class that is declared using "abstract" keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.

Let us say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class

must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

Notes

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Final

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Class, method and variable can be final.

Java Final Method:

If we use final keyword with variables then we cannot change the value of that variable. A final variable which is not initialised that means which does not have any value then it is called blank final variable. It can be initialized in the constructor only. The blank final variable can be static also which can be initialized in the static block only.

Example of final variable: Car.Java

There is a final variable speed, we are going to change the value of this variable, but It can’t be changed because final variable once assigned a value can never be changed.

```
class Car{
final int speed=120;//final variable
void run(){
speed=100; //cannot change as final variable will act as constant
}
public static void main(String args[]){
Car obj=new Car();
obj.run();
}
}
```

In the above program you will get compile time error as you want to change the value of final variable.

Example of blank final variable: Car1.java

```
class Car1{
final int speed;//blank final variable
Car(){
speedlimit=70;
System.out.println(speedlimit);
}
public static void main(String args[]){
Car obj = new Car();
}
```

Notes

```
}
```

Output:

70

Java final Method:

We can make any method as final, but whenever we are making one method as final then we will not be able to override it. Method which is final can be inherited but cannot be overridden. Remember that we cannot make a constructor final as constructor is never inherited.

Example of final method: Car3.java

```
class Car3{
final void run(){
System.out.println("I am inside parent class");}
}
class Swift extends Car{
void run(){ // we are trying to override the final method
System.out.println("I am inside child class");
 }
public static void main(String args[]){
Swift obj= new Swift();
obj.run();
}
}
```

In the above program you will get compile time error again as you want to override a final method which is not possible.

Java Final Class:

We can make class as final. Once we make a class final, we cannot extend it.

Example of final class: Car4.java

```
final class Car{
 }
class ZenEstilo extends Car{ //We are trying to inherit from a final class
 void run(){
System.out.println("I am inside child class");
 }
public static void main(String args[]){
ZenEstilo obj= new ZenEstilo();
obj.run();
}
```

}

In the above program you will get compile time error because you want to inherit from a final class which is not possible.

Notes

2.2.3 Types of Inheritance

Inheritance:

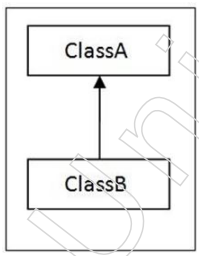
Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

We are now going to learn a very important feature of object oriented paradigm which is called the Inheritance. Inheritance is very common biological term and you know inheritance means basically taking properties from ancestor. Previously we discussed about this concept where we came to know about Single Inheritance, Multilevel Inheritance, Hierarchical Inheritance, Multiple Inheritance and Hybrid Inheritance.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.

Single inheritance: If a child is inheriting properties from a parent then it is known as single inheritance. Here, in Java language single inheritance implies that when a class inherits properties from another class then it is called as Single Inheritance.



Example: B.java

```
class A{
void display(){
System.out.println("I am inside parent class.");
}
}
public class B extends A{
void myMethod(){
System.out.println("I am inside child class.");
}
public static void main(String args[]){
B obj = new B();
obj.myMethod();
obj.display();
```

Notes

```
}
}
```

Output:

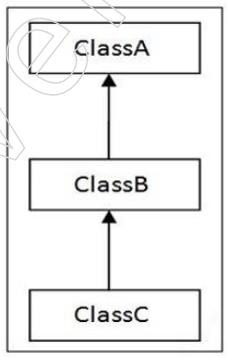
I am inside child class.

I am inside parent class.

From the above example, you can easily understand that A class in parent class and B is child class and as B class is inheriting properties of A class that is why B class can access A class's data and method. In this example, we have created B class's object (obj) and we are accessing display() method by B class's object. Hence, we can write obj. display().

Multilevel Inheritance:

Multilevel inheritance refers to a technique in Object Oriented technology where one class can inherit from a derived class, thereby making that derived class the base class for the new class. An easy real life example would be children inherits from the parent, grandchildren inherits from the children. Multilevel inheritance refers to a technique in Object Oriented technology where one class can inherit from a derived class, thereby making that derived class the base class for the new class. An easy real life example would be children inherits from the parent, grandchildren inherits from the children.



From above fig you can see in that C is subclass or child class of B and B is a child class of A. For more details and

Example: C.java

```
class A{
void draw(){
System.out.println("I am inside parent class A.");
}
}
class B extends A{
void display(){
System.out.println("I am inside class B.");
}
}
public class C extends B{
void myMethod(){
```

Notes

```
System.out.println("I am inside child class C.");
}
public static void main(String args[]){
B obj = new B();
C obj1 = new C();
obj.draw();
obj.display();
obj1.myMethod();
obj1.display();
obj1.draw();
}
}
```

Output:

I am inside parent class A.

I am inside class B.

am inside child class C.

I am inside class B.

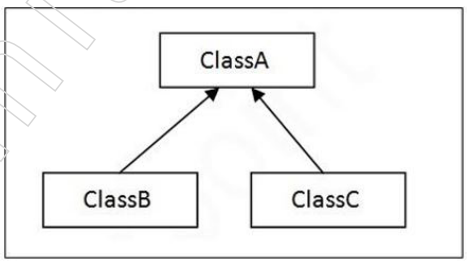
I am inside parent class A.

From the above example, you can easily understand that A class in parent class of B and C. Class B is parent class of class C. So, class B is inheriting properties of class A and class C is inheriting the properties of class B. Hence, class C is actually inheriting properties of class B as well as class A.

In this example, we have created B class's object (obj) and C class's object (obj1) and we are accessing draw() and display() method by B class's object and by C class's object, we can access draw(), display and myMethod().

Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes. Suppose, class A represents Animal class and class B represents Dos and class C represents Cat class. So, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.



Example: HiechicalInheritanceExample.java

```
class Animal{
void run(){
System.out.println("Animals are running.");
}
```

Notes

```
}
class Dog extends Animal{
void bark(){
System.out.println("Dog barks.");
}
}
class Cat extends Animal{
void sound(){
System.out.println("Cat sounds like meow.");
}
}
public class HiechicalInheritanceExample{
public static void main(String args[]){
Cat obj=new Cat();
obj.eat();
obj.sound();
}
}
```

Output:

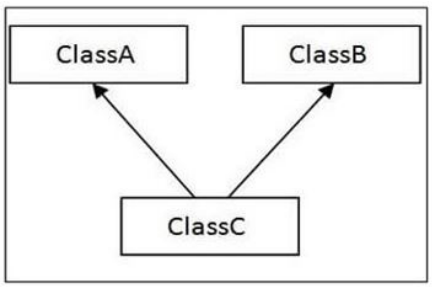
Animals are running.

Cat sounds like meow.

From the above example, you can easily understand that Animal class in parent class of Dog class and Cat class. So, class Dog is inheriting properties from class Animal and class Cat is also inheriting from class Animal. In this example, we have created Cat class's object (obj) with that object we accessed Animal class's run() method.

Multiple Inheritance

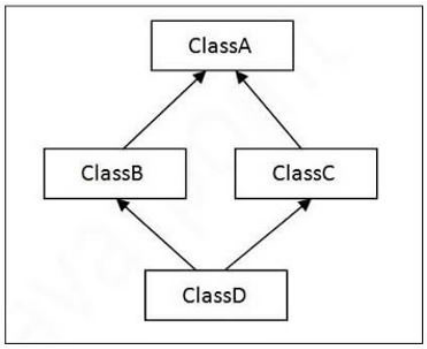
In Multiple Inheritance, a class can inherit the properties from more than one parent class. We can say multiple inheritance means a class extending more than one class. In Java programming language multiple inheritance is not supported in class level rather it can be achieved indirectly through interfaces which we will learn later. In Multiple Inheritance, a class can inherit the properties from more than one parent class. We can say multiple inheritance means a class extending more than one class. In Java programming language multiple inheritance is not supported in class level rather it can be achieved indirectly through interfaces which we will learn later.



Notes

Hybrid Inheritance

Hybrid inheritance is mixture of Single and Multiple inheritance and as Multiple inheritance is not supported in class level in Java hence, Hybrid inheritance is also not supported in class level and by using interfaces you can have hybrid inheritance in Java.



In the above diagram you can see Class A is the Parent for both class B and class C which is Single Inheritance and both class B and class C are Parent for class C which is Multiple Inheritance and is not supported in class level in Java.

Example:

```
class A
{
public void show()
{
System.out.println("I am in class A");
}
}
class B extends A
{
public void show()
{
System.out.println("I am in class B ");
}
}
public class C extends A
{
public void show()
{
System.out.println("I am in class C");
}
}
public class D extends B,C
{
public void display()
{
```

Notes

```
System.out.println("This method is for displaying.");
}
public static void main(String args[])
{
Notes
D obj = new D();
obj.displayD();
obj.show();
}
}
```

Output :

Error..

In the above example you can see when we are calling show() method by class D's object then confusion happens, compiler does not understand which shoe() method has to call. This the reason why multiple inheritance is not supported in class level in Java. Hence, Hybrid Inheritance is also not supported in class level.

2.2.4 Method Overloading

When more than one method having same name but different parameter list reside in a same class, it is known as method overloading. The advantage of method overloading is it increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs.

Different Ways to Overload the Method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Example of by changing number of arguments:

OverloadingExample.java

```
class OverloadingExample{
void add(int a, int b){
System.out.println("Result is: " + (a+b));
}
void add(int a, int b, int c)
{
System.out.println("Result is: " + (a+b+c));
}
public static void main(String[] args){
```

Notes

```
estOverloading obj = new TestOverloading();
obj.add(4,7,8);
obj.add(5,7);
}
```

Output:

Result is: 19

Result is: 12

In the above example, we have created two method of having same name (add()) but in first method we have passed two parameters and in case of second we have passed three parameters and both the methods are in same class. Compiler understands by counting the number of argument that which method is being called.

Example of By changing data type of arguments: OverloadingExample1.java

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class OverloadingExample1{
void add(int a, int b){
System.out.println("Result is:          " + (a+b));
}
void add(double p, double q){
System.out.println("Result is:          " + (p+q));
}
public static void main(String[] args){
TestOverloading obj = new TestOverloading();
obj.add(4,8);
obj.add(1.2, 4.5);
}
}
```

Output:

12

5.7

In the above example, we have created two method of having same name (add()) but in first method we have passed two integer type parameters and in case of second we have passed two double type parameters and both the methods are in same class. Compiler understands by data type which method is being called.

2.2.5 Method Over-riding

Method overriding is a process of overriding parent class method by child class method with more specific way. Method overriding performs only if two classes have parent-child relationship (IS-A relationship) which means classes must have inheritance relationship. If

Notes

a child class has a method same as declared in the parent class then method overriding takes place in Java. Java programming language does not allow method overriding if child class has more restricted access modifier than parent class. Access modifier will be discussed later.

Rules of method overriding

- In method overriding, name of method of both classes must have same and equal number of parameters.
- Method overriding is referred to as runtime polymorphism because during runtime which method is being called is decided by JVM.
- Access modifier of child class's method must not be more restrictive than parent class's method.
- Private, final and static methods cannot be overridden.
- There must be an IS-A relationship between classes (inheritance).

Example: SamsungNote10.java

Below we have simple code example with one parent class and one child class wherein the child class will override the method provided by the parent class.

```
class Samsung {
void camera(){
System.out.println(" Camera has basic features"); }
}
public class SamsungNote10 extends Samsung{
void camera(){
System.out.println("Camera with advanced feature"); }
public static void main(String args[]){
SamsungNote10 obj = new SamsungNote10();
obj.camera();
}
}
```

In the above example you can see that both Samsung class (parent class) and SamsungNote10 class (child class) has the same method name camera() with same number of parameter. Here, you can see, SamsungNote10 class gives its own implementation of camera() method.

2.2.6 Garbage Collection

When an object is unused then by the process of garbage collection Java language releases fallow memory occupied by unused objects. This process is inevitably done by the JVM and this garbage collection is essential for memory management. When Java programs run, objects are generated and memory is allocated to the program. When there is no reference to an object is there that object is assumed to be no longer needed and the memory occupied by the object are released.

This technique is called Garbage Collection. Java gc() method is applied to call garbage collector unequivocally. However gc() method does not guarantee that JVM will

perform the garbage collection. It only request the JVM for garbage collection. This method is present in System and Runtime class.

Notes

Advantages of Garbage Collection

1. It is done automatically by JVM.
2. Programmers need not to worry about dereferencing of an object.
3. It increases memory efficiency.

Work of Garbage Collection:

The garbage collection process is a part of the JVM and done by JVM itself. We, as a developer, do not need to explicitly mark objects to be deleted but we can request to JVM for garbage collection of object. Garbage Collection cannot be done explicitly in Java. We can request JVM for garbage collection by calling System.gc() method. But our request does not guarantee that JVM will perform the garbage collection.

An object is able to get garbage collect if it is non-reference. We can make an object non-reference by using three ways.

Set Null to object Reference which Makes it Able for Garbage Collection

For example:

A obj = new A();

obj=null;

By Setting New Reference Which Makes it Able for Garbage Collection

For example

A obj1 = new A();

A obj2 = new A();

obj2 = obj1;

By Making Anonymous Which Makes it Able for Garbage Collection.

Finalize() Method

In some cases, there may be some specific task to be performed before it is destroyed such as releasing any resources or closing an connection etc. We use finalize() to handle such situation.

Request for Garbage Collection

We can request to JVM for garbage collection however, it is upto the JVM when to start the garbage collector.

Example: GCTest.java

```
class GCTest{
public static void main(String[] args){
GCTest obj = new GCTest();
obj = null;
System.gc();
}
```

Notes

```
public void finalize(){
System.out.println("Garbage collected");
}
}
```

Output:

Garbage Collected

2.2.7 Constructors and Types in Java

Constructor is a special kind of method. It is called to create an instance of class and memory for the object is allocated. Whenever an object is created using the new keyword, a constructor must be called. Constructor can use any access specifier. You can write a constructor private but there scope of that constructor would be within the class only.

Types of Constructors:

There are two types of constructors in Java:

1. Default constructor or no-argument constructor: If programmer does not write constructor then Java compiler provides a default constructor. The default constructor is used to provide the default values to the object like 0, null etc.
2. Parameterized constructor: When a constructor is accepting a specific number of parameters then it is called a parameterized constructor. This type of constructor is used to provide values to objects.

Function of constructor:

1. To instantiate object.
2. To initialize the instance variable.

Rules for creating Java constructor

Few rules are there which a programmer must follow.

1. Constructor name must be the same as its class name.
2. Constructor must not have any explicit return type.
3. Constructor cannot be abstract, static, final, and synchronized.

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.

Example: Car.java

```
class Car{
String name;
Car(String n){ // parameterized constructor
name = n;
}
void display(){
```

Notes

```
System.out.println("Car's name        is "+name);
}
public static void main(String args[]){
Car obj1 = new Car("Zen Estilo");
Car obj2 = new Car("Swift VDI");
obj1.display();
obj2.display();
}
}
```

Output:

Zen Estilo

Swift VDI

2.2.8 Role of Constructors in Inheritance

Constructor is special kind of method that allows us to create an object of class and has same name as class name with no explicit return type. Whenever a child class inherits from parent class, the child class inherits state and behaviour in the form of variables and methods from its parent class but it cannot inherit constructor of parent class or super class because constructors are special kind of method and have same name as class name. Hence, if constructors are inherited in child class then child class would contain a parent class constructor which is not possible as constructor must have same name as class name.

If we define constructor of parent class inside constructor of child class it will give compile time error. If constructors could be inherited then it would be impossible to achieve encapsulation as by using a parent class's constructor we can access private members of a class.

Summary

- Object-oriented programming (OOP) in Java is a programming paradigm that emphasizes the concept of objects, which encapsulate data and behavior. In Java, classes are used to define objects, and they serve as blueprints for creating instances. Key principles of OOP in Java include encapsulation, inheritance, polymorphism, and abstraction. Encapsulation hides the internal state of objects and exposes only the necessary methods to interact with them, ensuring data integrity and security.
- Inheritance allows classes to inherit properties and behaviors from other classes, promoting code reuse and hierarchy. Polymorphism enables objects to take on multiple forms, allowing methods to be overridden in subclasses. Abstraction simplifies complex systems by providing a high-level view while hiding implementation details. Java's strong support for OOP facilitates modular, scalable, and maintainable software development.
- Encapsulation means putting together data and method into a single unit called class. Abstraction features is a concept or idea not associated with any specific instance and does not have a concrete existence.
- Abstract classes in Java are classes that cannot be instantiated directly and are typically used as base classes for other classes to inherit from. They may contain

Notes

abstract methods, which are methods without a body, and concrete methods, which have implementations. Abstract classes provide a way to define common behavior and properties that subclasses can inherit, while allowing for specific implementation details to be defined in the subclasses.

- Constructors play a crucial role in inheritance by initializing the state of objects and ensuring that the superclass and subclass objects are properly initialized. In Java, when a subclass is instantiated, the constructor of its superclass is implicitly called to initialize inherited members before the subclass's constructor body executes. If the superclass constructor requires arguments, the subclass constructor must explicitly invoke it using the super() keyword, passing the required arguments.
- Constructors allow subclasses to inherit and initialize the state of their superclass, ensuring the integrity of the inheritance hierarchy and facilitating the proper functioning of polymorphism and method overriding. They help maintain the encapsulation of data and behavior by ensuring that the superclass's internal state is initialized correctly within the context of subclass instances.
- Method overloading in Java refers to the ability to define multiple methods within the same class or in a subclass with the same name but with different parameter lists. The idea is to have multiple methods with the same name that perform different tasks based on the type or number of parameters they accept.
- Method overriding in Java occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. In other words, the subclass provides its own version of a method that is already present in its parent class. Method overriding is a key feature of object-oriented programming and allows for the implementation of polymorphism, where the same method name is used in both the superclass and subclass.

Glossary

- OOP: Object-Oriented Programming
- JVM: Java Virtual Machine
- Class: A class is a design or pattern from which objects are created. In the real world, many individual objects of the same kind can be found.
- Garbage collection is an automatic memory management process in Java that reclaims memory occupied by objects that are no longer in use, thereby preventing memory leaks and improving performance.
- Objects are the run time entities in object oriented system. Objects are real life entity like person, table, pen, pencil, car etc. Objects take up space in the memory.
- Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts and refers to the bundling of data and methods that operate on the data into a single unit, known as a class.
- Abstraction is the process of simplifying complex systems by modeling classes based on the essential features and ignoring unnecessary details. It involves creating abstract classes or interfaces that define the common characteristics and behaviours of a set of related objects.
- Polymorphism is the ability of a single entity, such as a method or an object, to take on multiple forms. In Java, polymorphism is achieved through method overloading and method overriding. Polymorphism allows a single interface to be used for different types of objects, providing flexibility and extensibility in the code.

Notes

Check Your Understanding

1. What is the main principle of encapsulation in OOP?
   a) Inheritance
   b) Polymorphism
   c) Abstraction
   d) Data hiding
2. Which method in Java can be used to explicitly request the garbage collector to perform garbage collection?
   a) clear()
   b) clean()
   c) gc()
   d) collect()
3. Why can't constructors be inherited in Java?
   a) Because constructors are not allowed to have parameters
   b) Because constructors have the same name as the class and cannot be inherited
   c) Because constructors are private by default
   d) Because constructors cannot access private members of a class
4. What are the two ways to overload a method in Java?
   a) By changing the return type and the number of parameters
   b) By changing the method name and the number of parameters
   c) By changing the method name and the return type
   d) By changing the number of parameters and the data type of parameters
5. In Java, when does method overriding occur?
   a) When two classes have a parent-child relationship
   b) When two classes have the same method name
   c) When a child class has a method with the same signature as the method in the parent class
   d) When a child class has a method with a more restricted access modifier than the method in the parent class

Exercise

1. Define method overloading and explain how it allows multiple methods with the same name but different parameter lists. Provide examples of method overloading in Java and discuss its advantages in simplifying code and enhancing readability.
2. Explain the significance of method visibility modifiers and provide examples of public, private, protected, and default access modifiers.
3. Describe abstraction and its role in software development. Provide examples of abstract classes and interfaces in Java and explain how they promote code reuse and flexibility.
4. Discuss the key principles and advantages of Object-Oriented Programming (OOP) over procedural programming paradigms. Provide examples to illustrate the concepts.
5. Explain the concepts of classes and objects in Java. Provide an example of a class with attributes and methods, and demonstrate how objects are instantiated from classes.

Notes

Learning Activities

1. Discuss method overriding and its role in achieving runtime polymorphism. Provide examples of method overriding in Java and explain how it allows subclasses to provide specific implementations of inherited methods.
2. Given a scenario, such as designing a banking system or a school management system, explain how you would apply OOP principles like encapsulation, abstraction, inheritance, and polymorphism to design the system's classes and their relationships.

Check Your Understanding- Answers

1. d) 2. c) 3. b) 4. d)
5. c)

Further Readings and Bibliography

1. R. Nageswara Rao. (2016). Core Java: An Integrated Approach. Dreamtech Press, 720 pages.
2. E. Balagurusamy. (2023). Programming with Java. 7th Edition. McGrawHill Publications, 592 pages.
3. Barry A. Burd. (2017). Beginning Programming with Java for Dummies. 5th Edition. Wiley Publications.
4. Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action. Dreamtech Press.

Module - III: Exception Handling Interface and Thread in Java

Notes

Learning Objectives:

At the end of this module, you will be able to:

- Define exceptions and its types
- Differentiate between Exception and Error
- Know handling exceptions with try, catch, throw, throws, finally
- Discuss about interfaces and how to implement multiple inheritance through interface
- Discuss about uncaught exception
- Define threads and thread life cycle

Introduction

A key component of Java programming that enables developers to regulate the handling of errors and unforeseen circumstances is exception handling.

3.1 Overview of Exception Handling

Java uses try, catch, throw, throws and finally blocks to provide a strong method for managing exceptions.

3.1.1 Exception Handling: Part 1

A problem that arises during the execution of a program is known as exception. In this situation, the normal flow of the program is disrupted and the program terminates abnormally. An exception is an event that occurs during the execution of a program which disturb the normal flow of instructions. Creating an exception object and handing it at the runtime is called throwing an exception.

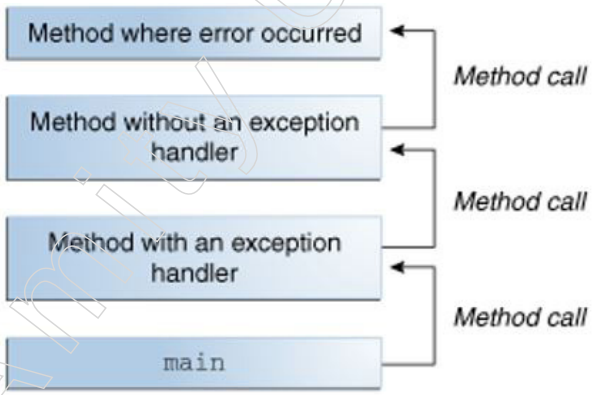


Figure: Method call

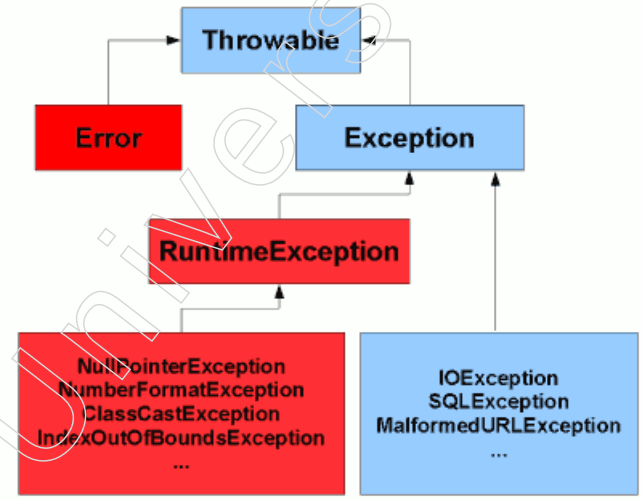
```
int a=50/0;                          //ArithmeticException
String s=null;
int a=50/0;                          //ArithmeticException
String s=null;
```

Notes

```
System.out.println(s.length());          //NullPointerException
String s="abc";
System.out.println(s.length());          //NullPointerException
String s="abc";
int i=Integer.parseInt(s);               //NumberFormatException
int i=Integer.parseInt(s);               //NumberFormatException
 int num = Integer.parseInt("abhay");    //NumberFormatException
int a[]=new int[5];
 int num = Integer.parseInt("abhay");    /NumberFormatException
a[10]=50;                                //ArrayIndexOutOfBoundsException
```

Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error (which are marked in Blue colour in Figure below) are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.



Source: https://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Unchecked Exception

The classes which inherit Runtime Exception are known as unchecked exceptions e.g. NullPointer Exception, Arithmetic Exception, Array Index Out Of Bounds Exception etc are unchecked exception.

An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions.

Checked Exception	Unchecked Exception
Checked exceptions occur at compile time	Unchecked exceptions occur at runtime.
The compiler checks checked exception.	The compiler does not check these types of exceptions.
JVM needs the exception to catch and handle.	JVM does not catch and handle this type of exception.

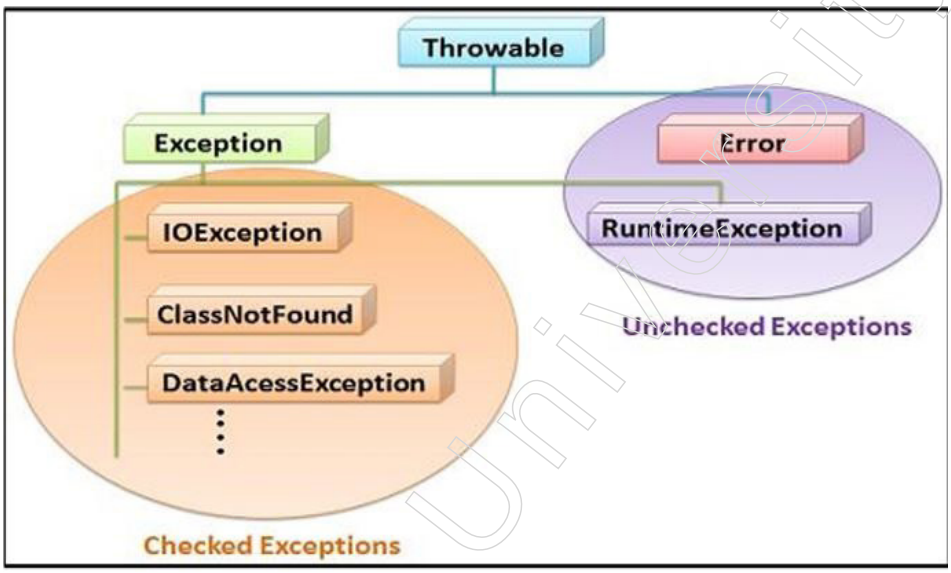
Notes

Examples of Checked exceptions: IOException, SQLException, FileNotFoundException etc	Examples of Unchecked Exceptions: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

3.1.2 Exception Handling: Part 2

Error class is child class of the built-in class “Throwable”. Due to the lack of the system resources errors occur and cannot be handled by the programmer. Errors cannot be created, thrown or caught, hence errors cannot be recovered by any means because they. Errors are caused due to the catastrophic failure which usually cannot be handled by your program.

Errors are always of unchecked type, as compiler do not have any knowledge about its occurrence. Errors always occur in the run-time environment. The error can be explained with the help of an example, the program has an error of stack overflow, out of memory error, or a system crash error, this kind of error are due to the system.



Source: https://techdifferences.com/difference-between-checked-and-unchecked-exception-2.html

Difference Between Exception and Error:

Exception	Error
Exception is classified as checked and unchecked.	Error is classified as an unchecked type.
It belongs to java.lang.Exception class.	It belongs to java.lang.Errors class.
The use of try-catch blocks can handle exceptions and recover the application from them.	It is not possible to recover from an error.
It can occur at run time compile time both.	It can't be occur at compile time.
Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time.	Errors happen at run time. They will not be known to compiler.
Example: “ArrayIndexOutOfBounds” is an example of an unchecked exception, while “ClassNotFound” is an example of a checked exception.	Example: “OutOfMemory” and “StackOverflow” are examples of errors.

Notes

3.1.3 Exception Handling: Part 3

Keywords which are used in Exception Handling are try, catch, throw, throws and finally.

The most important process or mechanism of handling runtime errors (so that the normal flow can be maintained) in Java is known as Exception Handling. The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained. The use of exceptions to manage errors has some advantages over traditional error-management techniques. Below are the methods discussed by which one can catch and handle exception.

Try and Catch

In “try” block you have to place those code that you think can generate exception. The try block must be followed by either catch block or finally block. You cannot use try block alone. In “catch” block you have to place code that will handle exception.

Syntax:

```
try {
// Block of code that can generate exception  }
catch(Exception e) {
// Block of code to handle exception          }
```

A method catches an exception using a combination of try and catch keyword. A try block is placed around the code that might generate or throw an exception. A catch block is used to handle the exception.

A try block can be followed by multiple catch blocks. But at a time only one catch block is executed.

Example:

```
class ExceptionDemo {
public static void main(String args[]) {
try{
int[] a = new int[2];
System.out.println(“Trying   to access a[5]: “+ a[5]);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println(“Exception  thrown:      “+    e); }
System.out.println(“ Out of block “);
}
}
```

Output:

```
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 5
Out of block
```

Multiple Catch Block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Notes

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

```
public class MultipleCatchBlockDemo {
public static void main(String[] args) {
try{
int b=5, c=0;
int a = b/c;
System.out.println(a);
}
catch (ArithmeticExceptione)
{System.out.println (“Arithmetic Exception occurs”);
}
catch(ArrayIndexOutOfBoundsExceptione)
{ System.out.println(“ArrayIndexOutOfBounds Exception occurs”);
}
catch(Exception e)  {
System.out.println(“Parent Exception occurs”);
}
System.out.println(“rest of the code”);
}
}
```

Throw Keyword

The throw statement allows you to create a custom error and is used together with an exception type. There are many exception types available in Java. The “throws” keyword is used to declare exceptions. It doesn’t throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Example: Arithmetic Exception, Array Index Out Of Bounds Exception, File Not Found Exception etc.

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

If a method does not handle a checked exception, the method must declare it using the throws keyword.

Syntax:

```
throw Instance
```

Example:

```
void method(){
throw new ArithmeticException(“/ by zero”);
}
```

Notes

But this exception i.e., Instance 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.

```
public class TestThrowDemo{
static void validate(int age) {
if(age<18)
throw new ArithmeticException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[]){
validate(10);
}
}
```

Output

```
java.lang.ArithmeticException: not valid
 at TestThrowDemo.validate(TestThrowDemo.java:4)
 at TestThrowDemo.main(TestThrowDemo.java:9)
```

Throws Keyword

The Java throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

The throws keyword appears at the end of method's signature throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.

By the help of throws keyword we can provide information to the caller of the method about the exception.

Syntax:

```
return_type method_name() throws exception_class_name{
//method code
}
```

Example

```
public class ThrowsDemo {
static void checkAge(int age) throws ArithmeticException {
if (age < 18) {
```

```
throw new ArithmeticException("You must be at least 18 years old to vote.");    }
else {
System.out.println("Access granted - You are old enough!");} }
public static void main(String[] args) {
checkAge(15); // Set age to 15 (which is below 18...)
}
}
```

Output

```
run TestThrowDemo
java.lang.ArithmeticException: not valid
at TestThrowDemo.validate(TestThrowDemo.java:4)
 at TestThrowDemo.main(TestThrowDemo.java:9)
 ...
```

Example:

```
// Java program that demonstrates the use of throw
class ThrowExcep {
      static void help()
      {
       try {
      throw new NullPointerException("error_unknown");
       }
       catch (NullPointerException e) {
       System.out.println("Caught inside help().");
       // rethrowing the exception
       throw e;
       }
      }
      public static void main(String args[])
      {
       try {
       help();
       }
       catch (NullPointerException e) {
       System.out.println(
       "Caught in main error name given below:");
       System.out.println(e);
       }
      }
}
```

Notes

Notes

Output

Caught inside help().

Caught in main error name given below:

java.lang.NullPointerException: error_unknown

Finally Block

Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block. If you don't handle exception, before terminating the program, JVM executes finally block(if any exist).

Example:

```
public class TestFinallyBlock2{
public static void main(String args[]){
try{
int data=25/0;
System.out.println(data);
}    catch(ArithmeticException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
}
}
```

Output

java.lang.ArithmeticException: / by zero

finally block is always executed or each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits (either by calling System. exit() or by causing a fatal error that causes the process to abort).

Difference Between Throw and Throws

Throw	Throws
Throw keyword is used to throw an exception explicitly.	Throws keyword is used to declare an exception.
Throw is followed by an instance.	Throws is followed by a class.
Throw is used within method.	Throws is followed by class.
Throw is used inside method.	Throws is used with method signature.
Multiple exception cannot be thrown by throw keyword.	Multiple exception can be declared by throws keyword. You can write void myMethod() throws IOExcepton, SQLException

Uncaught Exceptions, Creating and Using User Defined Exception

Java actually handles uncaught exceptions according to the thread in which they occur. When an uncaught exception occurs in a particular thread, Java looks for what is

called an uncaught exception handler, actually an implementation of the interface Uncaught Exception Handler.

Notes

The latter interface has a method handleException(), which the implementer overrides to take appropriate action, such as printing the stack trace to the console. We can actually install our own instance of Uncaught Exception Handler to handle uncaught exceptions of a particular thread, or even for the whole system.

The specific procedure is as follows. When an uncaught exception occurs, the JVM does the following:

- it calls a special private method, dispatch Uncaught Exception(), on the Thread class in which the exception occurs;
- it then terminates the thread in which the exception occurred1.

The dispatch Uncaught Exception method, in turn, calls the thread's get Uncaught Exception Handler() method to find out the appropriate uncaught exception handler to use. Normally, this will actually be the thread's parent Thread Group, whose handle Exception() method by default will print the stack trace.

The Uncaught Exception Handler is an interface inside a Thread class. When the main thread is about to terminate due to an uncaught exception the JVM will invoke the thread's Uncaught Exception Handler for a chance to perform some error handling like logging the exception to a file or uploading the log to the server before it gets killed. We can set a Default Exception Handler which will be called for the all unhandled exceptions. It is introduced in Java 5 Version.

This Handler can be set by using the below static method of java.lang. Thread class.

When an exception is not caught, it is caught by a function called the uncaught exception handler. The uncaught exception handler always force the program to exit but before shut down it may perform some task.

The default uncaught exception handler puts a message to the console before it exits the program. You can set a custom function as the uncaught exception handler using the NSSet Uncaught Exception Handler function. You can obtain the current uncaught exception handler with the NSGet Uncaught Exception Handler function. We have to provide an implementation of the interface Thread. Uncaught Exception Handler and it has only one method.

Example

```
public class UncaughtExTest{
public static void main(String args[]) throws Exception{
Thread.setDefaultUncaughtExceptionHandler(new MyHandler());
throw new Exception("Test Exception");
}
 private static final class MyHandler implements ThreadUncaughtExceptionHandler{
 public void UncaughtException(Thread t, Throwable e){
 System.out.println("Caught exception:        "+e);
 }
 }
```

Notes

}

Output

Caught exception: java.lang.Exception: Test Exception

Java's build in exception

Java has several exception classes inside the package java.lang. Below Mentioned exceptions are Checked Exceptions Defined in java.lang.

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

Java has several other types of exceptions. Below mentioned exceptions are Unchecked RuntimeException.

1. ArithmeticException (Divide-by-zero kind of arithmetic error.)
2. ArrayIndexOutOfBoundsException (Array index is out-of-bounds.)
3. ArrayStoreException (Assigning an array element which has incompatible type.)
4. ClassCastException (Invalid cast.)
5. IllegalArgumentException (Illegal argument used to invoke a method.)
6. IllegalMonitorStateException (Illegal monitor operation, such as waiting on an unlocked thread.)
7. IllegalStateException (Application is in incorrect state.)
8. IllegalThreadStateException (Asked operation is not compatible with the current thread state.)
9. IndexOutOfBoundsException (Some type of index is out-of-bounds.)
10. NegativeArraySizeException (Array created with a negative size.)
11. NullPointerException (Invalid use of a null reference.)
12. NumberFormatException (Invalid conversion of a string to a numeric format.)
13. SecurityException (Security violation)
14. StringIndexOutOfBounds (Attempt to index outside the bounds of a string.)
15. UnsupportedOperationException (An unsupported operation was encountered.)

3.2 Overview of Interfaces and Threads

An interface is a reference type in Java that resembles a class. It is a grouping of static constants (final variables) and abstract methods (methods without a body), and it could also contain default and static methods. Multiple inheritance, abstraction, and the definition of contracts for classes that implement them are all accomplished through the usage of interfaces.

Notes

An interface is a reference type in Java that resembles a class. It is a grouping of static constants (final variables) and abstract methods (methods without a body), and it could also contain default and static methods. Multiple inheritance, abstraction, and the definition of contracts for classes that implement them are all accomplished through the usage of interfaces.

3.2.1 Interface: Defining Interface

To achieve abstraction there are two ways in Java. First one by abstract class and the other one by interface. So, we can tell an interface in Java is a mechanism to achieve abstraction. The way we write class like the same way we write interface in Java. Hence, you can say interface is a blueprint of a class. In interface variables must be public, static and final and all the methods must be public and abstract (the methods which does not have any body). Because all the methods are abstract hence interface provides 100% abstraction.

During the discussion of inheritance we have discussed about multiple inheritance and we know that multiple inheritance is not possible in class level but that is possible in interface level. So, we use interface to achieve abstraction and multiple inheritance.

While writing an interface, you must use interface keyword. and all the variables will by default be public, static and final.

Syntax:

```
interface interface_name {
// public static final variable
// public abstract void myMethod();
}
```

Abstract Methods in Interfaces

All methods inside interface must be abstract hence no codes will be there inside methods. Abstract methods will have a semicolon (;) at the end of closing bracket. Also remember that even if you do not explicitly write public and abstract keyword , compiler will add those keywords on behalf of you.

Syxtax:

```
void show(); //abstract methods
```

3.2.2 Implementing and Extending Interfaces

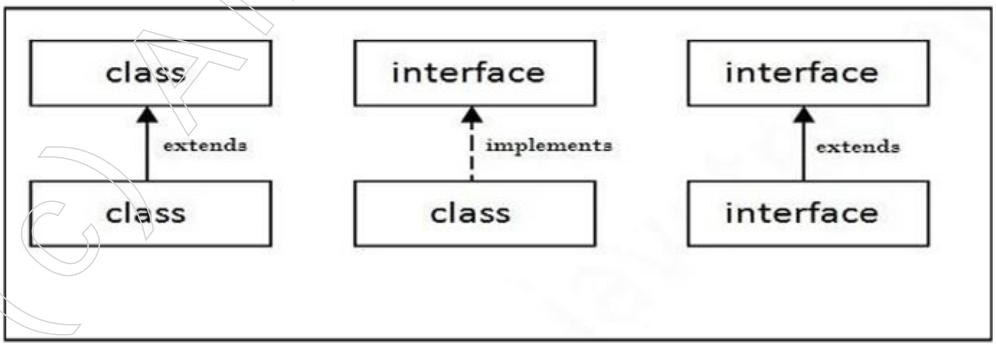


Figure: Class and interface inheritance

Notes

To inherit properties from class we have used “extends” keyword and to inherit properties from an interface we will use “extends” keyword if parent and child both are interface but we will use “implements” keyword if parent is interface type and child is class type.

Example : InterfaceExample.java

```
interface Drawable{
public abstract void draw();
}
interface Showable extends Drawable{
1void show();
interface Showable extends Drawable{
1void show();
}
public class InterfaceExample implements Showable{
public void show(){
System.out.println(“Implementing show() of Showable Interface”);
}
public void draw(){
System.out.println(“Implementing draw() of Drawable Interface”);
}
public static void main(String args[]){
InterfaceExample  obj = new InterfaceExample  ();
obj.show();
obj.draw();
}
}
```

Output:

Implementing show() of Showable Interface

Implementing draw() of Drawable Interface

Interface References

An interface is a reference type in Java and is to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

If variables are declared as interface type, it can reference any object of any class that implements the interface. We cannot create objects of interface but we can get and Java permits reference variables creation with interfaces but not objects.

Example: InterfaceReferenceExample.java

```
interface Car {
```

Notes

```
public abstract void run();
}
class Zen implements Car {
public void run() {
System.out.println("Zen is a type of Car.");
}
}
public class InterfaceReferenceExample {
public static void main(String args[]) {
Car obj1;
Zen obj2 = new Zen();
obj1 = obj2;
obj1.run();
}
}
```

Output:

Zen is a type of Car.

3.2.3 Default Methods in Interface

Default methods add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

Interfaces could only have abstract methods prior to Java 8. It is necessary to give these methods' implementation in a different class. Therefore, the class implementing the same interface must supply the implementation code for any new methods added to it. Java 8 introduced the idea of default methods to address this problem, enabling interfaces to have implemented methods without changing the classes that use the interface.

Example:

```
 interface TestInterface
{
public void square(int a);      // abstract method
default void show()  {        // default method
System.out.println("Default Method Executed");
}
}
 public class DefaultExample implements TestInterface
{
// implementation of square abstract method
public void square(int a)
```

Notes

```
{
System.out.println(a*a);
}
ublic static void main(String args[])
{
DefaultExample d = new DefaultExample();
d.square(4);
// default method executed
d.show();
}
}
// A simple program to Test Interface default
// methods in java
interface TestInterface
{
    // abstract method
    public void square(int a);
    // default method
    default void show()
    {
    System.out.println("Default Method Executed");
    }
}
class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
     System.out.println(a*a);
    }
    public static void main(String args[])
    {
     TestClass d = new TestClass();
     d.square(4);
     // default method executed
     d.show();
    }
}
```

Output

16

Default Method Executed

In order to allow existing interfaces to use the lambda expressions without having to implement the methods in the implementation class, the default methods were created for backward compatibility. Defensor methods and virtual extension methods are other names for default methods.

Static Methods: Interfaces may also have static methods, which are comparable to class static methods.

```
// A simple Java program to TestClassnstrate static
// methods in java
interface TestInterface
{
    // abstract method
    public void square (int a);
    // static method
    static void show()
    {
     System.out.println(“Static Method Executed”);
    }
}
class TestClass implements TestInterface
{
    // Implementation of square abstract method
    public void square (int a)
    {
     System.out.println(a*a);
    }
    public static void main(String args[])
    {
     TestClass d = new TestClass();
     d.square(4);

     // Static method executed
     TestInterface.show();
    }
}
```

Output:

16

Default Method Executed

Default Methods and Multiple Inheritance

The implementing class should specifically state which default method is to be used or

Notes

override the default method if both implemented interfaces contain default methods with the same method signature.

Example:

```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface TestInterface1
{
    // default method
    default void show()
    {
    System.out.println("Default TestInterface1");
    }
}
interface TestInterface2
{
    // Default method
    default void show()
    {
    System.out.println("Default TestInterface2");
    }
}
// Implementation class code
class TestClass implements TestInterface1, TestInterface2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of TestInterface1 interface
        TestInterface1.super.show();
        // use super keyword to call the show
        // method of TestInterface2 interface
        TestInterface2.super.show();
    }
    public static void main(String args[])
    {
        TestClass d = new TestClass();
```

```
    d.show();
}
}
```

Output:

Default TestInterface1

Default TestInterface2

3.2.4 Static and Constant in Interface

Static Methods in Interface are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but these methods cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

In this program, a simple static method is defined and declared in an interface which is being called in the main() method of the Implementation Class InterfaceDemo. Unlike the default method, the static method defines in Interface hello(), cannot be overridden in implementing the class.

Example:

```
interface NewInterface {
// static method
static void hello()
{
System.out.println(“Hello, New Static Method Here”);
}
// Public and abstract method of Interface
void overrideMethod(String str); }
// Implementation Class
public class InterfaceDemo implements NewInterface {
public static void main(String[] args) {
 InterfaceDemo interfaceDemo = new InterfaceDemo();
// Calling the static method of interface
NewInterface.hello();
// Calling the abstract method of interface
interfaceDemo.overrideMethod(“Hello, Override Method here”);
}
// Implementing interface method
@Override
```

Notes

Notes

```
public void overrideMethod(String str)
{
System.out.println(str);
}
}
```

Output:

Hello, New Static Method Here

Hello, Override Method Here

Constants in Interfaces

Placing constants in an interface was a popular technique in the early days of Java, but now many consider it a distasteful use of interfaces, since interfaces should deal with the services provided by an object, not its data. As well, the constants used by a class are typically an implementation detail, but placing them in an interface promotes them to the public API of the class.

Example

```
interface OlympicMedal {
static final String GOLD = “Gold”;
static final String SILVER = “Silver”;
static final String BRONZE = “Bronze”;
}
public final class OlympicAthlete implements OlympicMedal {
public OlympicAthlete(int id){
//..elided
}
//..elided
public void winEvent(){
medal = GOLD;
}
private String medal; //possibly null
}
```

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as − public interface Hockey extends Sports, Event.

Example

Notes

Starting with Java 8, interfaces could choose to define default implementations for its methods.

```
public interface Floatable {
default void repair() {
System.out.println("Repairing Floatable object");
}
}
public interface Flyable {
default void repair() {
System.out.println("Repairing Flyable object");
}
}
public class ArmoredCar extends Car implements Floatable, Flyable {
// this won't compile
}
```

If we do want to implement both interfaces, we'll have to override the repair() method.

If the interfaces in the preceding examples define variables with the same name, say duration, we can't access them without preceding the variable name with the interface name:

```
public interface Floatable {
int duration = 10;
}
public interface Flyable {
int duration = 20;
}
public class ArmoredCar extends Car implements Floatable, Flyable {
public void aMethod() {
System.out.println(duration); // won't compile
System.out.println(Floatable.duration); // outputs 10
System.out.println(Flyable.duration); // outputs 20
}
}
```

An interface can extend multiple interfaces.

Example:

```
public interface Floatable {
void floatOnWater();
}
interface interface Flyable {
void fly();
}
```

Notes

```
public interface SpaceTraveller extends Floatable, Flyable {
void remoteControl();
}
```

An interface inherits other interfaces by using the keyword extends. Classes use the keyword implements to inherit an interface.

When a class inherits another class or interfaces, apart from inheriting their members, it also inherits their type. This also applies to an interface that inherits other interfaces.

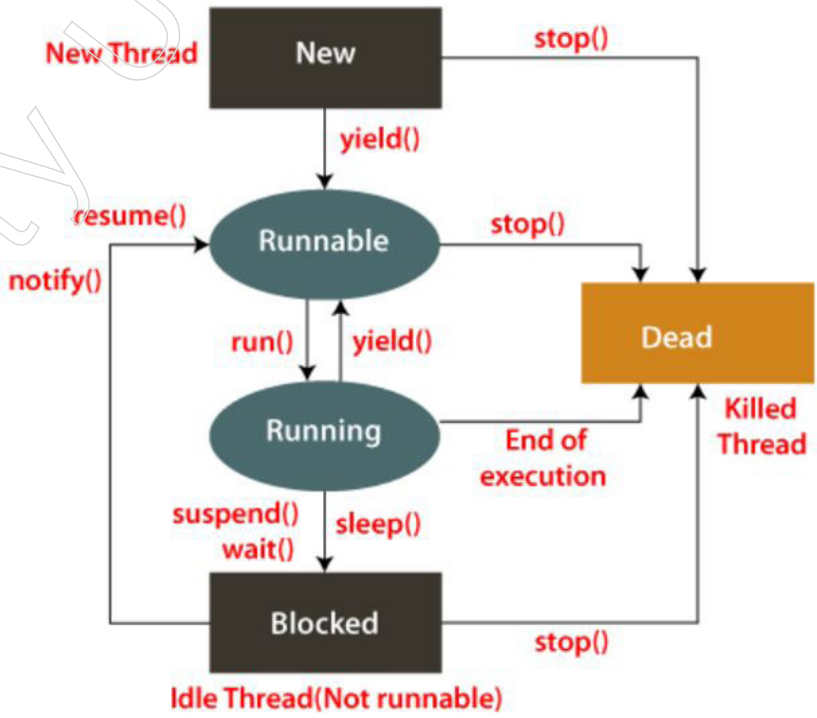
3.2.5 Threads: Part 1

Thread is nothing but a process and we must remember thread is a light weight process. We all know process is a program in execution. If we are considering a gaming app then if one car is running at a time then you can say single thread is running in the system but if multiple cars are running at the same time then you can say multiple threads are running in the system. Execution of threads start from main() method.

Thread Life Cycle

A thread can be in one of the five states in its life time and is known as thread life cycle. A thread can be in any one of new, runnable, running, blocked and terminated state. JVM controls the life cycle of the thread in Java. Below figure depicts the states of thread.

1. New: The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. Running: The thread is in running state if the thread scheduler has selected it.
4. Blocked : This is the state when the thread is still alive, but is currently not eligible to run.
5. Terminated: A thread is in terminated or dead state when its run() method exits.



Source: https://www.javatpoint.com/thread-states-in-java

Notes

Creating and Implementing Thread

There are two approaches by which we can create thread.

1. By extending Thread class:

Thread class provides constructors and methods to create thread and to perform different operations. Executions starts from main() method as we already know and also to start user thread we must depend on main thread.

Thread(), Thread(String name) are commonly used constructors of Thread class. public void start(), public void run(), public void sleep(long milliseconds), public int getPriority(), public int setPriority(int priority), public String getName(),public String setName(String name), public void yield(), public void suspend(),public void resume(),public void stop(), public boolean isAlive(), public Thread currentThread() etc. are few commonly used methods of Thread class.

Steps of creating thread:

1. Create your own process
2. Override run() method to write the logic
3. Instantiate thread
4. Start your thread with start() method

Example:

```
class MyThread extends Thread {
public void run() {
// here you have to write thread logic
for(int i=0; i<10; i++)            {
System.out.println("This is user thread"); } } }
public class ThreadDemo {
public static void main(String args[])       // main thread
{
MyThread t = new MyThread();
t.start();          // user thread has started, so now two threads are there
//logic of main thread
for(int    i     =0;     i<10;     i++)      {
System.out.println("This is main thread"); } } }
```

Output:

```
This is main thread
This is main thread
This is main thread
This is main thread
This is main thread
This is main thread
```

Notes

This is main thread

This is main thread

This is main thread

This is main thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

This is user thread

In the above example, you can see there is no start() method is explicitly mentioned in MyThread class but we can start() method (t.start()) in our program because start() method is available in Thread class as we discussed earlier.

When we call t.start() then JVM searches in MyThread class but as it is not there so JVM will search in Thread class and Thread class's start() automatically will call run(). So, run() method will run when we call t.start().

If your application contains three threads and execution starts from main method, so now the question is which thread will run next? Thread execution will be decided by Thread scheduler which is having algorithms like preemptive, time slicing etc. Thread schedular is part of JVM.

2. By implementing Runnable interface:

You can create thread by implementing Runnable interface.

Example:

```
class RunnableExample implements Runnable{
public void run(){
for(int i = 0; i<10; i++) {
System.out.println("User Thread");
}
}
public static void main(String args[]) {
RunnableExample m=new RunnableExample ();
Thread t =new Thread(m);
t.start();
for(int i = 0; i<10; i++) {
```

```
System.out.println("Main Thread");
}
}
}
```

Output:

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
```

In the above example you can see we have not written m.start() because neither RunnableExample class nor Runnable class have start() method in them. So, here we need to take Thread class's help and have to create object of Thread class and passing object of RunnableExample class. Then with the help of Thread class's object, we are calling start() method.

3.2.6 Threads: Part 2

Multi-Threaded Programming

Multithreading is a process of executing multiple threads simultaneously. Why are we using multithreading and not multiprocessing because threads use shared memory area. When child thread is created it shares the memory area of parent thread. So saves memory, and context-switching between the threads takes less time than process.

Notes

Notes

Advantage Of Multithreading:

1. Multiple operations can be done at the same time. Hence, saves time.
2. Threads are independent. If an exception occurs in a single thread, it doesn't affect other threads.

Example: MultiThreadDemo.java

```
class Thread1 extends Thread{
public void run(){
System.out.println("Thread 1");
}
}
class Thread2 extends Thread{
public void run(){
System.out.println("Thread 2");
 }
}
class Thread3 extends Thread{
public void run(){
System.out.println("Thread 3");
}
}
class MultiThreadDemo extends Thread{
public static void main(String args[]){
 Thread1 t1 = new Thread1();
 t1.start();
 Thread2 t2 = new Thread2();
 t2.start();
 Thread3 t3 = new Thread3();
 t3.start();
 }
 }
 Output:
 Thread 1
 Thread 3
 Thread 2
```

Thread Priorities

Each and every thread is having some priority. If multiple threads are having same priority then thread scheduler decides which thread will run and when. Priorities are number ranging from 1 to 10. In Thread class there are 3 types of constants.

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY

3. public static int MAX_PRIORITY

If no priority is mentioned for a thread that means it holds default priority 5 which is known as NORM_PRIORITY. The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Notes

Example: PriorityExample.java

```
class PriorityExample extends Thread{
public void run(){
System.out.println("The running thread is:"+Thread.currentThread().getName());
System.out.println("Running
getPriority());
}
thread's
public static void main(String args[]){
PriorityExample m1=new PriorityExample();
m1.setPriority(Thread.MAX_PRIORITY);
m1.start();
}
}
```

Output:

```
The running thread is:Thread-3
The running thread's priority is:10
```

3.2.7 Threads: Part 3

Synchronization of Thread

Synchronization of Thread is the capability to manage concurrent execution of two or more threads to shared resources. To avoid critical resource use conflicts, threads should be synchronized. Otherwise, conflicts may happen when two threads those are running parallel and attempt to modify a common variable at the same time. If we want to allow only one thread to access the shared resource then Java Synchronization is better option.

Let us take an real life example in order to make the clear concept about synchronization and non-synchronization. Think of railway ticket reservation system where people are waiting in a queue to buy ticket. Here data is consistent synchronized (only one thread at a time able to access). But in this case waiting time increases and performance decreases.

Now think of reservation system in application level where all passengers can buy ticket online, here data inconsistency occurred because multiple threads are running at a time. But in this case waiting time decreases and performance increases.

Example: NonSyncTest.java

```
//Program without synchronization
class Table{
```

Notes

```
void printTable(int n){ //method is not synchronized
 for(int                    i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class NonSyncTest{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
```

Notes

200

10

300

15

400

20

500

25

Java Synchronized Method

If synchronized keyword is used before any method then it is known as synchronized method. This method lock an object for shared resource. A thread automatically acquires the lock for an object when a thread invokes a synchronized method and releases it when the thread completes its task.

Example: SyncTest.java

```
class Table{
synchronized void printTable(int n){//synchronized method
for(int                    i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
 }
 }
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
```

Notes

```
t.printTable(100);
}
}
public class SyncTest{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Non synchronized means multiple threads are running at a time. Reservation system without queue is non synchronized.

Resuming and Stopping Threads

The suspend() method of thread class sends the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily stop execution. The suspended thread can be resumed using the resume() method.

The suspend(), resume() and stop() methods are deprecated because suspend() sometimes causes serious system problem though resume() method did not create any problem but it has also been deprecated because it cannot work without suspend() method. The stop() method also caused system problem hence it has also been deprecated.

Summary

- Exception handling in Java is a robust mechanism that manages runtime errors, ensuring the normal flow of application execution. It involves the use of try, catch, finally, and throw blocks. The try block contains code that might throw an exception, while catch blocks handle specific exceptions that may occur. The finally block, if present, executes regardless of whether an exception was thrown, typically used for cleanup activities.

Notes

The throw statement is used to explicitly throw an exception. Java's exception handling framework allows developers to create reliable and maintainable code by separating error handling from regular code logic and providing a way to handle unexpected situations gracefully.

- An interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors, and the methods in interfaces are abstract by default, meaning they do not have a body. An interface is defined using the interface keyword, and any class that implements an interface must provide concrete implementations for all of its methods, unless the class itself is abstract.
- In Java, starting from Java 8, interfaces can contain static methods. These methods are associated with the interface itself rather than with any instance of a class that implements the interface. Static methods in interfaces can have a body and can be called without creating an instance of the implementing class.
- Thread is a lightweight sub-process, the smallest unit of processing. It is a path of execution within a program and enables concurrent execution of two or more parts of a program for maximum utilization of CPU. Java provides built-in support for multithreading by way of the java.lang.Thread class and the java.lang.Runnable interface.

Glossary

- Exception handling in Java is a mechanism that allows a program to deal with unexpected or exceptional situations during runtime.
- Checked Exceptions: These are exceptions that the Java compiler requires you to handle explicitly using try-catch blocks or declare in the method signature using the throws keyword. Examples include IOException and SQLException.
- Unchecked Exceptions (RuntimeExceptions): These are exceptions that the compiler does not require you to handle explicitly. They usually indicate programming errors and are subclasses of RuntimeException. Examples include NullPointerException and ArithmeticException.
- try-catch Blocks: The try block contains the code where an exception might occur. The catch block handles the exception if it occurs. Multiple catch blocks can be used to handle different types of exceptions.
- In Java, an interface is a reference type that is similar to a class. It is a collection of abstract methods (methods without a body) that are meant to be implemented by classes. Interfaces provide a way to achieve abstraction, allowing a class to declare its contract (set of methods) without providing the implementation.
- In Java, a thread is the smallest unit of execution within a process. It represents an independent path of execution and allows a program to perform multiple tasks concurrently.

Check Your Understanding

1. What is an exception in Java?
   a) A syntax error
   b) An abnormal event during program execution
   c) A runtime error
   d) A compile-time error

Notes

2. Which keyword is used to handle exceptions in Java?
   a) try
   b) catch
   c) throw
   d) finally
3. In a try-catch block, where should the code that may throw an exception be placed?
   a) Inside the try block
   b) Inside the catch block
   c) Outside both try and catch
   d) Inside the finally block
4. What is the purpose of the finally block in exception handling?
   a) To handle exceptions
   b) To specify code that must be executed regardless of whether an exception is thrown or not
   c) To throw exceptions
   d) To catch exceptions
5. Which of the following elements can an interface in Java contain?
   a) Static constants, abstract methods, default methods, and instance variables
   b) Static constants, abstract methods, default methods, and static methods
   c) Static constants, abstract methods, constructors, and instance methods
   d) Static constants, abstract methods, final methods, and static methods

Exercise

1. Discuss in brief about exception handling.
2. Explain default methods in interface.
3. Summarise static and constant in interface.
4. Write in short about threads.

Learning Activities

1. Consider the following Java code snippet. Identify and correct the issues related to exception handling:

```
import java.util.Scanner;
public class DivideNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print(“Enter the numerator: “);
        int numerator = scanner.nextInt();
        System.out.print(“Enter the denominator: “);
        int denominator = scanner.nextInt();
        int result = divideNumbers(numerator, denominator);
        System.out.println(“Result: “ + result);
```

```
        scanner.close();
    }
    public static int divideNumbers(int numerator, int denominator) {
    int result = 0;
        try {
         result = numerator / denominator;
    } catch (ArithmeticException e) {
         System.out.println("Error: Division by zero.");
    } finally {
          System.out.println("Division operation completed.");
    }
        return result;
    }
}
```

- ❖ Identify and fix any issues related to exception handling in the code.
- ❖ Ensure that the program handles potential division by zero errors gracefully.
- ❖ Implement proper exception handling practices.

2. Write a program to multi-threaded application that simulates a simple banking system with multiple accounts. Each account has a balance, and transactions (withdrawals and deposits) can occur concurrently.

Check Your Understanding- Answers

1. b) 2. a) 3. a) 4. b)
5. b)

Further Readings and Bibliography

1. R. Nageswara Rao. (2016). Core Java: An Integrated Approach. Dreamtech Press, 720 pages.
2. E. Balagurusamy. (2023). Programming with Java. 7th Edition. McGrawHill Publications, 592 pages.
3. Barry A. Burd. (2017). Beginning Programming with Java for Dummies. 5th Edition. Wiley Publications.
4. Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action. Dreamtech Press.

Notes

Notes

Module - IV: Java Packages and GUI

Learning Objectives

At the end of this module, you will be able to:

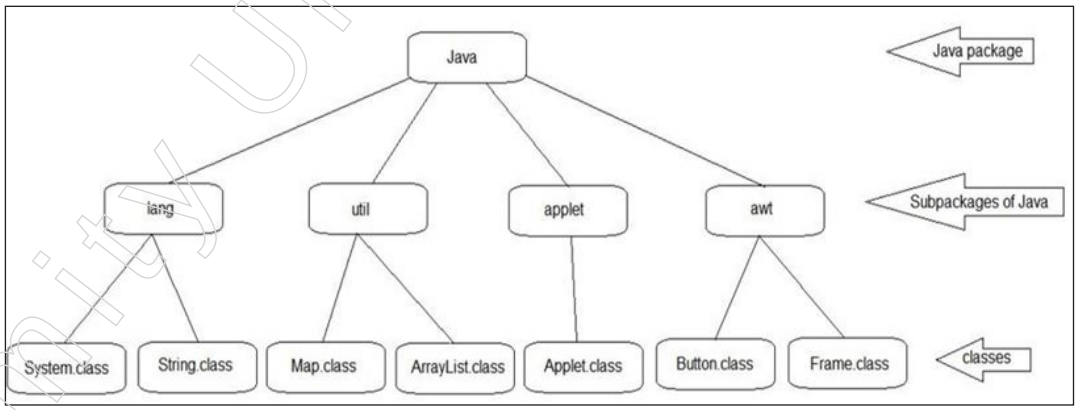
- Discuss about the advantages of using packages
- Define types of packages
- Summarise built-in packages
- Create user define packages
- Discuss how to Create strings and Handle strings
- Define Applet class
- Create your own applet
- Discuss about graphics class

Introduction

Packages are used in Java to group and arrange related classes and interfaces. They provide Java projects a structure and assist in preventing naming conflicts. Numerous libraries and frameworks for graphical user interface creation are available in Java. Swing and AWT (Abstract Window Toolkit) are two frequently used libraries.

4.1 Overview of Packages

A group of similar types of classes, interfaces and sub-packages are known as Package. There are two types of Packages in java, built-in package and user-defined package. Examples of built-in packages such are java, lang, awt, javax, swing, net, io, util, sql etc.



Source: https://www.javatpoint.com/package

4.1.1 Introduction to Packages

A package is a way to arrange classes and interfaces in a hierarchical structure in Java. It facilitates better code organisation, helps to prevent naming conflicts, and offers a mechanism to control which classes are visible within a project.

Advantage of Java Package

- Removes naming collision. Java package is used to categorize the classes and interfaces so that they can be easily maintained.

- Java package provides access protection.
- Java package removes naming collision.

Notes

The Java language provides a huge class library (i.e. set of packages) for using in your applications which is known as the Application Programming Interface (API). For example, a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the file system , a String object contains state and behaviour for character strings, various Graphical User Interface (GUI) objects like text box, buttons, checkboxes and many other things related to graphical user interfaces. These allows a programmer to design of particular application.

4.1.2 Importing and Implementing Packages

Types of Packages:

There are two types of packages as we already know. Predefine package and user define package. Predefine packages are defined by Java programming language along with Java software. You already know lang, IO, awt, net etc are examples of predefined package.

a) User define package: User defined packages are packages that are created by developer as per their application requirement.

b) Creation of package: To create a package, you have to use package keyword.Package statement must be the first statement in your program. It must have unique name.

Syntax: package package_name

Example: PackageExample.java

```
package mypackage;
class PackageExample {
public static void main(String args[]) {
System.out.println(“My first package.”);
}
}
```

To compile java package below steps you have to follow if you are using command prompt.

Syntax: javac -d directory javafilename

To compile our above program we have to write: javac -d . PackageExample.java

The above command forces the compiler to create a package. The “.” operator represents the current working directory. You can use any directory name like /home (in case of Linux), d:/program (in case of windows) etc.

To run java package program, you need to use fully qualified name e.g. mypackage. PackageExample to run the class.

To Compile: javac -d . PackageExample.java

To Run: java mypackage.PackageExample

Output: My first package.

When you execute the code, it creates a package mypackage. When you open the Java package mypackage inside that you will find PackageExample.class file. If you want to

Notes

create subpackage (Package inside another package is called subpackage) of mypackage then you have to write as mypackage.mysubpackage.

Importing Packages:

There are 3 ways by which you can access package from another program.

- import mypackage.*;
- import mypackage.PackageExample ;
- by fully qualified name.

Example of package that import the packagename:

```
//save by Example1.java
package pack1;
public class Example1{
public void msg(){System.out.println("I am in package 1");
}
}
// save byPackageExample.java
import    pack1.*; //*implies all classes of package pack1
class PackageExample{
public static void main(String args[]){
Example1 obj = new Example1();
obj.msg();
}
}
```

Output:

I am in package 1

Example of package that import the packagename.classname

```
//save by Example1.java
package pack1;
public class Example1{
public void msg(){System.out.println("I am in package 1");
}
}
//save by PackageExample.java
import pack1.Example1;
class PackageExample{
public static void main(String args[]){
Example1 obj = new Example1();
obj.msg();
```

Notes

```
}
}
```

Output:

I am in package 1

Example of package using fully qualified name

```
//save by Example1.java
package pack1;
public class Example1{
public void msg(){System.out.println("I am in package 1");
}
}
//save by PackageExample.java
class PackageExample{
public static void main(String args[]){
pack1.Example1 obj = new pack1.Example1();
obj.msg();
}
}
```

Output:

I am in package 1

All the classes and interface of a package will be imported excluding the classes and interfaces of the subpackages when you import a package. So, you need to import the subpackage too.

Predefine Packages

Lang Package Classes: It is a default package and we do not need to import this package in our program. All basic classes and interfaces which are required to prepare basic programs are String, StringBuffer, System, Thread, Runnable, all wrapper classes, Exceptions and its subclasses are residing inside lang package.

4.1.3 IO Packages

Java Input and Output (I/O) is used to process the input and produce the output. To make I/O operation fast, Java uses the concept of a stream. The java. io package contains all the classes required for input and output operations. We can perform file handling in Java by Java I/O API.

It provides predefined classes and interfaces to perform input - output (I/O) operations. InputStream, FileInputStream, OutputStream, FileOutputStream, Reader, BufferedReader, InputStreamReader, Writer are residing inside IO package.

Input streams and Output streams:

A sequence of data is known as streams. A stream is composed of bytes. In Java, 3 streams are created for us automatically.

Notes

1. System.in refers to standard input stream
2. System.out refers to standard output stream
3. System.err refers to standard error stream

Input stream: An input stream is used to read data from source and source may be a device, a file, an array or socket. InputStream class is an abstract class. Subclasses of InputStream are FileInputStream, DataInputStream, ObjectInputStream etc.

Example:

System.in:

int i=System.in.read();//returns ASCII code of 1st character

System.out.println((char)i);//will print the character

Output stream: An output stream is used to write data to destination and destination it may a device, a file, an array or socket. OutputStream class is an abstract class. Subclasses of OutputStream are PrintStream, FileOutputStream, DataOutputStream etc.

System.out

System.out.println("simple message");

Standard Error: This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

System.err

System.err.println("error message");

Sample programs on I/O files:

Below example shows how to read file content line by line. To get this, you have to use BufferedReader object. By calling readLine() method you can get file content line by line. readLine() returns one line at each iteration, we have to iterate it till it returns null.

Example: ReadLinesExample.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class ReadLinesExample
public static void main(String a[])
BufferedReader br = null;
String strLine = "";
try {
br = new BufferedReader( new FileReader("fileName"));
while( (strLine = br.readLine()) != null)
System.out.println(strLine);
}
```

```
} catch (FileNotFoundException e) {
System.err.println("Unable to find the file: fileName");
} catch (IOException e) {
System.err.println("Unable to read the file: fileName");
}
}
}
```

4.1.4 Types of Packages: User Defined and Built-in Package

In Java, a group of classes, interfaces, and sub-packages can be contained within a package. It is used in Java to facilitate the usage and search of enumerations, classes, interfaces, and annotations. It also qualifies as data encapsulation. Put another way, a package can be thought of as a container for a collection of related classes, some of which are reserved for internal use and others of which are available.

User-defined Packages

Packages built or created by the developer to classify classes and packages are known as user-defined packages. They resemble Java's built-in features quite a bit. It can be used in the same way as built-in packages and imported into other classes. However, the class names are placed into the default package—which has no name—if the package declaration is left out.

The package keyword is what we should utilise in order to construct a package.

Syntax:

```
package package-name;
```

Steps to create User-defined Packages

Step 1: Building a Java class package. The format is really easy to follow and basic. Simply type a package by going with its name.

```
package example1;
```

Step 2: Add the class to the Java package, but keep in mind that it has a single package declaration.

```
package example1;
class gfg {
public static void main(Strings[] args){
-------function--------
}
}
```

Step 3: After the user-defined package has been successfully established, its functions can be used and imported into other packages.

Note: it will be added to the current default package if there are no classes written in the package.

The examples that follow begin with the creation of a user-defined package called "example," under which is a class called "gfg," which has a message-printing function. In

Notes

the second example, we will use a function from our user-defined package “example” and import it.

Example 1: In this example, we’ll create a user-defined package and function to print a message for the users.

```
// Java program to create a user-defined
// package and function to print
// a message for the users
package example;
// Class
public class gfg {
public void show()
{
System.out.println(“Welcome to Java”);
}
public static void main(String args[])
{
gfg obj = new gfg();
obj.show();
}
}
```

Output:

Welcome to Java

Example 2: We’ll import the user-defined package “example” made in the previous example in the example that follows. and make use of the message printing feature.

```
import example.gfg;
public class GFG {
public static void main(String args[])
{
    gfg obj = new gfg();
    obj.show();
    }
}
```

Output

Welcome to Java

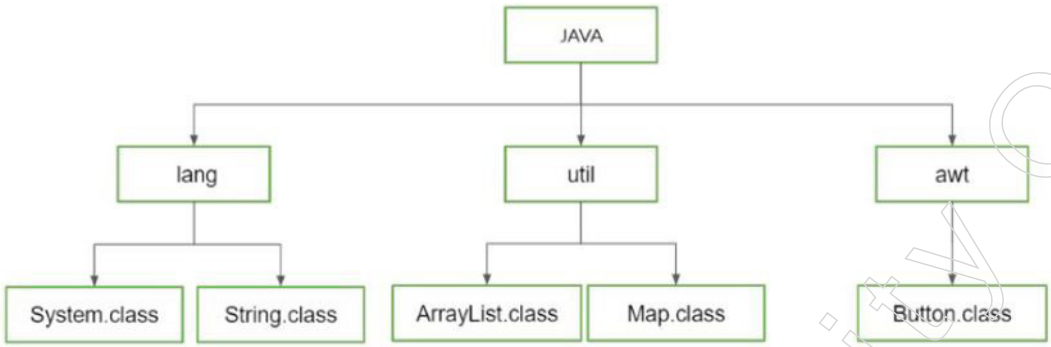
The user-defined package “example” was first formed in the aforementioned examples, and under it was the class name “gfg,” which included a function that printed greetings. Upon code compilation and execution, we will see the message “Hey, nerds! “How are you doing?” Additionally, we used the function display and imported the user-defined package

named “example” in the following example. We have a welcoming message, as you can see from the output of both programmes.

Notes

Built-in Packages

Built-in packages are those that are downloaded with JDK or JRD. The built-in packages are contained in JAR files, which we can easily view when we unzip them. Some of the packages in JAR files are lang, io, util, SQL, and so on. Numerous built-in packages are available for Java, such as java.awt.



Examples of Built-in Packages

1. java.sql: Provide the classes needed to handle and retrieve data from a database. This package contains classes such as PreparedStatement, ResultSet, Statement, Connection, DriverManager, and so on.
2. java.lang: includes interfaces and classes that are essential to the Java programming language’s architecture. This package contains classes such as String, StringBuffer, System, Math, and Integer, among others.
3. java.util: It includes the random number generator classes, properties, the collections framework, and various support classes for internationalisation. This package contains classes such as Calendar, Date, Time Zone, LinkedList, HashMap, and ArrayList.
4. java.net: It offers courses for putting networking applications into practice. This package contains classes such as Authenticator, HTTP Cookie, Socket, URL, URLConnection, URLEncoder, URLDecoder, etc.
5. java.io: It provide classes for input/output operations on systems. This package contains classes such as BufferedReader, BufferedWriter, File, InputStream, OutputStream, PrintStream, Serializable, etc.
6. java.awt: It includes classes for painting graphics and images, as well as for developing user interfaces. This package includes classes such as Button, Colour, Event, Font, Graphics, Image, and so on.

Java. awt package: The abstract window kit’s primary package is this one. It also defines Java’s graphical user interface (GUI) foundation and contains classes for graphics, including Java 2D graphics. This package contained a number of large Java-based GUI objects. package swing.

```
import java.awt.*;
public class AWTExample{
AWTExample()
{
    Frame fr1=new Frame();
```

Notes

```
        Label la = new Label("Welcome to the java graphics ");
        fr1.add(la);
        fr1.setSize(200, 200);
        fr1.setVisible(true);
    }
    public static void main(String args[])
    {
        Testawt tw = new Testawt();
    }
    }
```

Output

In the example above, we used the awt package to construct a 200 x 200 frame, and we then printed the message "Welcome to the Java graphics."

Java.net package:

The networking classes and interfaces found in this package include the DatagramSocket, MulticastSocket, URLConnection, Socket, and Server classes. Java is compatible with TCP (Transmission control protocol) and UDP (User Datagram Protocol, a connection-less protocol) among other network protocols.

```
// using DatagramSocket under java.net package
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class UDP {
    public static void main(String[] args)
    throws IOException
{
    // Create a socket to listen at port 4567
    int port_no = 4567;
```

Notes

```
        DatagramSocket ds = new DatagramSocket(port_no);
        byte[] receive = new byte[65535];
        DatagramPacket DpReceive = null;
    while (true) {
        // create a DatgramPacket to receive the data.
        DpReceive = new DatagramPacket(receive, receive.length);
        // receive the data in byte buffer.
        ds.receive(DpReceive);
        System.out.println("DATA:- " + data(receive));
        // Clear the buffer after every message.
        receive = new byte[65535];
        }
    }
    }
```

Output

DATA:- //whatever to get from server

We have developed a UDP socket receiver in the aforementioned example, which receives data from the server-side via port no. To accept the data in the byte buffer, we first constructed a socket using the DatagramSocket function to listen on port 4567. Next, we created a datagram packet to receive the data, and finally, once we had the data, we printed it.

java.sql package

The classes and interfaces needed to supply and carry out all JDBC operations, including writing and running SQL queries, are included in this package. It offers an API for accessing and processing databases that are stored in databases, as well as a framework where various drivers can be dynamically added to access various databases.

java.lang package

The Java language's fundamental classes are contained in this package. Every programme automatically imports this package, thus we may utilise its classes directly in our own programmes. Here's an example using the Math class from the lang package, which offers a number of functions and methods for doing mathematical operations.

```
import java.lang.*;
class example_lang {
public static void main(String args []) {
    int a = 100, b = 200,maxi;
    maxi = Math.max(a,b);
    System.out.printf("Maximum is = "+maxi);
    }
}
```

Output

Maximum is = 200

Notes

In the example above, we utilise the Math class and the lang package to use the "max" method to find the maximum of two digits. First, we made three integer variables, maxi, "a," and "b." where maxi will store the highest value from either an or b and a and b = 200. We utilised the max function to get the maximum and saved it in maxi after giving the variable values. Therefore, the greatest number of two that we can observe in the desired output is 200.

java.io package

Classes and an interface for managing the system (input/output) are provided by this package. With the help of these classes, programmers may receive user input, process it, and present the results to the user. In addition, we may use the classes in this package to read and write file handling operations.

```
import java.io.Console;
class example_io {
public static void main(String args []) {
    Console cons = System.console();
    System.out.println("Enter your favorite colour");
    String colour ;
    colour = cons.readLine();
    System.out.println("Favorite colour is " + colour);
    }
}
```

Output

```
Enter your favorite colour
RED
Favorite colour is RED
```

In the aforementioned example, we utilise the Console class and the java.io package to gather user input, process it, and then display the outcome on the console. According to the code, we invite the user to enter colour by displaying the message "Enter your favourite colour" initially. then the user system used cons to read that line.readLine(), save it in a string variable, and then display the line at the end.

java.util package

This package gives Java programmers the essential tools they need. Arrays is the most frequently used class in this package, and programmers utilise it for a variety of purposes. Because it makes it simple to use pre-written classes to accomplish various criteria, we may argue that it is the most valuable package for Java. Let's look at an Arrays class example.

```
import java.util.Arrays;
public class MyClass {
    public static void main(String args[]) {
    int[] arr = { 40, 30, 20, 70, 80 };
    Arrays.sort(arr);
```

```
        System.out.printf("Sorted Array is = "+Arrays.toString(arr));
        }
}
```

Output:

Sorted Array is = [20, 30, 40, 70, 80]

The Arrays class and the Java.util package were utilised in the aforementioned example to sort the integer array. Using the "sort" function of the Arrays class, we first constructed an integer type array with the items 40, 30, 20, 70, and 80. We will print the sorted array in the appropriate output once it has been sorted.

4.2 String Handling

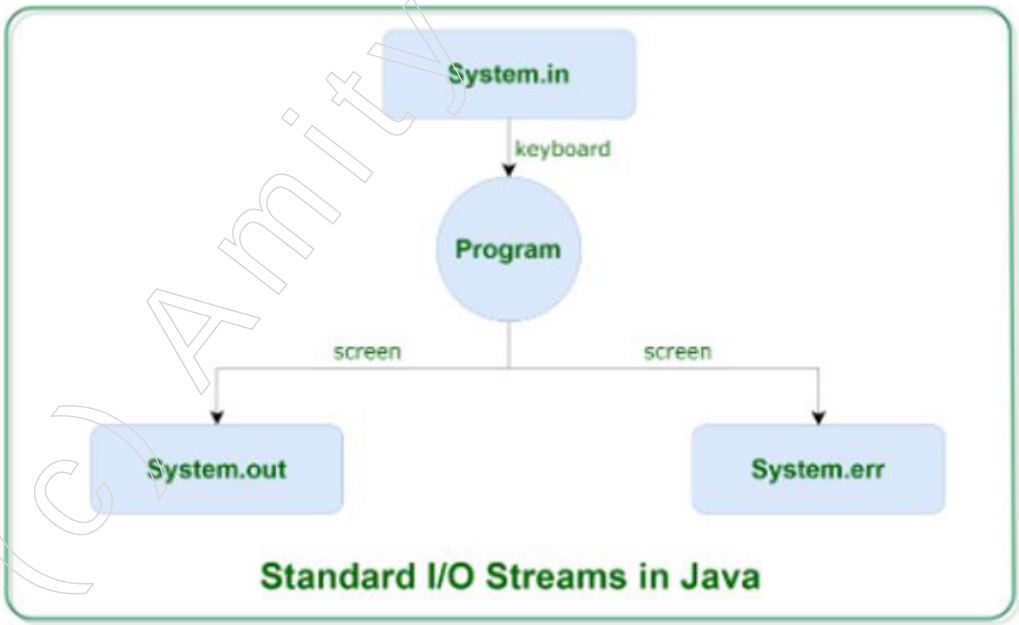
Programming requires the ability to handle strings because strings are frequently used to represent and manipulate textual data. This is especially true for Java. Java offers a wide range of features and techniques for handling strings.

4.2.1 Programs on IO Files

With its I/O package, Java offers a variety of Streams that enable the user to carry out all input-output tasks. To completely perform I/O operations, these streams support any object, data, character, file, and other types.

Source: https://www.geeksforgeeks.org/java-io-input-output-in-java-with-examples/

Let's examine the three standard or default streams that Java offers, which are also the most frequently used, before delving into other input and output streams:



Notes

System.in: Characters from the keyboard or any other standard input device are read using this standard input stream.

System.out: This is the typical output stream that shows a program's output on an output device, such as a computer screen.

The different print functions that we utilise to output statements are listed below:

print():To show text on the console, utilise Java's text display function. The text in question is sent to this procedure as a String parameter. With this method, the text is printed on the console while the cursor stays at the end of the text. This is where the next printing is done.

Syntax:

```
System.out.print(parameter);
```

```
// Java code to illustrate print()
import java.io.*;
class Demo_print {
    public static void main(String[] args)
    {
        // using print()
        // all are printed in the
        // same line
        System.out.print("Demo! ");
        System.out.print("Demo! ");
        System.out.print("Demo! ");
    }
}
```

Output:

Demo! Demo! Demo!

println():You can also use this Java method to see text displayed on the console. The cursor advances to the beginning of the following line at the console as the text is printed there. The subsequent line is where the following printing begins.

```
// A Java program to demonstrate working of printf() in Java
class JavaFormatter1 {
    public static void main(String args[])
    {
    int x = 100;
    System.out.printf(
        "Printing simple"
        + " integer: x = %d\n", x);
    // this will print it upto
```

Notes

```
        // 2 decimal places
        System.out.printf(  "Formatted with"  + " precision: PI = %.2f\n", Math.PI);
        float n = 5.2f;
        // automatically appends zero
        // to the rightmost part of decimal
        System.out.printf(
            "Formatted to "  + "specific width: n = %.4f\n", n);
        n = 2324435.3f;
        // here number is formatted from
        // right margin and occupies a
        // width of 20 characters
        System.out.printf( "Formatted to " + "right margin: n = %20.4f\n",  n);
    }
    }
```

Output:

Printing simple integer: x = 100

Formatted with precision: PI = 3.14

Formatted to specific width: n = 5.2000

Formatted to right margin: n = 2324435.2500

System.err: All error data that a programme may throw is output into a computer screen or other standard output device using this standard error stream.

The error data is also output by this stream using all three of the previously described functions:

- print()
- println()
- printf()

```
// Java code to illustrate standard
// input output streams
import java.io.*;
public class SimpleIO {
    public static void main(String args[])
    throws IOException
{
    // InputStreamReader class to read input
    InputStreamReader inp = null;
    // Storing the input in inp
    inp = new InputStreamReader(System.in);
```

Notes

```
        System.out.println("Enter characters, " + " and '0' to quit.");
        char c;
    do {
        c = (char)inp.read();
        System.out.println(c);
        } while (c != '0');
    }
    }
```

Input:

Welcome0

Output:

Enter characters, and '0' to quit.

W

e

l

c

o

m

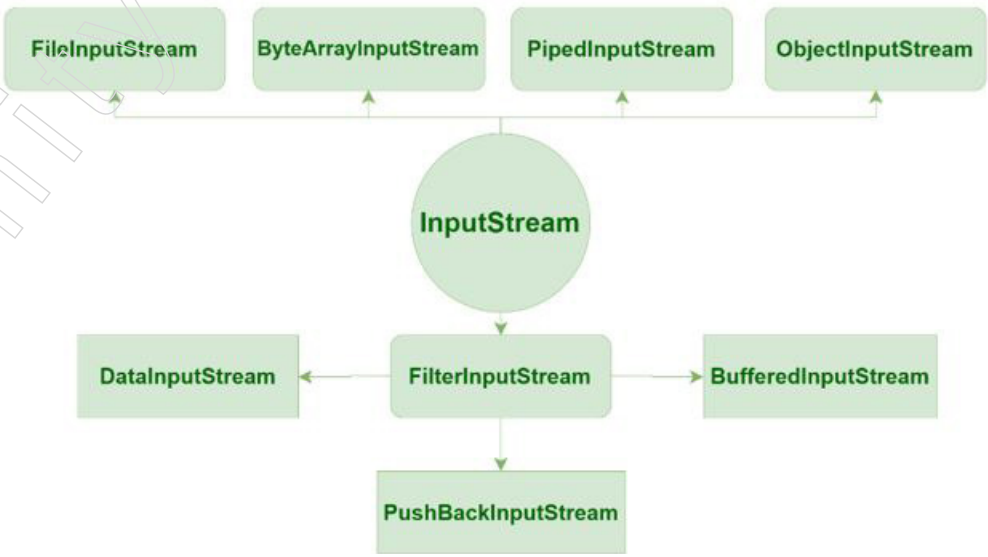
e

0

Types of Streams:

Streams fall into one of two main categories, depending on the kind of operations they perform:

Input Stream: Data that has to be read as input from a source array, file, or any peripheral device is read using these streams. For instance, ByteArrayInputStream, FileInputStream, and BufferedInputStream.

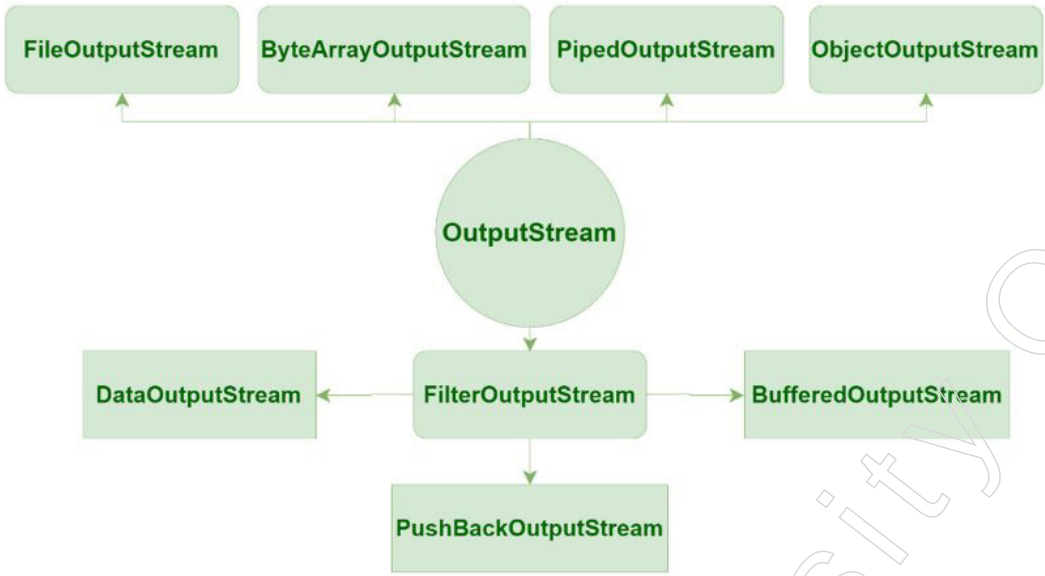


Various InputStream Classes

Source: https://www.geeksforgeeks.org/java-io-input-output-in-java-with-examples/

Notes

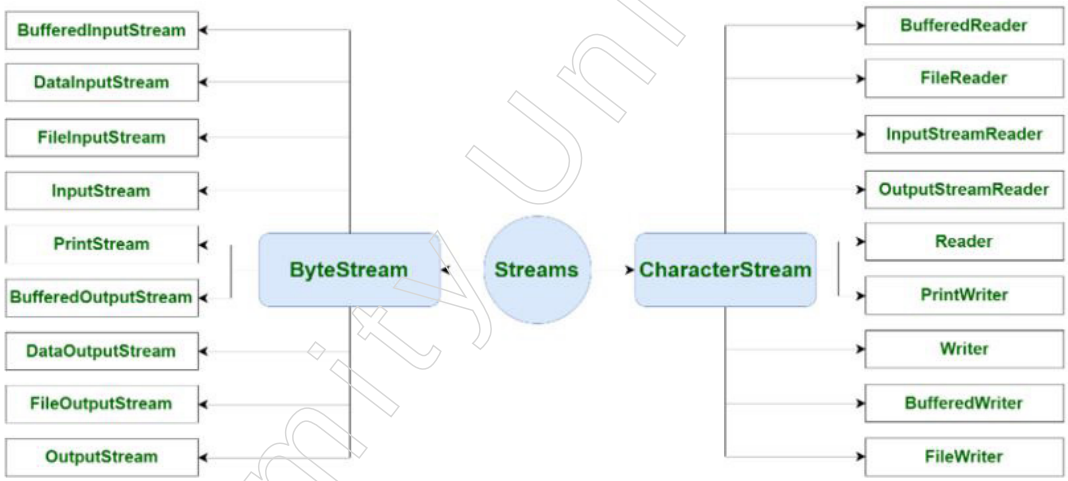
Output Stream: Data can be written via these streams into files, arrays, or any other output-related peripheral device. For instance, ByteArrayOutputStream, FileOutputStream, and BufferedOutputStream.



Various OutputStream Classes

Source: https://www.geeksforgeeks.org/java-io-input-output-in-java-with-examples/

Streams can be classified into two main classes based on the kinds of files they include. These classes can then be further subdivided into further classes, as shown in the image below, which is followed by an explanation.



Steam Classifications based on file types

Source: https://www.geeksforgeeks.org/java-io-input-output-in-java-with-examples/

ByteStream: Data is processed using this, byte by byte (8 bits). Despite having numerous classes, the most widely used ones are FileInputStream and FileOutputStream. The FileOutputStream is used to write to the destination, and the FileInputStream is used to read from the source. The list of different ByteStream Classes is as follows:

Notes

Stream class	Description
BufferedInputStream	It is used for Buffered Input Stream.
DataInputStream	It contains method for reading java standard datatypes.
FileInputStream	This is used to reads from a file
InputStream	This is an abstract class that describes stream input.
PrintStream	This contains the most used print() and println() method
BufferedOutputStream	This is used for Buffered Output Stream.
DataOutputStream	This contains method for writing java standard data types.
FileOutputStream	This is used to write to a file.
OutputStream	This is an abstract class that describe stream output.

```
// Java Program illustrating the
// Byte Stream to copy
// contents of one file to another file.
import java.io.*;
public class BStream {
    public static void main(
    String[] args) throws IOException
{
    FileInputStream sourceStream = null;
    FileOutputStream targetStream = null;
    try {
        sourceStream
        = new FileInputStream(“sorcefile.txt”);
        targetStream
        = new FileOutputStream(“targetfile.txt”);

        // Reading source file and writing
        // content to target file byte by byte
        int temp;
        while ((
        temp = sourceStream.read())
                != -1)
```

```
        targetStream.write((byte)temp);
    }
        finally {
        if (sourceStream != null)
        sourceStream.close();
        if (targetStream != null)
        targetStream.close();
        }
    }
    }
```

Output:

Shows contents of file test.txt

CharacterStream: Java uses Unicode conventions for character storage (see this for more information). Data can be read and written automatically character by character using the character stream. Despite having numerous classes, the most widely used ones are FileReader and FileWriter. Character streams called FileReader and FileWriter are used to write to the destination and read from the source, respectively. The list of different CharacterStream Classes is as follows:

Stream class	Description
BufferedReader	It is used to handle buffered input stream.
FileReader	This is an input stream that reads from file.
InputStreamReader	This input stream is used to translate byte to character.
OutputStreamReader	This output stream is used to translate character to byte.
Reader	This is an abstract class that define character stream input.
PrintWriter	This contains the most used print() and println() method
Writer	This is an abstract class that define character stream output.
BufferedWriter	This is used to handle buffered output stream.
FileWriter	This is used to output stream that writes to file.

```
// Java Program illustrating that
// we can read a file in a human-readable
// format using FileReader
// Accessing FileReader, FileWriter,
```

Notes

Notes

```
// and IOException
import java.io.*;
public class GfG {
    public static void main(
    String[] args) throws IOException
{
    FileReader sourceStream = null;
    try {
    sourceStream = new FileReader("test.txt");
    // Reading sourcefile and
    // writing content to target file
    // character by character.
    int temp;
    while (( temp = sourceStream.read()) != -1)
    System.out.println((char)temp);
}
    finally {
    // Closing stream as no longer in use
    if (sourceStream != null)
    sourceStream.close();
    }
}
}
```

4.2.2 Working with Strings

Working with strings in Java involves various operations such as creating, manipulating, and comparing strings.

1. StartsWith() and Endswith() Method

```
class StrHandling1{
public static void main(String args[]){
String str="Rabindranath";
System.out.println(str.startsWith("Rabin"));
System.out.println(str.endsWith("th"));
 }
 }
```

Output:

true

true

2. Charat() Method

The string charAt() method returns a character at specified index.

```
class StrHandling2{
public static void main(String args[]){
String str=”Rabindranath”;
System.out.println(str.charAt(0));
System.out.println(str.charAt(3));
}
}
```

Output:

R

i

3. toUpperCase() and toLowerCase() Method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
class StrHandling3{
public static void main(String args[]){
String str=”Rabindranath”;
System.out.println(str);
System.out.println(str.toUpperCase());
System.out.println(str.toLowerCase());
}
}
```

Output:

Rabindranath

RABINDRANATH

rabindranath

4. Length() Method

The string length() method returns length of the string.

```
class StrHandling4{
public static void main(String args[]){
String str=”Rabindranath”;
System.out.println(str.length());
}
}
```

Notes

Notes

Output:

12

5. Replace() Method

The replace() method replaces all occurrence of first string with second string.

```
class StrHandling5{
public static void main(String args[]){
String str=”Sri Sri Ravishankar is a spiritual leader”;
String str1=str.replace(“Ravishankar”,”Guruji”);
System.out.println(str1);
}
}
```

Output:

Sri Sri Guruji is a spiritual leader.

Trim() Method

The string trim() method eliminates white spaces before and after string.

```
class StrHandling6{
public static void main(String args[]){
String str=”  Ravishankar  “;
System.out.println(str);
System.out.println(str.trim());
}
}
```

Output:

Ravishankar

Ravishankar

In above example you can see in first output has been printed with spaces before and after the string. When we are using trim() method then all the spaces before and after are getting removed.

Substring in Java

A subset of string is called substring. When working with substring you have to remember that startIndex is inclusive and endIndex is exclusive. You can get substring from the given string object by one of the two methods:

public String substring(int startIndex): This method returns new String object containing the substring of the given string from specified startIndex (inclusive)

public String substring(int startIndex, int endIndex): This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

Example: SubstringExample.java

Notes

```
Public class SubstringExample{
Public static void main(String args[]){
String s=”RabindranathTagore”;
System.out.println(s.substring(12));
System.out.println(s.substring(0,12));
}
}
```

Output:

Tagore

Rabindranath

4.2.3 String Handling: Part 1

Strings are sequence of characters in Java programming language. Strings are treated as objects in Java. The String class is used to create and manipulate strings in Java programming language. String objects are immutable which means unchangeable once created.

Example:

```
char[] ch={‘o’,’b’,’j’,’e’,’c’,’t’,’o’,’r’,’i’,’e’,’n’,’t’,’e’,’d’};
String s=new String(ch);
```

is same as:

```
String s=”objectoriented”;
```

Java String class provides many methods to perform operations on Strings like compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

There are two ways by which you can create String object.

1. By String Literal

Each time when we create a string literal, first the JVM checks the “constant string pool”. If the string is there in the pool, a reference to the pool is returned. If the string doesn’t exist in the pool, a new string instance is created and put in the pool.

Example:

```
String str = “Java is an object oriented programming”;
```

2. By “New” Keyword

JVM will create a new string object in heap memory when you use “new” keyword.

Example:

```
public class StringExample
{
public static void main(String args[]) {
String s1=”java”;     //creating string by java string literal
```

Notes

```
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);              //converting char array to string
String s3=new String("Java Programming");   //creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}
```

Output:

```
java
strings
Java Programming
```

In the above program we have used String class, System class but we did not import the lang package. Other than lang package, if you wish to use other classes resides in other packages then you have to import those built-in package.

4.2.4 String Handling: Part 2

String Concatenation in Java

If you want to combine two or more strings in java then you can do it by string concatenation which adds two or more stings and forms a new string.

There are two ways to concat string in java

- By + operator
- By concat() method

String Concatenation by + (string concatenation) operator

The + operator is used to add strings.

Example: StringConcat.java

```
class StringConcat{
public static void main(String args[]){
String str="Ratan"+" Tata";
System.out.println(str);
 }
 }
```

Output:

```
Ratan Tata
```

The string concatenation operator can concat all primitive values. String concatenation operator produces a new string by appending the second string at the end of the first string.

String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string.

Notes

Example: StringConcat2.java

```
class StringConcat2{
public static void main(String args[]){
String s1=”Sri “;
String s2=”Ravishankar”;   String s3=s1.concat(s2);
 System.out.println(s3);
 }
 }
```

Output:

Sri Ravishankar

Example of concat()

```
// Java program to Illustrate Working of concat() method
// in strings where we are sequentially
// adding multiple strings as we need
// Main class
public class abc {
    // Main driver method
    public static void main(String args[])
    {
        // Custom string 1
        String str1 = “Computer-”;
        // Custom string 2
        String str2 = “-Science”;
        // Combining above strings by
        // passing one string as an argument
        String str3 = str1.concat(str2);
        // Print and display temporary combined string
        System.out.println(str3);
        String str4 = “-Portal”;
        String str5 = str3.concat(str4);
        System.out.println(str5);
    }
}
```

Output

Computer--Science

Computer--Science-Portal

Notes

String Comparison

We can compare string in java on the basis of content and reference.

It is used in authentication (equals() method), sorting (compareTo() method), reference matching (== operator) etc.

1. By equals() method
2. By = = operator
3. By compareTo() method

1. String Compare By Equals() Method

The equals() method checks the values of strings for comparing the equality. String class provides two methods: a) equals() and equalsIgnoreCase()

Example: StringCompareExample.java (Comparing strings with equals() method)

```
class StringCompareExample{
public static void main(String args[]){
String s1=”SubhasChandra”;
String s2=”SubhasChandra”;
String s3=new String(“SubhasChandra”);
String s4=”Bose”;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
}
}
```

Output:

```
true
true
false
```

In above example you can see that though we have created String s1 and s3 in different manner still when we are comparing s1 and s3, it is giving result as true because the value of s1 and s3 are same.

Example: StringComparison.java (Comparing strings with equalsIgnoreCase() method)

```
class StringComparison {
public static void main(String args[]){
String s1=”SubhasChandra”;
String s2=”SUBHASCHANDRA”;
System.out.println(s1.equals(s2));
System.out.println(s1.equalsIgnoreCase(s2));
```

Notes

```
}
}
```

Output:

```
false
true
```

In the above example you can see that both s1 and s2 are same name but case is different, so when equals() method will compare then it will give false result as values are different but when you are using equalsIgnoreCase() method will ignore the case of two strings and compare it and it will give true result as both string are containing same name if you ignore the case.

2. String Comparison By == Operator

When we will use = = operator then it will not compare the value but it will compare references.

Example: StrExample.java

```
class StrExample{
public static void main(String args[]){
String s2=”Birendra”;
String s3=new String(“Birendra”);
System.out.println(s1==s2);
System.out.println(s1==s3);
}
}
```

Output:

```
true
false
```

In the above example you can see s1==s2 is producing true result as both strings are referring to the same instance but as the reference of s1 and s3 are different hence s1==s3 is giving result as false.

3. String Compare By Compareto() Method

The compareTo() method compares values lexicographically and returns an integer value which tells whether first string is less than or equal to or greater than second string.

For better understanding let us take two Strings s1 and s2 .

If s1 == s2 then the method will produce 0

If s1 > s2 then the method will produce positive value

If s1 < s2 then the method will produce negative value

Example: StrCompare.java

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

Notes

```
String s1=" Amitava";
String s2=" Amitava";
String s3="Bachan";
System.out.println("s1.compareTo(s2) resulting:      "+ s1.compareTo(s2));
System.out.println("s1.compareTo(s2) resulting:      "+ s1.compareTo(s3));
System.out.println("s1.compareTo(s2) resulting:      "+ s3.compareTo(s1));
}
}
```

Output:

```
s1.compareTo(s2) resulting: 0
s1.compareTo(s3) resulting: -1
s3.compareTo(s1) resulting: 1
```

In the above program you can notice that as s1 and s2 are same hence when comparing it is giving result as 0. As s1>s3 hence it is giving result as positive integer and as s3

4.3 GUI

Java's GUI tools, such as AWT (Abstract Window Toolkit) and Swing, are commonly used for developing graphical user interfaces (GUIs).

4.3.1 Applets: Part 1

A Java applet is a special kind of Java program that is embedded in webpage and runs in web browser. You can download applet from the internet and run. An applet must be a child class of the java.applet.Applet class. The Applet class provides the standard interface between the applet and the browser environment.

There are some important differences between an applet and a standalone Java application, including the following –

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file

Notes

Advantage of Applet

- It is secured
- It works at client side so less response time.
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

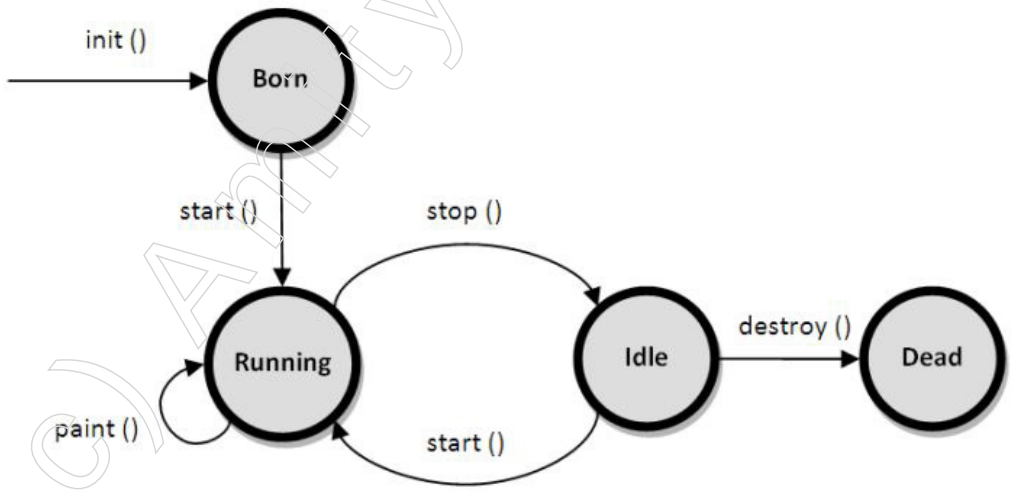
Drawback of Applet

- To execute applet, plugin is required at client browser.
- If we are running an applet from a provider who is not trustworthy than security is important.
- Applet itself cannot run or modify any application on the local system.
- Applets has no access to client-side resources such as files , OS etc.
- Applets can have special privileges. They have to be tagged as trusted applets and they must be registered to APS (Applet Security Manager).
- Applet has little restriction when it comes to communication. It can communicate only with the machine from which it was loaded.
- Applet cannot work with native methods.
- Applet can only extract information about client-machine is its name, java version, OS, version etc.
- Applets tend to be slow on execution because all the classes and resources which it needs have to be transported over the network.

4.3.2 Applets Lifecycle

There are four main steps in applet's lifecycle.

- Applet is started.
- Applet is painted.
- Applet is stopped.
- Applet is destroyed



Source: https://www.shahucollegelatur.org.in/Department/Studymaterial/sci/compsci/Applet%20in%20Java-converted.pdf

Notes

4.3.3 Creating and Applet

Java.applet.Applet Class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. public void init(): To initialize and Applet, init() method is invoked. This method is invoked only once.
2. public void start(): To start an Applet, start() method is invoked.
3. public void stop(): To make an Applet stop you have to use stop() method.
4. public void destroy(): To destroy an Applet, use destroy().This method is also invoked only once.

Java.awt.component Class

In life cycle of applet, the Component class provides one method.

1. public void paint(Graphics g): To paint an Applet, paint() method is used. It is used to draw square, rectangle etc and provides Graphics class object which helps to draw.

Steps to run an Applet

There are two ways to run an applet

1. By html file.
2. By appletViewer tool (for testing purpose).

Creating an executable applet:

Steps: 1. First create child class of Applet class

Steps: 2. Implement init() method and paint() method of Applet class

Execution of Applet by Appletviewer Tool

We have to put applet tag in comment while creating an applet in order to execute applet by appletviewer tool.

Example: Draw.java

```
import  java.applet.*;
import  java.awt.*;
public class Draw extends Applet{
public void paint(Graphics g){
g.drawString("Welcome to applet class", 50, 50);
}
}
/*<applet code = "Draw.class" width = "300" height = "200">
</applet>*/
```

To compile your file you have to write javac Draw.java

After compilation you have to run it by appletviewer First.java.

Here, Html file is not required.

Notes

Example: FirstApplet.java

```
import    java.applet.*;
import    java.awt.*;
public class FirstApplet extends Applet{
public void paint(Graphics g){
g.setColour(Colour.red);
g.drawString("Welcome", 100,100);
g.drawLine(20,30,20,300);
g.setColour(Colour.blue);
g.drawRect(70,100,30,30);
g.fillRect(70,30,30,30);
}
}
/*<applet code        ="FirstApplet.class" width = "300" height = "300">
</applet>     */
```

In the above example, we have imported applet class and awt class in order to run the program. AWT (Abstract Windowing Toolkit) is an API to develop GUI or window based application. It is to be noted that for applet programming, the upper left corner of screen has coordinate (0,0). X coordinate increases from left to right and y coordinator increases from top to bottom of the screen.

4.3.4 Applet with HTML file

To create an applet and execute by html file, you have to create an html file and place the applet code in html file. After creation of HTML file you have to click the html file. You must make class as public because its object is created by Java Plugin software that resides on the browser.

Example: MyApplet.java

```
import java.applet.Applet;
import java.awt.Graphics;
public class MyApplet extends Applet{
public void paint(Graphics g){
g.drawString("I love Java.", 250,250);
}
}
```

myapplet.html

```
<html>
body>
<applet code=" MyApplet.class" width="350" height="350">
</applet>
```

Notes

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

4.3.5 Using Control Loops in Applet

We'll demonstrate loops using a straightforward example:

Take a look at the applet first.

For those who are bored or want to play a board game, the applet simply draws an 8 × 8 grid.

Here's the program:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class GridDrawingApplet extends Applet {
public void paint (Graphics g)
{
// Draw the vertical lines:
g.drawLine (0,0, 0,240);
g.drawLine (30,0, 30,240);
g.drawLine (60,0, 60,240);
g.drawLine (90,0, 90,240);
g.drawLine (120,0, 120,240);
g.drawLine (150,0, 150,240);
g.drawLine (180,0, 180,240);
g.drawLine (210,0, 210,240);
g.drawLine (240,0, 240,240);
// Draw the horizontal lines:
g.drawLine (0,0, 240,0);
g.drawLine (0,30, 240,30);
g.drawLine (0,60, 240,60);
g.drawLine (0,90, 240,90);
g.drawLine (0,120, 240,120);
g.drawLine (0,150, 240,150);
g.drawLine (0,180, 240,180);
g.drawLine (0,210, 240,210);
g.drawLine (0,240, 240,240);
}
}
```

Notes

We will now use a loop to rewrite the code mentioned above:

Observe the repetition of the motif. We would need to input hundreds of lines of code if we had to draw a grid that was 100 by 100.

A loop is a computer construct used to help repeat an action with a modification made to it each time.

Writers can write loops in a variety of ways. This one is this:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class GridDrawingAppletForLoop extends Applet {
public void paint (Graphics g)
{
// Draw the vertical lines:
for (int x=0; x<=240; x+=30) {
g.drawLine (x,0, x,240);
}
// Draw the horizontal lines:
for (int y=0; y<=240; y+=30) {
g.drawLine (0,y, 240,y);
}
 }
}
```

Note:

Take note of how much shorter the show is. If we drew a grid that was 100 by 100, the programme length would remain the same.

The word “for” is reserved.

The for statement’s structure is as follows:

```
for ( starting condition;  continue-condition; change ) {
body of loop - a bunch of statements
}
```

Therefore, “for x starting at zero, as long as x <= 240, increment x by 30 at each iteration of the loop” is how you “read” the first loop.

Take note of the operator +=’s usage. The initial loop is comparable to:

```
// Draw the vertical lines:
for (int x=0; x<=240; x=x+30) {
g.drawLine (x,0, x,240);
}
```

Notes

To provide more clarity, we could add some whitespace: // Draw the vertical lines:

```
for (int x = 0;  x <= 240;  x = x + 30) {
 g.drawLine (x,0, x,240);
}
```

4.3.6 The Graphics Class: Part 1

Every applet is an extension of the java.applet.Applet class. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following :

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet.

Displaying Graphics in Applet

For graphics programming, java.awt.Graphics class provides many methods.

Methods of Graphics class:

1. public abstract void drawString(String str, int x, int y): This method is used to draw string.
2. public void drawRect(int x, int y, int width, int height): This method is used to draws rectangle with the specified width and height.
3. public abstract void fillRect(int x, int y, int width, int height): This method is used to fill rectangle with the default colour and specified width and height.
4. public abstract void drawOval(int x, int y, int width, int height): This method is used to draw oval with the specified width and height.
5. public abstract void fillOval(int x, int y, int width, int height): This method is used to fill oval with the default colour and specified width and height.
6. public abstract void drawLine(int x1, int y1, int x2, int y2): This method is used to draw line between the points(x1, y1) and (x2, y2).
7. public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer): This method is used to draw the specified image.
8. public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle): This method is used to draw a circular or elliptical arc.
9. public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle): This method is used to fill a circular or elliptical arc.
10. public abstract void setColour(Colour c): This method is used to set the graphics current colour to the specified colour.

11. public abstract void setFont(Font font): This method is used to set the graphics current font to the specified font.

Notes

4.3.7 The Graphics Class: Part 2

Inside paint() or update() method

It is passed as an argument to paint and update methods and therefore can be accessed inside these methods. paint() and update() methods are present in the Component class and thus can be overridden for the component to be painted.

void paint(Graphics g)

void update(Graphics g)

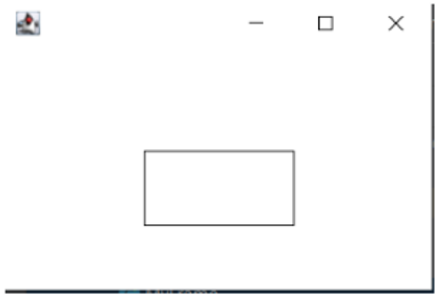
Example:

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;


public class MyFrame extends Frame {
    public MyFrame()
    {
        setVisible(true);
        setSize(300, 200);
        addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
     });
    }
    public void paint(Graphics g)
    {
    g.drawRect(100, 100, 100, 50);
    }
    public static void main(String[] args)
    {
    new MyFrame();
    }
}
```

Output

Notes



getGraphics() method

Since this function is part of the Component class, it may be used to obtain the Graphics object for any component by calling it.

public Graphics getGraphics(): Establishes the component's graphics context. In the event that the component is not currently displayable, this method will return null.

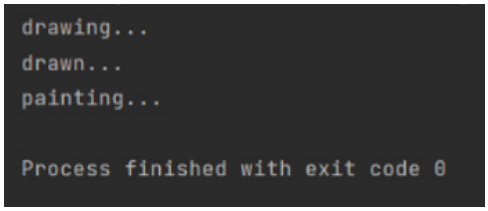
Example:

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyFrame extends Frame {
    public MyFrame()
    {
     setVisible(true);
     setSize(300, 200);
     addWindowListener(new WindowAdapter() {
     @Override
     public void windowClosing(WindowEvent e)
    {
     System.exit(0);
    }
     });
    }
    public void paint(Graphics g)
    {
     System.out.println("painting...");
    }
    public static void main(String[] args)
    {
     MyFrame f = new MyFrame();
     Graphics g = f.getGraphics();
     System.out.println("drawing...");
     g.drawRect(100, 100, 10, 10);
     System.out.println("drawn...");
```

Notes

```
    }
}
```

Output



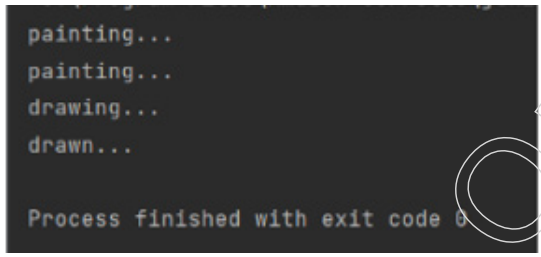
This code will not render the graphics drawn via getGraphics() because paint() is called after the draw().

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyFrame extends Frame {
    public MyFrame()
{
    setVisible(true);
    setSize(300, 200);
    addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
});
}
`public void paint(Graphics g)
{
    System.out.println("painting...");
}
public static void main(String[] args)
{
    MyFrame f = new MyFrame();
    Graphics g = f.getGraphics();
    try {
    Thread.sleep(1000);
}
     catch (Exception e) {
}
     System.out.println("drawing...");
```

Notes

```
        g.drawRect(100, 100, 100, 50);
        System.out.println("drawn...");
}
}
```

Output



The above code will display the rectangle as paint() is called before draw().

Drawing shapes using Graphics Object

Line

The following method of Graphics class is used to draw the line:

void drawLine(int startX, int startY, int endX, int endY)

Example:

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyFrame extends Frame {
public MyFrame()
{
    setVisible(true);
    setSize(300, 300);
    addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
});
}
    public void paint(Graphics g)
{
    g.drawLine(100, 100, 200, 200);
}
    public static void main(String[] args)
{
    new MyFrame();
```

```
}
}
```

Notes

Output

Rectangle

The following methods of Graphics class are used to draw rectangles:

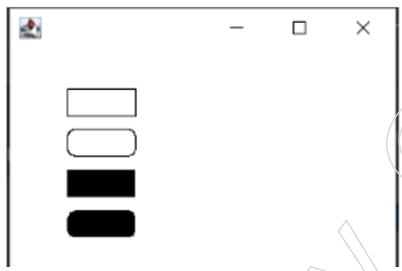
```
void drawRect(int x, int y, int width, int height)
void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
void fillRect(int x, int y, int width, int height)
void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class MyFrame extends Frame {
public MyFrame()
    {
setVisible(true);
setSize(300, 300);
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e)
    {
     System.exit(0);
    }
    });
   }
    public void paint(Graphics g)
    {
     g.drawRect(50, 60, 50, 20);
     g.drawRoundRect(50, 90, 50, 20, 10, 10);
     g.fillRect(50, 120, 50, 20);
     g.fillRoundRect(50, 150, 50, 20, 10, 10);
    }
```

Notes

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

Output



Oval and Circle

The following methods are used to draw ovals and circles :

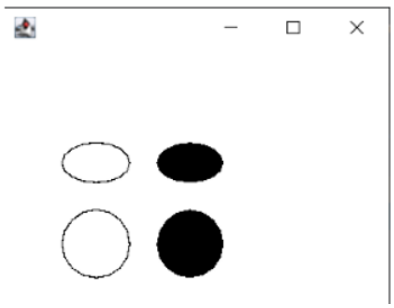
```
 void drawOval(int x, int y, int width, int height)
void fillOval(int x, int y, int width, int height)
```

Example:

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyFrame extends Frame {
public MyFrame()
{
    setVisible(true);
     setSize(300, 300);
     addWindowListener(new WindowAdapter() {
     @Override
     public void windowClosing(WindowEvent e)
{
     System.exit(0);
}
     });
    }
    public void paint(Graphics g)
    {
     g.drawOval(50, 100, 50, 30);
     g.fillOval(120, 100, 50, 30);
     g.drawOval(50, 150, 50, 50);
     g.fillOval(120, 150, 50, 50);
    }
```

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

Output



Example: GraphicsExample.java

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsExample extends Applet{
public void paint(Graphics g){
g.setColour(Colour.red);
g.drawString("Welcome to applet",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
g.setColour(Colour.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}
}
```

Summary

- In Java, a package is a namespace that organizes a group of related classes and interfaces, helping to manage and modularize large code bases. It provides access protection and namespace management, allowing classes to be grouped logically and making it easier to locate and use them. Packages prevent naming conflicts and can be hierarchical, enabling a structured way to organize code. The import statement is used to access classes and interfaces from other packages, promoting code reuse and better organization.
- The Applet class extends the AWT Panel class, which extends the AWT Container class, which extends the AWT Component class. From Component, an applet inherits the ability to draw and handle events.

Notes

Notes

- The java.io package is essential for performing basic and advanced input and output operations in Java. It includes classes for handling files, byte streams, character streams, and object serialization. Proper use of these classes enables efficient and effective data processing and manipulation in Java applications.
- In Java, strings are sequences of characters and are represented by the String class, which is immutable, meaning once a string object is created, its value cannot be changed. The String class provides numerous methods for string manipulation, such as concatenation, comparison, substring extraction, and searching. Strings can be created using string literals or the new keyword. Because strings are so fundamental, Java offers special support for them, including overloaded operators for concatenation (+) and a string pool to optimize memory usage and performance. The immutability of strings ensures security, thread-safety, and efficient memory management.
- Applets in Java are small Java programs that can be embedded in a web page and run in a web browser. They are part of the java.applet package and are typically used to provide interactive features to web applications that cannot be provided by HTML alone. An applet must be a subclass of the java.applet.Applet class or its more feature-rich successor, javax.swing.JApplet.

Glossary

- HTML: Hyper Text Markup Language
- API: Application Programming Interface
- GUI: Graphical User Interface
- I/O: Input and Output
- JDK: Java Development Toolkit
- java.sql: Provide the classes needed to handle and retrieve data from a database. This package contains classes such as PreparedStatement, ResultSet, Statement, Connection, DriverManager, and so on.
- java.lang: includes interfaces and classes that are essential to the Java programming language's architecture. This package contains classes such as String, StringBuffer, System, Math, and Integer, among others.
- java.util: It includes the random number generator classes, properties, the collections framework, and various support classes for internationalisation. This package contains classes such as Calendar, Date, Time Zone, LinkedList, HashMap, and ArrayList.

Check Your Understanding

1. What is the primary purpose of using packages in Java?
   a) To organize classes and interfaces into a single file
   b) To improve code readability by adding comments
   c) To encapsulate the implementation details and provide a namespace
   d) To reduce the size of the compiled code
2. Which keyword is used to import a specific class from a package in Java?
   a) import
   b) include
   c) require
   d) load

Notes

3. What is the default package in Java?
   a) java.util
   b) java.lang
   c) java.default
   d) java.main
4. Which of the following statements is true about the java.lang package?
   a) It must be imported explicitly in every Java program.
   b) Classes in this package are not accessible by default and require import statements.
   c) It is automatically imported into every Java program.
   d) It contains only advanced features and is not suitable for basic programming.
5. Which package in Java is commonly used for basic I/O operations, such as reading from or writing to files?
   a) java.util
   b) java.nio
   c) java.io
   d) java.net

Exercise

1. Define each term of System.out.println()?
2. How to reverse a string in java?
3. How do you remove all white spaces from a string in java?
4. Write a java program to find the duplicate words and their number of occurrences in a string?
5. Write a java program to count the number of words in a string?

Learning Activities

1. Write a program to show how to add applet to HTML?
2. Write a program to show how to display graphics in applet?

Check Your Understanding- Answers

1. c) 2. a) 3. b) 4. c
5. c)

Further Readings and Bibliography

1. R. Nageswara Rao. (2016). Core Java: An Integrated Approach. Dreamtech Press, 720 pages.
2. E. Balagurusamy. (2023). Programming with Java. 7th Edition. McGrawHill Publications, 592 pages.
3. Barry A. Burd. (2017). Beginning Programming with Java for Dummies. 5th Edition. Wiley Publications.
4. Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action. Dreamtech Press.

Notes

Module - V: Event Driven Programming and Database Programming Using JDBC

Learning Objectives:

At the end of this module, you will be able to:

- Define Abstract Window Toolkit
- Define AWT components
- Discuss Event model
- Define Event class
- Define Event handling processes
- Discuss about AWT controls
- Define different Layout Managers
- Define Swing classes

Introduction

Event-driven programming is a programming paradigm that focuses on the occurrence of events, such as user actions (e.g., mouse clicks, keyboard input) or messages from other parts of a program. In an event-driven model, the flow of the program is determined by events, and the program responds to these events through event handlers or callbacks.

5.1 Overview of Event Driven Programming

Components are displayed in frame which is a container, but these components are static i.e. they can't respond. Suppose you have created a login button and if you press it now, it will not act. To make it dynamic, we need to add event. So, we can tell Buttons, Checkboxes are not able to perform any actions. If you click a button, automatically an event will be raised. to handle events we require listener. Responsibility of listener is to listen to the event generated by GUI components, handle that event and sending results back to GUI program.

5.1.1 Abstract Window Toolkit: Part 1

The Abstract Window Toolkit (AWT) is an Application Program Interface (API) to support Graphical User Interface (GUI) or we can say it is a framework (collection of classes and interfaces) to prepare desktop application. Features like Graphical tools including shape, colour, and font classes. Components of AWT are platform-dependent i.e. depending on the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The AWT provides two levels of APIs:

A general interface between Java and the native system, used for windowing, events, and layout managers. This API is at the core of Java GUI programming and is also used by Swing and Java 2D. It contains:

The interface between the native windowing system and the Java application;

The core of the GUI event subsystem;

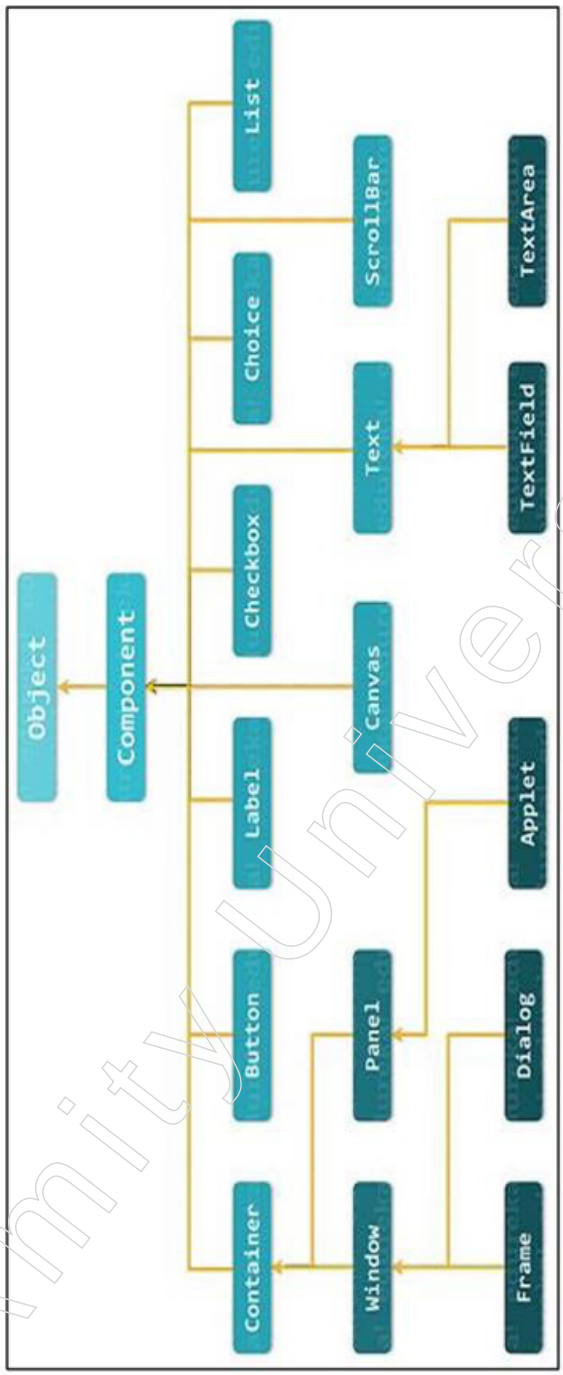
Several layout managers;

Notes

AWT also makes some higher level functionality available to applications, such as:

Access to the system tray on supporting systems;

The ability to launch some desktop applications such as web browsers and email clients from a Java application.



As you can see in above diagram, Component is the parent class of all the GUI controls. It is an abstract class. For the look and feel of interface, component class instance is responsible.

5.1.2 Abstract Window Toolkit: Part 2

1. Container: In Java AWT, Container class is a GUI component that is used to manage other GUI components such as Frame, Window, Panel etc. For AWT, java.awt package

Notes

is there and Container is a subclass of java.awt.Component and is responsible for components being added. There are four types of containers provided by AWT in Java.

Types of Containers

a) Window: It is Window class's instance which does not have border or title. It is used for creating a top-level window.

b) Frame: Frame is a container and able to manage some other GUI components like label, text field etc. It is the most widely used container for developing AWT applications. You can create a Java AWT Frame in two ways:

- By instantiating Frame class
- By extending Frame class

c) Dialog: Dialog class is also a subclass of Window. It contains border, title etc.

d) Panel: Panel is the concrete subclass of Container. Panel doesn't contain title bar, birder or menu bar.

2. Button: To create a labeled button, you can use Button class. When user will click a button, it will trigger a certain programmed action. Button class has two constructors.
3. Text Field: To create a single-line text box so that user can enter texts, you can use TextField. The TextField class has three constructors.
4. Label: To create a descriptive text string to be visible on GUI, you can use Label class. An AWT Label object is a component for placing text in a container. Label class has three constructors.
5. Canvas: To draw in an application or receive inputs created by the user, you can use Canvas class. It is a rectangular area.
6. Choice: To create a pop-up menu of different choices, you can use Choice class. The selected choice is shown on the top of the given menu.
7. Scroll Bar: To add horizontal and vertical scrollbar in the GUI, Scrollbar class object is used It enables user to see the more number of rows and columns.
8. List: To enable user to choose from a list of text items, List class is used.
9. CheckBox: To create a checkbox which can accept either of the two option, you can use Checkbox class. It has two state options; true and false. At any point in time, it can have either of the two.

5.1.3 Introduction to Swing Classes and Controls

Swing is a lightweight toolkit having huge variety of widgets for making optimized window based applications. It is a part of the Java Foundation Classes (JFC) and is build on top of the AWT API. It is platform independent. To build applications in Swing is easy as we already have GUI components like buttons, checkboxes etc and very much helpful as you need not to start from the scratch.

Features of Swing Class

- Pluggable look and feel.
- Uses MVC architecture.
- Lightweight Components
- Platform Independent
- Advanced features such as JTable, JTabbedPane, JScollPane, etc.

Notes

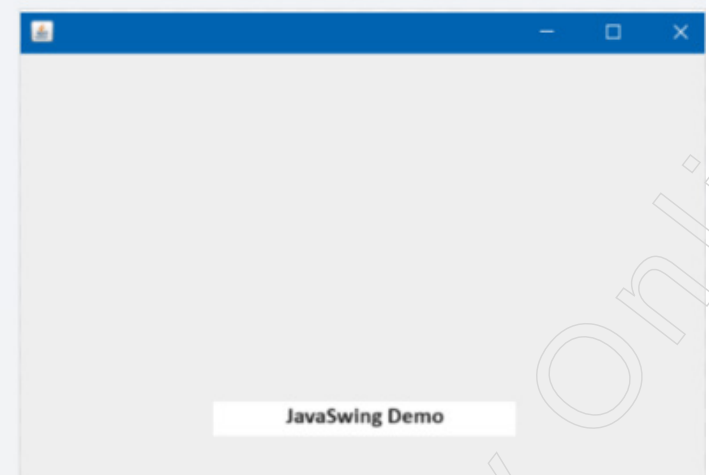
- Because Java is a platform-independent language that may run on any client machine, an application's GUI built using Swing components is unaffected by the GUI's ownership and delivery by a platform-specific operating system.
- Lightweight Elements: Its AWT-supported lightweight component development is available starting with JDK 1.1. A component cannot be considered lightweight if it depends on any system classes that aren't Java-based (O/s). Java's look and feel classes support the unique perspective of Swing components.
- Pluggable Look and Feel: This feature allows the user to change the Swing components' appearance and feel without having to restart an application. Wherever the programme runs, the Swing library supports components that have the same appearance and feel on all platforms. The Swing framework offers an API that offers significant freedom in customising the appearance and feel of an application's graphical user interface.
- quite configurable — Since the external appearance of Swing controls is independent of internal representation, customisation is quite simple.
- Rich controls: Tree TabbedPane, sliders, colorpickers, and table controls are just a few of the sophisticated controls that Swing offers.

Example of Java Swing Programs

```
// Java program using label (swing)
// to display the message "JavaSwing Demo"
import java.io.*;
import javax.swing.*;
// Main class
class abc {
    // Main driver method
    public static void main(String[] args)
    {
     // Creating instance of JFrame
     JFrame frame = new JFrame();
     // Creating instance of JButton
     JButton button = new JButton(" JavaSwing Demo ");
     // x axis, y axis, width, height
     button.setBounds(150, 200, 220, 50);
     // adding button in JFrame
     frame.add(button);
     // 400 width and 500 height
     frame.setSize(500, 600);
     // using no layout managers
     frame.setLayout(null);
     // making the frame visible
     frame.setVisible(true);
    }
}
```

Notes

Output



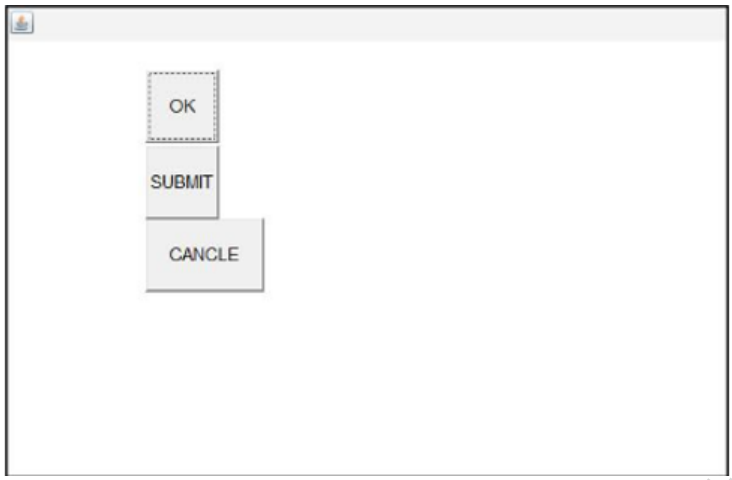
Example 2: To create three buttons with caption OK, SUBMIT, CANCEL.

```
// Java program to create three buttons
// with caption OK, SUBMIT, CANCEL
import java.awt.*;
class button {
    button()
    {
     Frame f = new Frame();
     // Button 1 created
     // OK button
     Button b1 = new Button(“OK”);
     b1.setBounds(100, 50, 50, 50);
     f.add(b1);
     // Button 2 created
     // Submit button
     Button b2 = new Button(“SUBMIT”);
     b2.setBounds(100, 101, 50, 50);
     f.add(b2);
     // Button 3 created
     // Cancel button
     Button b3 = new Button(“CANCEL”);
     b3.setBounds(100, 150, 80, 50);
     f.add(b3);
     f.setSize(500, 500);
     f.setLayout(null);
     f.setVisible(true);
    }
    public static void main(String a[]) { new button(); }
```

}

Output

Notes



Example 3: Program to Add Checkbox in the Frame

```
// Java Swing Program to Add Checkbox
// in the Frame
import java.awt.*;
// Driver Class
class Lan {
    // Main Function
    Lan()
    {
     // Frame Created
     Frame f = new Frame();
     Label l1 = new Label("Select known Languages");
     l1.setBounds(100, 50, 120, 80);
     f.add(l1);
     // CheckBox created
     Checkbox c2 = new Checkbox("Hindi");
     c2.setBounds(100, 150, 50, 50);
     f.add(c2);
     // CheckBox created
     Checkbox c3 = new Checkbox("English");
     c3.setBounds(100, 200, 80, 50);
     f.add(c3);
     // CheckBox created
     Checkbox c4 = new Checkbox("marathi");
     c4.setBounds(100, 250, 80, 50);
     f.add(c4);
     f.setSize(500, 500);
```

Notes

```
        f.setLayout(null);
        f.setVisible(true);
      }
      public static void main(String ar[]) { new Lan(); }
}
```

Output

Container Class

A container class is a class which has other components. For building GUI applications at least one container class is necessary.

Following are the three types of container classes:

- Panel: It is used to organize components on to a window
- Frame: A fully functioning window with icons and titles
- Dialog: It is like a pop up window but not fully functional like the frame

JFrame: A frame is an instance of JFrame. Frame is a window that can have title, border, menu, text fields , buttons etc. A Swing application must have a frame to have the components added to it.

JPanel: A panel is an instance of JPanel. Frame can have more than one panels and each panel can have several components. Those are parts of Frame. Panels are useful for grouping components and placing them to appropriate locations in a frame.

JLabel: A label is an instance of JLabel class. If you want to display a string or an image on a frame, you can do it by using labels.

JButton: A button is an instance of JButton class.

JTextField: This is used for taking user inputs from the text boxes where user enters the data.

JPasswordField: Similar to text fields but the entered data gets hidden and displayed as dots on GUI.

Notes

5.1.4 Advantages of Swings over AWT

AWT and Swing are both part of a group of Java class libraries called the Java Foundation Classes (JFC). The Abstract Windowing Toolkit (AWT) is the original GUI toolkit shipped with the Java Development Kit (JDK). The AWT provides a basic set of graphical interface components similar to those available with HTML forms. Swing is the latest GUI toolkit, and provides a richer set of interface components than the AWT. In addition, Swing components offer the following advantages over AWT components:

- The behaviour and appearance of Swing components is consistent across platforms, whereas AWT components differ from platform to platform
- Swing components can change their appearance based on the current "look and feel" library.
- Swing uses a more efficient event model than AWT; therefore, Swing components can run more quickly than their AWT counterparts
- Swing provides "extras" for components, such as: Icons on many components, Decorative borders for components, Tooltips for components
- Swing provides built-in double buffering
- Rich Set of Components: Swing provides a more extensive and versatile set of GUI components compared to AWT. It includes advanced components like tables, trees, tabbed panes, scroll panes, and sliders, allowing developers to create more sophisticated and visually appealing user interfaces.
- Platform Independence: Swing components are implemented entirely in Java, ensuring consistent behavior and appearance across different platforms and operating systems. In contrast, AWT components rely on the underlying native windowing system, which can lead to inconsistencies in look and feel.
- Customization and Look and Feel: More customisation possibilities are provided by Swing for GUI components, giving developers more precise control over elements like colours, fonts, borders, and layout. Furthermore, pluggable look and feel (PLAF) is supported by Swing, allowing programmers to dynamically alter the appearance of their applications.
- Performance and Efficiency: When compared to AWT components, swing components are more effective and lightweight. In order to increase performance and scalability, they employ a model-view-controller (MVC) design, which divides the visual presentation (view) from the underlying data (model) and event processing (controller).
- Advanced Features: Double buffering, transparency, drag-and-drop functionality, keyboard navigation, accessibility capabilities, and integrated support for internationalisation and accessibility are just a few of the sophisticated features that Swing offers that are not present in AWT.
- Concurrency and Thread Safety: Swing has built-in support for concurrency and event dispatching techniques, making it more thread-safe than AWT. By doing this, problems like race situations and deadlocks are avoided and event handling and GUI updates are guaranteed to happen on the event dispatch thread (EDT).

5.1.5 Layout Managers

To keep all the elements in a particular order, we need Layout Mechanism. We use the layout manager to arrange the components inside a container. Below are several layout managers:

Notes

1. Flow layout
2. Border layout
3. Grid layout
4. GridBag layout
5. Card layout

Flow Layout: This layout is able to arrange all GUI component in row manner one after another. To represent this Java has FlowLayout class. In order to use this you have to import java.awt.FlowLayout.

Border Layout: This layout is able to arrange all GUI component along with borders of the container. Java has BorderLayout class. In order to use this you have to import java.awt.BorderLayout. It places components in five places which is top, bottom, left, right and center.

Grid Layout: This layout places the GUI components in the form of grids i.e. in row and column manner. Predefine class java.awt.GridLayout is there.

GridBag Layout: This layout is almost same as GridLayout. Predefine class java.awt.GridBagLayout is there.

Advantages of this layout are:

1. Components can be of different size
2. Empty grids are allowed between components

Card Layout:

This layout is able to arrange GUI components in the form of layers along with borders of the container. Java has BorderLayout class. In order to use this you have to import java.awt.BorderLayout.

5.2 Event Handling

Components are displayed in frame, which is a container, but these components are static i.e. they can't respond. Suppose you have created a login button and if you press it now, it will not act. To make it dynamic, we need to add event. So, we can tell Buttons, Checkboxes are not able to perform any actions. If you click a button, automatically an event will be raised. to handle events we require listener. Responsibility of listener is to listen to the event generated by GUI components, handle that event and sending results back to GUI program.

5.2.1 Event Handling in Java

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Sources of Events

The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

Notes

Listener - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

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

Steps to Perform Event Handling:

1. Create container class and declare a 0-arg constructor, inside the constructor declare a GUI component.

   Example:

   ```
   class MyFrame extends Frame {
    MyFrame(){
    Button b = new Button("Submit");
    }
    }
   ```

2. Select a listener and provide Listener implementation class with the implementation of Listener methods.

   ```
    class MyActionListener implements ActionListener {
    public void actionPerformed (ActionEvent ae){
    .....
    }
    }
   ```

   In general, in GUI applications, we will implement Listener interface in the same container class.

   ```
   class MyFrame extends Frame implements ActionListener {
   .....
   }
   ```

3. Attach Listener to GUI component like public void addxxxListener (xxxListener l)

   Note that, xxxListener may be ActionListener, ItemListener and so on.

   Example: b.addActionListener (new MyActionListener());

5.2.2 Event Models and Classes

Java uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

- java.awt => to prepare static component
- java.awt.event => to make static component dynamic

To represent events, classes are provided by AWT and to represent listeners, interfaces are provided by AWT. Java has provided pre defined libraries to perform event

Notes

handling. A number of listeners are available in Java and for each listener, a number of listener methods are available.

GUI component	Event
Button	ActionEvent
Checkbox	ItemEvent
RadioButton	ItemEvent
List	ItemEvent
Choice	ItemEvent
Menu	ActionEvent
ScrollBar	AdjustmentEvent
Window	WindowEvent
Mouse	MouseEvent
Keyboard	KeyEvent
TextField	TextEvent
TextArea	TextEvent

For button, listener is ActionListener. Buttons will raise one event that is ActionEvent. To handle this ActionEvent, we will implement ActionListener as it is an interface. In ActionListener, actionPerformed() method is there which we have to use for putting implementation.

Let us take an example, let us think in a form there are two Text fields which will accept data from user and in 3rd field, it will produce the result once user click on Button. Once data is put in two fields and you click on button then automatically an event will be raised (ActionEvent) and immediately flow of execution will go to correspondint listener (here ActionListener) and inside the listener, event method will be executed.

Hence, in general we can tell that whenever we trigger a GUI component, an event will be raised. The process of raising an event and the process of delegating or bypassing event to the listener in order to handle. So, the total process of handling event is called Event delegation model or event handling.

Event Classes

Event is to change the state of an object. The java.awt.event package provides many event classes and Listener interfaces for event handling. For example dragging mouse of clicking a button etc are events. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Event classes	Listener interfaces
ActionEvent	ActionListerner
MouseEvent	MouseListener
MouseWheelEvent	MouseWheelListener
ItemEvent	ItemListener
KeyEvent	KeyListener
ComponentEvent	ActionListerner
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
TextEvent	TextListener

ContainerEvent	ContainerListener
FocusEvent	FocusListener

Notes

For registering the component with the Listener, many classes provide the registration methods. For example:

- Button (Creates a push button control)
  - ❖ public void addActionListener(ActionListener a){}
- MenuItem (Creates a menu item.)
  - ❖ public void addActionListener (ActionListener a){}
- TextField (Creates a single-line edit control
  - ❖ public void addActionListener (ActionListener a){}
  - ❖ public void addTextListener (TextListener a){}
- TextArea (Creates a multiline edit control)
  - ❖ public void addTextListener (TextListener a){}
- Checkbox(Creates a check box control)
  - ❖ public void addItemListener (ItemListener a){}
- Choice (Creates a pop-up list)
  - ❖ public void addItemListener (ItemListener a){}
- List (creates a list from which a user can choose)
  - ❖ public void addActionListener (ActionListener a){}
  - ❖ public void addItemListener (ItemListener a){}

Event Handling Within Class

```
// Java program to demonstrate the
// event handling within the class
import java.awt.*;
import java.awt.event.*;
class GFGTop extends Frame implements ActionListener {
    TextField textField;
    GFGTop()
    {
     // Component Creation
     textField = new TextField();
     // setBounds method is used to provide
     // position and size of the component
     textField.setBounds(60, 50, 180, 25);
     Button button = new Button("click Here");
     button.setBounds(100, 120, 80, 30);
     // Registering component with listener
     // this refers to current instance
```

Notes

```
        button.addActionListener(this);
        // add Components
        add(textField);
        add(button);
        // set visibility
        setVisible(true);
       }
       // implementing method of actionListener
       public void actionPerformed(ActionEvent e)
       {
        // Setting text to field
        textField.setText("ABC!");
       }
       public static void main(String[] args)
       {
       new GFGTop();
       }
}
```

Output

Event Handling by Other Class

```
// Java program to demonstrate the
// event handling by the other class
import java.awt.*;
import java.awt.event.*;
class GFG1 extends Frame {
      TextField textField;
      GFG2()
      {
       // Component Creation
       textField = new TextField();
```

Notes

```
        // setBounds method is used to provide
        // position and size of component
        textField.setBounds(60, 50, 180, 25);
        Button button = new Button("click Here");
        button.setBounds(100, 120, 80, 30);
        Other other = new Other(this);
        // Registering component with listener
        // Passing other class as reference
        button.addActionListener(other);
        // add Components
        add(textField);
        add(button);
        // set visibility
        setVisible(true);
       }
       public static void main(String[] args)
       {
       new GFG2();
       }
}
```

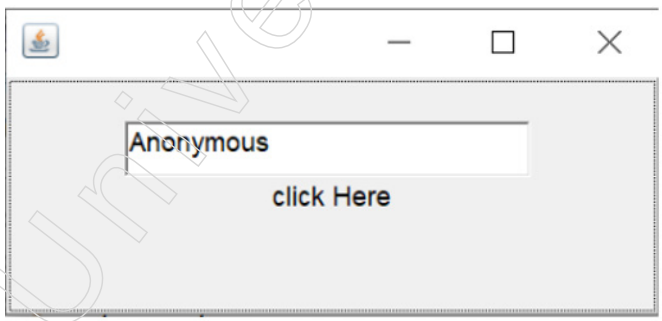
Event Handling By Anonymous Class

```
// Java program to demonstrate the
// event handling by the anonymous class
import java.awt.*;
import java.awt.event.*;
class GFG3 extends Frame {
       TextField textField;
       GFG3()
       {
        // Component Creation
        textField = new TextField();
        // setBounds method is used to provide
        // position and size of component
        textField.setBounds(60, 50, 180, 25);
       Button button = new Button("click Here");
        button.setBounds(100, 120, 80, 30);
        // Registering component with listener anonymously
        button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
```

Notes

```
        // Setting text to field
        textField.setText("Anonymous");
        }
        });
        // add Components
        add(textField);
        add(button);
        //make size viewable
        setSize(300,300);
        // set visibility
        setVisible(true);
       }
       public static void main(String[] args)
       {
        new GFG3();
       }
}
```

Output



5.2.3 Event Listener Interface

GUI component	Event	Listener interfaces
Button	ActionEvent	ActionListerner
Checkbox	ItemEvent	ItemListener
RadioButton	ItemEvent	ItemListener
List	ItemEvent	ItemListener
Choice	ItemEvent	ItemListener
Menu	ActionEvent	ActionListerner
ScrollBar	AdjustmentEvent	AdjustmentListener
Window	WindowEvent	WindowListener
Mouse	MouseEvent	MouseListener
Keyboard	KeyEvent	KeyListener
TextField	TextEvent	TextListener
TextArea	TextEvent	TextListener

Notes

5.2.4 Networking Classes and Interfaces

To execute a program across multiple devices is done by network programming. The devices are all connected to each other using a network. The java.net package contains a collection of classes and interfaces that provide the low-level communication details. Below are listed few important classes and interfaces used is networking.

Java Networking Classes

CacheRequest	CookieHandler
CookieManager	Datagrampacket
InetAddress	ServerCocket
Socket	DatagramSocket
Proxy	URL

Java Networking Interfaces

CookiePolicy	CookieStore
FileNameMap	SocketOption
InetAddress	ServerSocket
SocketImplFactory	ProtocolFamily

5.3 Java Database Connectivity

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. JDBC drivers are used in JDBC API to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver
- Native Driver
- Network Protocol Driver
- Thin Driver

5.3.1 Introduction to TCP\IP

Transmission Control Protocol (TCP) allows reliable communication between two applications. TCP is typically used over the Internet Protocol(IP) and is referred to as TCP/IP. TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP) and Telnet are all examples of applications that require a reliable communication channel. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you can have a jumbled HTML file or some other invalid information or a corrupt zip file.

User Datagram Protocol (UDP) is a connection-less protocol that allows for packets of data to be transmitted between applications. The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection oriented like TCP instead UDP sends independent packets of data (datagrams) from one application to another. The order of delivery is not important and is not guaranteed, and each message is independent of any other.

Socket Programming is the most widely used concept in Networking. Sockets provide the communication mechanism between two computers using TCP. A client program

Notes

attempts to connect the socket to a server by creating a socket on its end . The server creates a socket object on its end of the communication when the connection is made.

After connection is formed, the client and the server can communicate by writing to and reading from the socket. The java.net.Socket class represents a socket, and the java.net. ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

When establishing a TCP connection between two computers using sockets the below steps takes place:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

Communication can occur using I/O streams after the connections are established. Each socket has an InputStream and an OutputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol so data can be sent across both streams at the same time.

ServerSocket Class Methods

The java.net.ServerSocket class is used by server applications to obtain a port and listen for client requests.

Socket Class Methods

The java.net.Socket class represents the socket that both the client and the server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

5.3.2 Introduction to Datagram Programming

Networking in the TCP/IP style offers a serialised, dependable, and predictable stream of packet data. There is a price for this, though. TCP has algorithms for handling congestion control on congested networks and has low expectations regarding packet loss. This results in an ineffective method of data transfer.

A dedicated point-to-point channel exists between clients and servers that communicate over a dependable channel, like a TCP socket. They create a connection, send data over it, and then cut it off to communicate. Every bit of data transmitted across the channel is received in the same sequence that it was sent. The channel guarantees this.

Notes

Applications that communicate over datagrams, on the other hand, send and receive entirely separate information packets. A dedicated point-to-point channel is not required for these clients and servers, nor do they have one. There is no certainty that datagrams will reach their destinations. Nor is the sequence in which they arrive.

Datagram

A datagram is a stand-alone, self-contained message delivered via a network with no guarantees regarding its delivery, time of arrival, or content.

- As a substitute, datagrams are essential.
- Information packets known as datagrams are transmitted between devices. There is no guarantee that the datagram will reach its destination once it has been launched, nor that someone will be present to intercept it.
- It is important to remember that there is no guarantee that the datagram was not harmed during transit or that the sender is still alive and available for a response.

Java uses two classes to implement datagrams on top of the UDP (User Datagram Protocol) protocol:

- The data container is a DatagramPacket object.
- The method used to send and receive DatagramPackets is called a DatagramSocket.

Datagram Socket Class

- In DatagramSocket, four public constructors are defined. Here is how they appear:
- Creates a DatagramSocket tied to any open port on the local machine; DatagramSocket() throws SocketException.
- A DatagramSocket is created and bound to the port given by port when DatagramSocket(int port) produces a SocketException.
- DatagramSocket(int port, InetAddress ipAddress) creates a DatagramSocket tied to the given port and InetAddress; it then throws a SocketException.

DatagramSocket(SocketAddress address) creates a DatagramSocket tied to the given SocketAddress and raises a SocketException.

The concrete class InetSocketAddress implements the abstract class SocketAddress. InetSocketAddress uses a port number to encapsulate an IP address. In the event that a problem arises during socket creation, anyone can throw a socket exception. Many methods are defined by DatagramSocket. The two most crucial ones are receive() and send(), which are displayed here:

- ❖ void send(DatagramPacket packet) throws IOException
- ❖ void receive(DatagramPacket packet) throws IOException

The packet is sent to the port indicated by packet using the send() method. The receive method returns the result after waiting for a packet to be received from the port that the packet specifies.

You can access different properties related to a DatagramSocket using other techniques. Here's an example:

Notes

Function	Usage
InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, null is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns true if the socket is bound to an address. Returns false otherwise.
boolean isConnected( )	Returns true if the socket is connected to a server. Returns false otherwise.
void setSoTimeout(int millis) throws SocketException	Sets the time-out period to the number of milliseconds passed in millis.

Datagram Packet Class

DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte data[ ], int size): It defines the size of a packet and the buffer that will hold the data. It is employed in the DatagramSocket data reception process.

DatagramPacket(byte data[ ], int offset, int size): One can designate an offset into the buffer where the data will be kept.

Notes

DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port): It gives an address and port that a DatagramSocket uses to figure out where to send the packet's contents.

DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port): Starting at the designated offset into the data, it sends packets.

Consider filling and addressing an envelope for the first two forms, then creating a "in box" for the second two.

A packet's address, port number, raw data, and length can all be obtained using the methods defined by DatagramPacket, some of which are displayed here. Generally speaking, packets that are received are handled by the get methods, while packets that are sent are handled by the set methods.

Function	Usage
InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress ipAddress)	Sets the address to which a packet will be sent. The address is specified by ipAddress.
void setData(byte[ ] data)	Sets the data to data, the offset to zero, and the length to number of bytes in data
void setData(byte[ ] data, int idx, int size)	Sets the data to data, the offset to idx, and the length to size.
void setLength(int size)	Sets the length of the packet to size.
void setPort(int port)	Sets the port to port.

Notes

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Java program to illustrate datagrams
import java.net.*;
class WriteServer {
    // Specified server port
    public static int serverPort = 998;
    // Specified client port
    public static int clientPort = 999;
    public static int buffer_size = 1024;
    public static DatagramSocket ds;
    // an array of buffer_size
    public static byte buffer[] = new byte[buffer_size];
    // Function for server
    public static void TheServer() throws Exception
    {
     int pos = 0;
     while (true) {
     int c = System.in.read();
     switch (c) {
     case -1:
     // -1 is given then server quits and returns
     System.out.println("Server Quits.");
     return;
    case '\r':
     break; // loop broken
     case '\n':
     // send the data to client
     ds.send(new DatagramPacket(buffer, pos,
InetAddress.getLocalHost(), clientPort));
     pos = 0;
     break;
    default:
     // otherwise put the input in buffer array
     buffer[pos++] = (byte)c;
}
}
}
```

Notes

```
        // Function for client
        public static void TheClient() throws Exception
    {
         while (true) {
         // first one is array and later is its size
         DatagramPacket p = new DatagramPacket(buffer, buffer.length);
         ds.receive(p);
         // printing the data which has been sent by the server
         System.out.println(new String(p.getData(), 0, p.getLength()));
    }
    }
        // main driver function
        public static void main(String args[]) throws Exception
    {
         // if WriteServer 1 passed then this will run the server function
         // otherwise client function will run
         if (args.length == 1) {
         ds = new DatagramSocket(serverPort);
         TheServer();
    }
         else {
         ds = new DatagramSocket(clientPort);
         TheClient();
    }
    }
    }
```

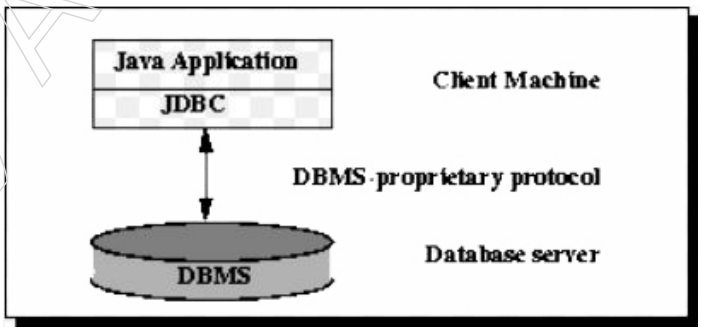
This sample program is restricted by the DatagramSocket constructor to running between two ports on local machine. To use the program, run

java WriteServer

in one window; this will be the client. Then run

java WriteServer 1

5.3.3 JDBC Architecture



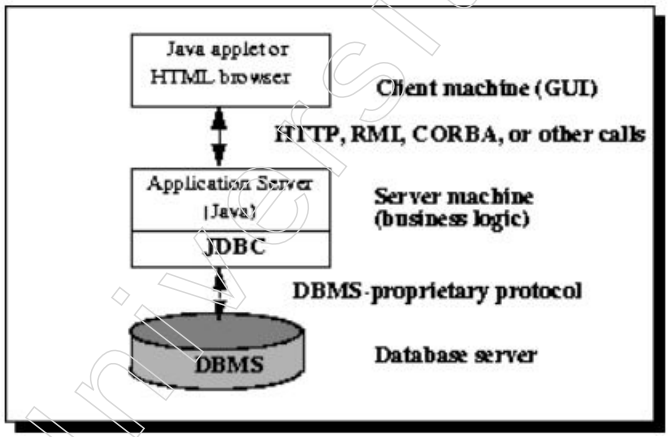
Notes

The JDBC API supports both two-tier and three-tier processing models for database access.

A Java application talks directly to the data source in two-tier model. This requires a JDBC driver that can communicate with the particular data source which is being accessed. User's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server architecture, with the user's machine as the client, and the machine housing the data source as the server.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.

MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.



Connection Interface:

The Connection interface is a factory of Statement. A Connection is the session between java application and database.

For Example, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be utilized to get the object of Statement and DatabaseMetaData. The Connection interface provide numerous methods for transaction management similar to commit(), rollback() etc.

public Statement createStatement(): creates a statement object that can be applied to execute SQL queries.

Purpose of JDBC

Databases must communicate with enterprise applications built with Java EE technology in order to store data unique to the application. Thus, effective database connectivity is necessary for database interaction, and this can be accomplished by utilising the Open Database Connectivity (ODBC) driver. This driver is used to interface or communicate with different database types, including Oracle, MS Access, MySQL, and SQL Server databases, when used with JDBC.

Components of JDBC

Notes

There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

1. JDBC API: It offers a range of interfaces and methods to facilitate simple database connectivity. In order to demonstrate WORA (write once, run anywhere) capabilities, it offers the following two packages, which include the Java SE and Java EE platforms. JDBC API classes and interfaces are contained in the java.sql package.

   Additionally, it offers a standard for database and client application connectivity.
2. JDBC Driver manager: It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.
3. JDBC Test suite: It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.
4. JDBC-ODBC Bridge Drivers: It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the sun.jdbc.odbc package which includes a native library to access ODBC characteristics.

Types of JDBC Architecture(2-tier and 3-tier)

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

Two-tier model: A Java programme speaks with the data source directly. The JDBC driver facilitates communication between the data source and the application. The responses to queries that a user submits to the data source are returned to the user in the form of results.

A user may be linked to a network while the data source is situated on a separate computer. This type of setup is called client/server; the user's computer serves as the client, and the machine that is running the data source serves as the server.

Three-tier model: This involves sending the user's inquiries to middle-tier services, which then send the commands back to the data source. The user receives the results after they have been forwarded back to the middle tier.

Directors of management information systems find this kind of approach to be very helpful.

What is API?

Before we get into JDBC drivers, let's review API in more detail.

Application Programming Interface is referred to as API. It is essentially a collection of guidelines and procedures that facilitate data transfers and enable communication between various software programmes. Without having direct access to the other programme's underlying code or data, one application can use an API to request information or carry out a function from another application.

JDBC Drivers are used by JDBC API to establish a database connection.

Working of JDBC

Java applications that require database communication must be programmed using the JDBC API. For Java applications to support JDBC, a JDBC driver that supports data sources like Oracle and SQL Server must be added. This can be done dynamically during runtime. The corresponding data source is intelligently communicated by this JDBC driver.

Notes

Creating a simple JDBC application:

```
//Java program to implement a simple JDBC application
package com.vinayak.jdbc;
import java.sql.*;
public class JDBCDemo {
    public static void main(String args[])
        throws SQLException, ClassNotFoundException
    {
        String driverClassName
        = “sun.jdbc.odbc.JdbcOdbcDriver”;
        String url = “jdbc:odbc:XE”;
        String username = “scott”;
        String password = “tiger”;
        String query
        = “insert into students values(109, ‘bhatt’)”;
        // Load driver class
        Class.forName(driverClassName);

        // Obtain a connection
        Connection con = DriverManager.getConnection(
            url, username, password);

        // Obtain a statement
        Statement st = con.createStatement();

        // Execute the query
        int count = st.executeUpdate(query);
        System.out.println(
            “number of rows affected by this query= “
            + count);
        // Closing the connection as per the
        // requirement with connection is completed
        con.close();
    }
} // class
```

The above example demonstrates the basic steps to access a database using JDBC. The application uses the JDBC-ODBC bridge driver to connect to the database. You must import java.sql package to provide basic SQL functionality and use the classes of the package.

5.3.4 Java.sql* Package

Notes

This package provides the APIs for accessing and processing data which is stored in the database especially relational database by using the java programming language. It includes a framework where we different drivers can be installed dynamically to access different databases especially relational databases.

This java.sql package contains API for the following:

1) Making a Connection with a Database With The Help of DriverManager Class
   a. DriverManager class: It helps to make a connection with the driver.
   b. SQLPermission class: It provides a permission when the code is running within a Security Manager, such as an applet. It attempts to set up a logging stream through the DriverManager class.
   c. Driver interface: This interface is mainly used by the DriverManager class for registering and connecting drivers based on JDBC technology.
   d. DriverPropertyInfo class: This class is generally not used by the general user.
2) Sending SQL Parameters to a Database:
   a. Statement interface: It is used to send basic SQL statements.
   b. PreparedStatement interface: It is used to send prepared statements or derived SQL statements from the Statement object.
   c. CallableStatement interface: This interface is used to call database stored procedures.
   d. Connection interface: It provides methods for creating statements and managing their connections and peoperties.
   e. Savepoint: It helps to make the savepoints in a transaction.
3. Updating and retrieving the results of a query:

   ResultSet interface: This object maintains a cursor pointing to its current row of data. The cursor is initially positioned before the first row. The next method of the resultset interface moves the cursor to the next row and it will return false if there are no more rows in the ResultSet object. By default ResultSet object is not updatable and has a cursor that moves forward only.

4) Providing Standard Mappings for SQL Types to Classes and Interfaces in Java Programming Language.
   - Array interface: It provides the mapping for SQL Array.
   - Blob interface : It provides the mapping for SQL Blob.
   - Clob interface: It provides the mapping for SQL Clob.
   - Date class: It provides the mapping for SQL Date.
   - Ref interface: It provides the mapping for SQL Ref.
   - Struct interface: It provides the mapping for SQL Struct.
   - Time class: It provides the mapping for SQL Time.
   - Timestamp: It provides the mapping for SQL Timestamp.
   - Types: It provides the mapping for SQL types.
6) Exceptions
   a. SQLException: It is thrown by the methods whenever there is a problem while accessing the data or any other things.

Notes

b. SQLWarning: This exception is thrown to indicate the warning.

c. BatchUpdateException: This exception is thrown to indicate that all commands in a batch update are not executed successfully.

d. DataTruncation: It is thrown to indicate that the data may have been truncated.

7) Custom Mapping an SQL User-defined Type (UDT) to a Class in The Java Programming Language.

a. SQLData interface: It gives the mapping of a UDT to an instance of this class.

b. SQLInput interface: It gives the methods for reading UDT attributes from a stream.

c. SQLOutput: It gives the methods for writing UDT attributes back to a stream.

5.3.5 SQL Statements

JDBC SELECT query: A Sample Database

Before looking at the SQL queries, let's take a quick look at our sample database.

In all of these examples I'll access a database named "Demo", and in these SELECT query examples I'll access a database table named "Customers" that's contained in the Demo database. Here's what the Customers table looks like:

Cnum	Lname	Salutation	City	Snum
1001	Simpson	Mr.	Springfield	2001
1002	MacBeal	Ms.	Boston	2004
1003	Flinstone	Mr.	Bedrock	2003
1004	Cramden	Mr.	New York	2001

How to perform a JDBC SELECT query against a database

Querying a SQL database with JDBC is typically a three-step process:

Create a JDBC ResultSet object.

Execute the SQL SELECT query you want to run.

Read the results.

The hardest part of the process is defining the query you want to run, and then writing the code to read and manipulate the results of your SELECT query.

Creating a valid SQL SELECT query

To demonstrate this , write this simple SQL SELECT query:

```
SELECT Lname FROM Customers
WHERE Snum = 2001;
```

This statement returns each Lname (last name) record from the Customers database, where Snum (salesperson id-number) equals 2001. In plain English, you might say "give me the last name of every customer where the salesperson id-number is 2001".

Now that we know the information we want to retrieve, how do we put this SQL statement into a Java program? It's actually very simple. Here's the JDBC code necessary to create and execute this query:

```
Statement stmt = conn.createStatement();
```

ResultSet rs = stmt.executeQuery("SELECT Lname FROM Customers WHERE Snum = 2001");

Notes

Reading the JDBC SELECT query results (i.e., a Java JDBC ResultSet)

After you execute the SQL query, how do you read the results? Well, JDBC makes this pretty easy also. In many cases, you can just use the next() method of the ResultSet object. After the previous two lines, you might add a loop like this to read the results:

```
while (rs.next()) {
String lastName = rs.getString("Lname");
 System.out.println(lastName +"\n");
}
```

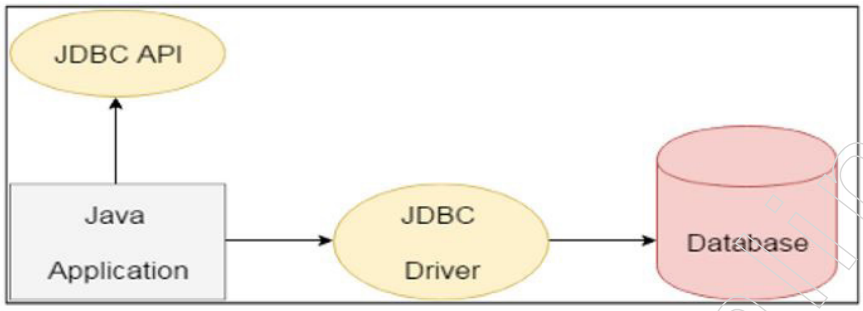
The full source code for our example JDBC program

```
import    java.sql.*;
 /**
 *    A JDBC SELECT (JDBC query) example program.
 */
class Query1 {
 public static void main (String[] args) {
 try {
 String url = "jdbc:msql://200.210.220.1:1114/Demo";
 Connection conn = DriverManager.getConnection(url,"","");
 Statement stmt = conn.createStatement();
 ResultSet rs;
 rs = stmt.executeQuery("SELECT Lname FROM Customers WHERE Snum = 2001");
 while ( rs.next() ) {
 String lastName = rs.getString("Lname");
 System.out.println(lastName);
 }
 conn.close();
 } catch (Exception e) {
 System.err.println("Got an exception! ");
 System.err.println(e.getMessage());
 }
 }
 }
```

5.3.6 Java Database Connectivity: Part 1

To access tabular data stores in any relational database, we can use JDBC API. With the help of JDBC API, we can save, update, fetch and delete from the database.

Notes



The java.sql package contains classes and interfaces for JDBC API. Below are a list of popular interfaces of JDBC API:

Driver interface

Connection interface

Statement interface

PreparedStatement interface

CallableStatement interface

ResultSet interface

ResultSetMetaData interface

DatabaseMetaData interface

RowSet interface

A list of popular classes of JDBC API are:

DriverManager class

Blob class

Clob class

Types class

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Because of thin driver this is now does not have that importance. The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

Notes

It is not written entirely in java. The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java. The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

5.3.7 Java Database Connectivity: Part 2

Java Database Connectivity

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity amongst the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks stated below that are commonly associated with database usage.

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

1. Register the driver class: The forName() method of class is used to register the driver class. This method is used to dynamically load the driver class.
2. Create the connection object: The getConnection() method of DriverManager class is used to establish connection with the database.
3. Create the Statement object: The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.
4. Execute the query: The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.
5. Close the connection object: By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Java Database Connectivity with Oracle

To connect java application with the oracle database, you must follow 5 steps. In this example, we are using Oracle 10g as the database. So we need to know following information for the oracle database:

1. Driver class: The driver class for the oracle database is oracle.jdbc.driver. OracleDriver.
2. Connection URL: The connection URL for the oracle10G database is jdbc:oracle:thin:@ localhost:1521:xe where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.
3. Username: The default username for the oracle database is system.

Notes

4. Password: It is the password given by the user at the time of installing the oracle database.

Summary

- AWT (Abstract Window Toolkit) in Java is a set of APIs used for creating graphical user interfaces (GUIs) and rich graphics applications. Part of the java.awt package, AWT provides a platform-independent way to create and manage windows, dialog boxes, buttons, text fields, and other GUI components. It includes classes for event handling, layout management, drawing graphics, and handling images. AWT is designed to be used in conjunction with the native GUI toolkit of the host operating system, which makes it highly portable but also somewhat heavyweight compared to more modern GUI frameworks like Swing and JavaFX. Despite being largely superseded by these newer frameworks, AWT remains fundamental for understanding Java GUI development.
- Event handling in Java involves the management and processing of user actions or system-generated events, such as mouse clicks, keyboard input, or window resizing. It is facilitated through event listeners and adapters provided by the java.awt.event and javax.swing.event packages. Event listeners are interfaces that define callback methods to respond to specific events, while adapters are abstract classes that provide default implementations for all methods in an event listener interface, allowing subclasses to override only the methods of interest.
- Swing is a GUI (Graphical User Interface) toolkit for Java, providing a set of classes and controls for creating rich, platform-independent user interfaces. It is part of the javax.swing package and is built on top of the AWT (Abstract Window Toolkit). Swing classes and controls include components such as buttons, labels, text fields, checkboxes, radio buttons, lists, tables, and more, allowing developers to create sophisticated and interactive GUI applications. Swing follows the Model-View-Controller (MVC) design pattern, where components represent the view, models hold the data, and controllers manage user interactions.
- JDBC (Java Database Connectivity) architecture in Java provides a standardized interface for accessing relational databases from Java applications. It consists of several layers, including the JDBC API, DriverManager, Driver, Connection, Statement, ResultSet, and Driver-specific implementation classes. The JDBC API defines methods and interfaces for connecting to databases, executing SQL queries, and processing query results. The DriverManager class manages the set of database drivers available to the application, while the Driver interface provides a mechanism for database-specific driver implementations to register themselves with the DriverManager.
- In Java, TCP/IP (Transmission Control Protocol/Internet Protocol) is the underlying communication protocol used for networking and data exchange over the Internet. Java provides comprehensive support for TCP/IP through classes and interfaces in the java.net package. This includes classes for creating network sockets, establishing connections, sending and receiving data streams, and handling network events. The Socket class represents an endpoint for communication between two machines over a network, while the ServerSocket class represents a server socket that waits for incoming client connections. Java also provides classes like URLConnection and URL for working with URLs, InetAddress for representing IP addresses, and DatagramPacket and DatagramSocket for working with UDP (User Datagram Protocol). With Java's TCP/IP support, developers can build robust networked applications, ranging from simple client-server applications to complex distributed

systems, while benefiting from Java's platform independence and built-in security features.

Notes

Glossary

- AWT: Abstract Window Toolkit
- API: Application Program Interface
- GUI: Graphical User Interface
- OS: Operating System
- JFC: Java Foundation Classes
- JDK: Java Development Kit
- JFrame: A frame is an instance of JFrame. Frame is a window that can have title, border, menu, text fields , buttons etc. A Swing application must have a frame to have the components added to it.
- JPanel: A panel is an instance of JPanel. Frame can have more than one panels and each panel can have several components. Those are parts of Frame. Panels are useful for grouping components and placing them to appropriate locations in a frame.

Check Your Understanding

1. In AWT, what is the purpose of the EventQueue class?
   a) It manages events in a multithreaded environment.
   b) It handles graphics rendering.
   c) It manages layout in containers.
   d) It controls the appearance of components.
2. In the three-tier model of application architecture, where are user commands sent before reaching the data source?
   a) Data source
   b) User's machine
   c) Middle tier
   d) Network
3. What does AWT stand for?
   a) Advanced Web Technologies
   b) Abstract Window Toolkit
   c) Application Windowing Toolkit
   d) Advanced Widget Toolbox
4. Which class is used to create a top-level window in Swing?
   a) JFrame
   b) JPanel
   c) JWindow
   d) JFrameWindow
5. Which protocol provides reliable communication between two applications and is typically used over the Internet Protocol (IP)?
   a) User Datagram Protocol (UDP)
   b) Transmission Control Protocol (TCP)

Notes

c) File Transfer Protocol (FTP)

d) Hypertext Transfer Protocol (HTTP)

Exercise

1. What is difference between Swing and AWT in Java?
2. What is difference between paint and repaint in Java Swing?
3. How many types of layout is there?
4. What is difference between BorderLayout and GridLayout?
5. Why Swing is called light weight?

Learning Activities

1. Create a simple Swing GUI application for a library system. The application should allow users to add new books to the library and view the list of existing books.
2. Create a Java application that interacts with a relational database to manage information about students. Assume you have a database named UniversityDB with a table named Students having the following columns: student_id, first_name, last_name, email, and enrollment_date.

Check Your Understanding- Answers

1. a) 2. c) 3. b) 4. a)
5. b)

Further Readings and Bibliography

1. R. Nageswara Rao. (2016). Core Java: An Integrated Approach. Dreamtech Press, 720 pages.
2. E. Balagurusamy. (2023). Programming with Java. 7th Edition. McGrawHill Publications, 592 pages.
3. Barry A. Burd. (2017). Beginning Programming with Java for Dummies. 5th Edition. Wiley Publications.
4. Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action. Dreamtech Press.