Institute for Advanced Computing And Software Development (IACSD) Akurdi, Pune

Object Oriented Programming with Java 8

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

Features of Java

Object Oriented

Everything in Java is coded using OO principles. This facilitates code modularization, reusability, testability, and performance.

Interpreted/Portable

Java source is compiled into platform-independent bytecode, which is then interpreted (compiled into native-code) at runtime. Java code is "Write Once, Run Everywhere"

Simple

Java has a familiar syntax, automatic memory management, exception handling, single inheritance, standardized documentation, and a very rich set of libraries .

Secure/Robust

Due to its support for strong type checking, exception handling, and memory management, Java is immune to buffer- overruns, leaked memory, illegal data access. Additionally, Java comes with a Security Manager too.

Scalable

Java is scalable both in terms of performance/throughput, and as a development environment. A single user can play a Java game on a mobile phone, or millions of users can shop though a Java-based ecommerce enterprise application. ***High-performance/Multi-threaded***

With its Just-in-Time compiler, Java can achieve (or exceed) performance of native applications. Java supports multi-threaded development out-of-the-box.

Dynamic

Java can load application components at run-time even if it knows nothing about them. Each class has a run-time representation.

Distributed

Java comes with support for networking, as well as for invoking methods on remote (distributed) objects through RMI.

About JVM,JRE,JDK

Java Development Kit [JDK] is the core component of Java Environment and provides all the tools, executable and binaries required to compile, debug and execute a Java Program. JDK is a platform specific software and that's why we have separate installers for Windows, Mac and Unix systems.

Java Virtual Machine[JVM] is the heart of java programming language. When we run a program, JVM is responsible to converting Byte code to the machine specific code. JVM is also platform dependent and provides core java functions like memory management, garbage collection, security etc.

Java Runtime Environment [JRE] is the implementation of JVM, it provides platform to execute java programs. JRE consists of JVM and java binaries and other class libraries to execute any program successfully. To execute any java program, JRE is required.

JVM architecture with journey of java program from source code to execution stage

As shown in the below architecture diagram, JVM subsystems are :

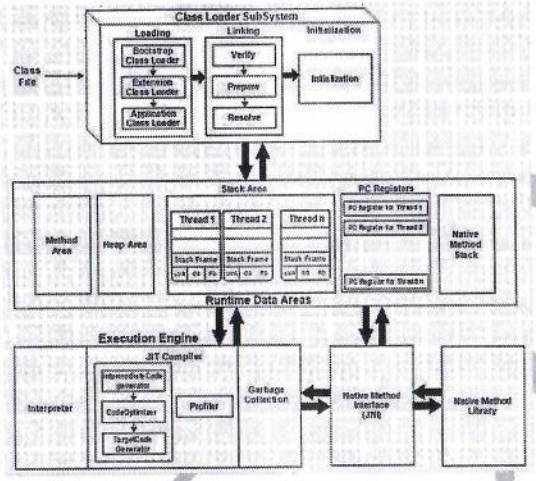
- Class Loader Subsystem
- Runtime Data Area
- Execution Engine

Class Loader Subsystem

Loading : Class loader dynamically loads java classes. It loads, links and initializes the class file when it refers to a class for the first time at runtime, not compile time. Class Loaders follow Delegation Hierarchy Algorithm while loading the class files. 3 types are,

1. Boot Strap Class Loader – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.
2. Extension Class Loader – Responsible for loading classes which are inside the ext folder (jre\lib).

3. Application Class Loader –Responsible for loading Application Level Class path, path mentioned Environment Variable etc.



Linking : Linking stage involves,

- Verify – Byte code verifier will verify byte code using checksum.
- Prepare – For all static variables memory will be allocated and assigned with default values.
- Resolve – All symbolic memory references are replaced with the original references from Method Area.
- Initialization: Here all static variables will be assigned with the original values, and the static blockwill be executed.

Runtime Data Area

The Runtime Data Area is divided into 5 major components:

- Method Area – All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- Heap Area – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
- Stack Area – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three sub entities:
  1. Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.
  2. Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
  3. Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.
- PC Registers – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

- Native Method stacks – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

Execution Engine

The byte code which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the byte code and executes it piece by piece.

- Interpreter – The interpreter interprets the byte code faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
- JIT Compiler – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
- Intermediate Code generator – Produces intermediate code
- Code Optimizer – Responsible for optimizing the intermediate code generated above
- Target Code Generator – Responsible for Generating Machine Code or Native Code
- Profiler – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

Garbage Collector: Collects and removes unreferenced objects.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides theNative Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries which is required for theExecution Engine.

Bytecode

Bytecode is in a compiled Java programming language [by javac command] format and has the .class extension executed by Java Virtual Machine (JVM). The Java bytecode gets processed by the Java virtual machine (JVM) instead of the processor. The JVM transforms program code into readable machine language for the CPU because platforms utilize different code interpretation techniques. A JVM converts bytecode for platform interoperability, but bytecode is not platform-specific. JVM is responsible for processing & running the bytecode.

JIT

The magic of java "Write once, run everywhere" is bytecode. JIT improves the performance of Java applications by compiling bytecode to native machine code at run time. JIT is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

Typical compilers take source code and completely convert it into machine code, JITs take the same source code and convert it into an intermediary "assembly language," which can then be pulled from when it's needed. And that's the key. Assembly code is interpreted into machine code on call—resulting in a faster translation of only the code that you need. JIT have access to dynamic runtime information and are able to optimize code. JITs monitor and optimize while they run by finding code more often called to make them run better in the future.

JITs reduce the CPU's workload by not compiling everything all at once, but also because the resulting compiled code is optimized for that particular CPU. It's why languages with JIT compilers are able to be so "portable" and run on any platform or OS.

Platform independence

Java is a platform independent programming language, because your source code can be executed on any platform [e.g. Windows, Mac or Linux etc..]. When you install JDK software on your system , JVM is automatically installed on your system. When we compile Java code then .class file or bytecode is generated

by javac compiler. For every operating system separate JVM is available which is capable to read the .class file or byte code and execute it by converting to native code for that specific machine. We compile code once and run everywhere.

Language Fundamentals

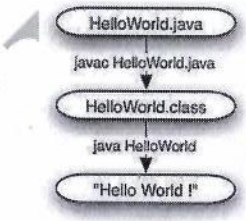
Sample Java Program

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

- All code is contained within a class, in this case HelloWorld.
- The file name must match the class name and have a .java extension, for example: HelloWorld.java
  All executable statements are contained within a method, in this case named main().
- Use System.out.println() to print text to the terminal.
- Classes and methods (including other flow-control structures) are always defined in blocks of code enclosed by curly braces ({ }).
- All other statements are terminated with a semi-colon (;).
- Java language is case-sensitive.

Compiling Java Programs

- The JDK comes with a command-line compiler: javac.
- It compiles source code into Java bytecode, which is low-level instruction set similar to binary machine code.
- The bytecode is executed by a Java virtual machine (JVM), rather than a specific physical processor.
- To compile our HelloWorld.java, you could go to the directory containing the source file and execute:
  javac HelloWorld.java
- This produces the file HelloWorld.class, which contains the Java bytecode.
- You can view the generated byte-code, using the -c option to javap, the Java class disassembler. For example: javap -c HelloWorld



To run the bytecode, execute:
java HelloWorld

The main() Method

- A Java application is a public Java class with a main() method.
- The main() method is the entry point into the application.
- The signature of the method is always:

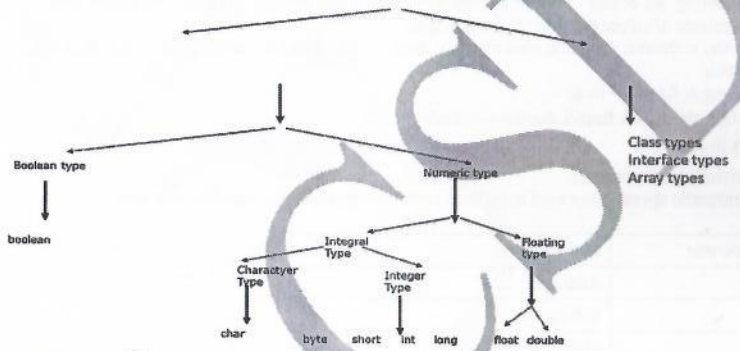
public static void main(String[] args)

- Command-line arguments are passed through the args parameter, which is an array of Strings

Basic rules For Writing Java Programs

1. Java compiler doesn't allow accessing of un initialized data members.
2. Files with no public classes(default scoped) can have a name that does not match with any of the classes in the file .
3. A file can have more than one non public class.
4. There can be only one public class per source code file.
5. If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Example {....} must be in a source code file named Example.java.
6. Java compiler doesn't allow accessing of un-inited vars. eg : int n; sop(n);//error

Data Types in Java



Primitive Data Types

Type	Size	Range	Default Value
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	[-263, 263-1]	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Variables

Variable is nothing but identification of memory location.

Types of variables Local

Variables declared inside method are local variable they have scope only within that methods.

Instance

Instance variables are declared inside class but used outside method.

Static

Static variable are same as that of instance variable but having keyword static.
They are also called as class variable because they are specified and can be access either class name or object name.

Type Casting in Java Conversions regarding primitive types

Automatic conversions(also called as widening) ---here it goes for Automatic promotions byte---->short--->int---> long--->float--->double char ---> int eg : char ch='a'; long --->float ---is considered automatic type of conversion(since float data type can hold larger range of values than long data type)

Narrowing conversion --- forced conversion(type-casting) eg double ---> int float --> long double --> float

Rules for Casting src & dest - must be compatible, typically dest data type must be able to store larger magnitude of values than that of src data type.

1. Any arithmetic operation involving byte,short --- automatically promoted to --int 2. int & long ---> long
3. long & float ---> float
4. byte,short.....& float & double----> double

Operators in Java

1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

3. Java Relational Operators

Relational operators are used to check the relationship between two operands.It returns either true or false.

Operator	Description	Example
==	Is Equal To	2 == 8 returns false
!=	Not Equal To	2 != 8 returns true
>	Greater Than	2 > 8 returns false
<	Less Than	2 < 8 returns true
>=	Greater Than or Equal To	2 >= 8 returns false
<=	Less Than or Equal To	2 <= 8 returns false

4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

5. Java Unary Operators

Unary operators are used with only one operand.

Operator	Meaning
+	Unary plus: not necessary to use since numbers are positive without using it
-	Unary minus: inverts the sign of an expression
++	Increment operator: increments value by 1
--	Decrement operator: decrements value by 1
!	Logical complement operator: inverts the value of a boolean

6. Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

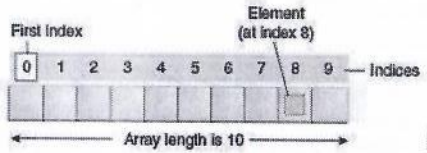
variable = Expression ? expression1 : expression2

Here,

If the Expression is true, expression1 is assigned to the variable.
If the Expression is false, expression2 is assigned to the variable.

Java Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.



Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.In Java, array is an object of a dynamically generated class. Java array inherits the Object class.

Types of Array in java

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java dataType[] arr; (or)
dataType []arr; (or) dataType arr[];

Instantiation of an Array in Java arrVar=new datatype[size];

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).
Syntax to Declare Multidimensional Array in Java dataType[][] arrVar; (or) dataType [][]arrVar; (or) dataType arrVar [][]; (or) dataType []arrVar [];

Instantiate Multidimensional Array in Java int[][] arr=new int[3][3];//3 row and 3 column

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

int arr[][] = new int[3][]; arr[0] = new int[3]; arr[1] = new int[4]; arr[2] = new int[2];

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos,
Object dest, int destPos, int length)

The two Object arguments specify the array to copy from and the array to copy to. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

for-each Loop for Java Array [Enhanced For Loop]

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below: for(data_type variable:array){
//body of the loop
}

Object Oriented Concepts in Java

Classes in Java

- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- As such, the class forms the basis for object-oriented programming in Java.
- Any concept you wish to implement in a Java program must be encapsulated within a class. a class is that it defines a new data type.
- Once defined, this new type can be used to create objects of that type.
- Thus, a class is a template for an object, and an object is an instance of a class. Everything in Java is defined in a class.
- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.
- In its simplest form, a class just defines a collection of data e.g.

```
class Box {
        //data members  double height,width,depth;

        double calVolume(); //methods
}
```

- The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class

Objects in Java

- Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.
- To create an object (instance) of a particular class, use the new operator, followed by an invocation of a constructor for that class, such as:new Box();
- The constructor method initializes the state of the new object.
- The new operator returns a reference to the newly created object.
- As with primitives, the variable type must be compatible with the value type when using object references, as in:
  Box b1 = new Box();
- To access member data or methods of an object, use the dot (.) notation: variable.field or variable.method().

Static vs. Instance members Static (or class) data members

- Unique to the entire class
- Shared by all instances (objects) of that class
- Accessible using ClassName.fieldName
- The class name is optional within static and instance methods of the class, unless a local variable of the same name exists in that scope.

- Subject to the declared access mode, accessible from outside the class using the same syntax **Instance or data members**
- Unique to each instance (object) of that class (that is, each object has its own set of instance fields)
- Accessible within instance methods and constructors using this.fieldName
- The **this** qualifier is optional, unless a local variable of the same name exists in that scope.
- Subject to the declared access mode, accessible from outside the class from an object reference using objectRef.fieldName

Static vs. Instance Methods

- Static methods can access only static data and invoke other static methods.
  - Often serve as helper procedures/functions
  - Use when the desire is to provide a utility or access to class data only Instance methods can access both instance and static data and methods.
  - Implement behavior for individual objects
  - Use when access to instance data/methods is required An example of static method use is Java's Math class.
  - All of its functionality is provided as static methods implementing mathematical functions (e.g., Math.sin()).
  - The Math class is designed so that you don't (and can't) create actual Math instances.
- Static methods also are used to implement factory methods for creating objects, a technique discussed later in this class.

Constructors

- Constructors are like special methods that are called implicitly as soon as an object is instantiated (i.e. on new ClassName()).
  - Constructors have no return type (not even void).
  - The constructor name must match the class name.
- If you don't define an explicit constructor, Java assumes a default constructor The default constructor accepts no arguments.
  - The default constructor automatically invokes its base class constructor with no arguments, as discussed later in this module.
- You can provide one or more explicit constructors to:
  - Simplify object initialization (one line of code to create and initialize the object)
  - Enforce the state of objects (require parameters in the constructor)
  - Invoke the base class constructor with arguments, as discussed later in this module.
- Adding any explicit constructor disables the implicit (no argument) constructor.
- As with methods, constructors can be overloaded.
- Each constructor must have a unique signature.
  - The parameter type list must be different, either different number or different order.
  - Only parameter types determine the signature, not parameter names.
- One constructor can invoke another by invoking this(param1, param2, ...) as the first line of its implementation.

Access Modifiers: Enforcing Encapsulation

 Access modifiers are Java keywords you include in a declaration to control access. You can apply access modifiers to:
  - Instance and static fields

  - Instance and static methods
  - Constructors
  - Classes
  - Interfaces (discussed later in this module)

Access Modifier	Description
public	Accessible from any class
protected	Accessible from all classes in the **same package** or any **child classes** regardless of the package
Default (no modifier)	Accessible only from the classes in the **same package** (also known as *friendly*)
private	Accessible only from within the same class (or any nested classes)

Primitive Wrappers / Wrapper Classes

- Many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object The java.lang package includes a class for each Java primitive type:
  - Boolean, Byte, Short, *Character*, *Integer*, Float, Long, Double, *Void*Used for:
  - Storing primitives in Object based collections
  - Parsing/decoding primitives from Strings, for example:
    int value = Integer.parseInt(str);
  - Converting/encoding primitives to Strings
- Since Java 5, Java supports implicit wrapping/unwrapping of primitives as needed. This compilerfeature is called auto-boxing/auto-unboxing.
- **Autoboxing:** Converting a primitive value into an object of the correspondingwrapper classis called autoboxing.
- **Unboxing:** Converting an object of a wrapper type to its corresponding primitive value is called unboxing

```
Integer Iobj = 10; //Boxing int k = Iobj //Unboxing
```

Java Packages

A package is simply a container that groups related types (Java classes, interfaces, enumerations, and annotations). For example, in core Java, the System class belongs to the java.lang package. The package contains all the related types that are needed for the basic java development.

 Built-in Package

Built-in packages are existing java packages that come along with the JDK. For example, java.lang, java.util, java.io

 User-defined Package

Java also allows you to create packages as per your need. These packages are called user-defined packages.

Defining a Java package

To define a package in Java, you use the keyword package.

package packageName;
Java uses file system directories to store packages.

For example:

```
└── com
    └── test
        └── TestPlanet.java
```

Now, edit Test.java file, and at the beginning of the file, write the package statement as:
package com.test;

Here, any class that is declared within the test directory belongs to the com.test package.

Importing packages in Java

Java has an import statement that allows you to import an entire package, or use only certain classes and interfaces defined in the package.

The general form of import statement is:

import package.name.ClassName; // To import a certain class only import package.name.* // To import the whole package
For example,

```
import java.util.Date; // imports only Date class
import java.io.*;      // imports everything inside java.io package
```

Static import in Java

In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly.

Garbage Collection in Java

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically.

The technique that accomplishes this is called *garbage collection*.

Java garbage collection is the process by which Java programs perform automatic memory management. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed or unreachable. An object is said to be unreachable iff it doesn't contain any reference to it. An object is said to be eligible for GC (garbage collection) iff it is unreachable. The garbage collector finds these unused objects and deletes them to free up memory.
Even though the programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable (thus eligible for GC) if it is no longer required.
There are generally four different ways to make an object eligible for garbage collection.

1. Nullifying the reference variable
2. Re-assigning the reference variable

3. Object created inside method
4. Island of Isolation

Once we made object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed.
But when JVM runs Garbage Collector, we can not expect.
We can also request JVM to run Garbage Collector. There are two ways to do it :

- **Using *System.gc()* method** : System class contain static method ***gc()*** for requesting JVM to run Garbage Collector.
- **Using *Runtime.getRuntime().gc()* method**:Runtime classallows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class.

Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

- Inheritance allows you to define a class based on the definition of another class.
  - The class it inherits from is called a base class or a parent class.
  - The derived class is called a subclass or child class.
- Subject to any access modifiers, which we'll discuss later, the subclass gets access to the fields and methods defined by the base class.
  - The subclass can add its own set of fields and methods to the set it inherits from its parent.
- Inheritance simplifies modeling of real-world hierarchies through generalization of common features.
  - Common features and functionality is implemented in the base classes, facilitating code reuse.
  - Subclasses can extended, specialize, and override base class functionality.

Inheritance provides a means to create specializations of existing classes. This is referred to as subtyping. Each subclass provides specialized state and/or behavior in addition to the state and behavior inherited by the parent class. For example, a manager is also an employee but it has a responsibility over a department, whereas a generic employee does not.

Inheritance in Java

- You define a subclass in Java using the extends keyword followed by the base class name. For example:

```
class Manager extends Employee
{
// mgr class members
}
```

- In Java, a class can extend at most one base class. That is, multiple inheritance is not supported.
- If you don't explicitly extend a base class, the class inherits from Java's Object class, discussed later in this module.

- Java supports multiple levels of inheritance.
  - For example, Child can extend Parent, which in turn extends Grandparent, and so on.

Overriding vs. Overloading

- The subclass can override its parent class definition of fields and methods, replacing them with its own definitions and implementations.
  - To successfully override the base class method definition, the subclass method must have the same signature.
  - If the subclass defines a method with the same name as one in the base class but a different signature, the method is overloaded not overridden.
- A subclass can explicitly invoked an ancestor class's implementation of a method by prefixing super. to the method call.

"super" Keyword

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super. super has two general forms.

- **To call the superclass constructor.**
  A subclass can call a constructor defined by its superclass by use of the following form of **super**:
  super(arg-list);
  Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass' constructor.
- **To access a member of the superclass that has been hidden by a member of a subclass.** The usage has the following general form:
  super.member
  Here, member can be either a method or an instance variable.
  This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Polymorphism

- Polymorphism is the ability for an object of one type to be treated as though it were another type.
- In Java, inheritance provides us one kind of polymorphism.
  - An object of a subclass can be treated as though it were an object of its parent class, or any of its ancestor classes. This is also known as upcasting.
  - For example, if Manager is a subclass of Employee:
    Employee e = new Manager(...);
- Why is polymorphism useful? It allows us to create more generalized programs that can be extended more easily.

Abstract Classes and Methods

- An abstract class is a class designed solely for subclassing.
  - You can't create actual instances of the abstract class. You get a compilation error if you attempt to do so.
  - You design abstract classes to implement common sets of behavior, which are then shared by the concrete(instantiable) classes you derive from them.
  - You declare a class as abstract with the abstract modifier:
- An abstract method is a method with no body.
  - It declares a method signature and return type that a concrete subclass must implement.
  - You declare a method as abstract with the abstract modifier and a semicolon terminator:

- If a class has any abstract methods declared, the class itself must also be declared as abstract.
- For example, if Employee is a class with method calSalary which is abstract then it is declared as,

```
public abstract class Employee{
abstract double calSalary();
    }
```

"final" Keyword

The keyword final has three uses.

Using final to declare Constants

It is used to create the equivalent of a named constant. A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. For example:

e.g. final int SIZE= 100;

Using final to Prevent Overriding

To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

Using final to Prevent Inheritance

If we want to prevent a class from being inherited precede the class declaration with final keyword. Declaring a class as final implicitly declares all of its methods as final, too.

Interfaces

- Interfaces are used for Role Based Inheritance & Programming by Contract. When we want a common role/ behavior to be implemented by multiple classes in different hierarchies we use interfaces.
- Also for creating a contract as the class that implements an interface must implement all the methods declared in the interface. In the Java programming language, an interface is a reference type, similar to a class, which can contain only constants, method signatures, default methods, static methods, and nested types.
- Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.
- An interface defines a set of methods, without actually defining their implementation.
  - A class can then implement the interface, providing actual definitions for the interface methods.
- In essence, an interface serves as a "contract" defining a set of capabilities through method signatures and return types.
  - By implementing the interface, a class "advertises" that it provides the functionality required by the interface, and agrees to follow that contract for interaction.

Defining a Java Interface

- Use the interface keyword to define an interface in Java.
- The interface definition consists of public abstract method declarations. For example: public interface Shape { double getArea(); double getPerimeter();
  }
- All methods declared by an interface are implicitly public abstract methods.
- If required any data member must be declared and initialized by the interface are also public static &final.

Implementing a Java Interface

- You define a class that implements a Java interface using the implements keyword followed by the interface name. For example: class Circle implements Shape { // ... }

- A concrete class must then provide implementations for all methods declared by the interface.
  - Omitting any method declared by the interface, or not following the same method signatures and return types, results in a compilation error.
- An abstract class can omit implementing some or all of the methods required by an interface.
  - In that case concrete subclasses of that base class must implement the methods.
- A Java class can implement as many interfaces as needed.
  - Simply provide a comma-separated list of interface names following the implements keyword. For example:
    class ColorCircle implements Shape, Color { // ... }

A Java class can extend a base class and implement one or more interfaces.

- In the declaration, provide the extends keyword and the base class name, followed by the implements keywords and the interface name(s).

Polymorphism through Interfaces

- Interfaces provide another kind of polymorphism in Java.
  - An object implementing an interface can be assigned to a reference variable typed to the interface.
  - For example, if Circle implements the Shape interface:
    Shape s = new Circle();

Object: Java Cosmic Superclass

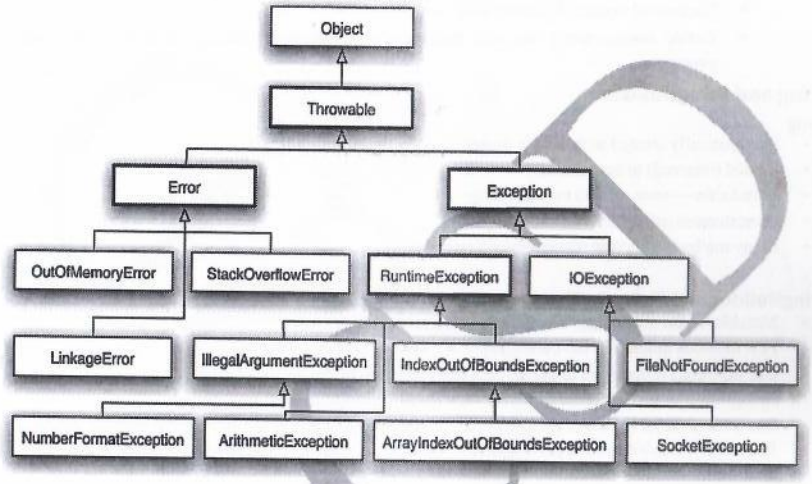
- Every class in Java ultimately has the Object class as an ancestor.
- Class **Object** is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
- This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.
- Methods of Object class are,
  - clone() - Creates and returns a copy of this object.
  - equals() - Indicates whether some other object is "equal to" this one.
  - finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
  - getClass() - Returns the runtime class of an object.
  - hashCode() - Returns a hash code value for the object.
  - notify() - Wakes up a single thread that is waiting on this object's monitor. ▪ notifyAll() - Wakes up all threads that are waiting on this object's monitor.
  - toString() - Returns a string representation of the object.
  - wait() - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Exception Handling

- Exceptions are events that occur during the execution of programs that disrupt the normal flow of instructions (e.g. divide by zero, array access out of bound, etc.).
- In Java, an exception is an object that wraps an error event that occurred within a method and contains:
  - Information about the error including its type
  - The state of the program when the error occurred
  - Optionally, other custom information □ Exception objects can be thrown and caught.
- Exceptions are used to indicate many different types of error conditions. JVM Errors:
  - OutOfMemoryError

  - StackOverflowError
  - LinkageError
- All exceptions and errors extend from a common java.lang.Throwable parent class. Only Throwable objects can be thrown and caught.

Exception Hierarchy



Checked vs. Unchecked Exceptions

□Errors and RuntimeExceptions are *unchecked* — that is, the compiler does not enforce (check) that you handle them explicitly.

All other Exceptions are *checked* — that is, the compiler enforces that you handle them explicitly.

Handling Exceptions

□ The try-catch-finally structure is used to handle exceptions.

```
try {
// Code bock
}
catch (ExceptionType1 e1) {
// Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2) {
// Handle ExceptionType2 exceptions
}
// ... finally {
// Code always executed after the
// try and any catch block
}
```

- Both catch and finally blocks are optional, but at least one must follow a try.
- The try-catch-finally structure can be nested in try, catch, or finally blocks.
- The finally block is used to clean up resources, particularly in the context of I/O.
- If you omit the catch block, the finally block is executed before the exception is propagated.

- Exceptions can be caught at any level.
- If they are not caught, they are said to *propagate* to the next method.

Custom or User Defined Exception

- If you wish to define your own exception:
  - Create an Exception class.
  - Decide if the exception should be checked or unchecked.
  - Checked extends Exception
  - Unchecked extends RuntimeException
  - Define constructor(s) that call into super's constructor(s), taking message and/or cause parameters.

String and StingBuilder

String

- Automatically created with double quotes
- Pooled (interned) to save memory
- Immutable — once created cannot change
- Concatenated using the expensive + operator
- Many methods for string manipulation/searching

StringBuilder and StringBuffer

- Mutable — can quickly grow and shrink as needed
- Few methods modifying the buffer: append(), insert(), delete(), replace()
- Use toString() if you need to get a String representation of the object's content
- StringBuffer is synchronized for thread-safe access in multithreaded applications
- Use StringBuilder for better performance unless you plan to access the object from multiple threads
  In a single-threaded context (as is generally the case), StringBuilder is faster than StringBuffer.

Java Collections and Generics

The Java Collections Framework

- Java arrays have limitations.
  - They cannot dynamically shrink and grow.
  - They have limited type safety.
  - Implementing efficient, complex data structures from scratch would be difficult.
- The Java Collections Framework is a set of classes and interfaces implementing complex collection data structures.
  - A *collection* is an object that represents a group of objects.
- It consists of the interfaces and classes which helps in working with different types of collections such as **lists, sets, maps, stacks and queues** etc.

The Collection Interface

- java.util.Collection is the root interface in the collections hierarchy.
  - It represents a group of Objects.
  - Primitive types (e.g., int) must be boxed (e.g., Integer) for inclusion in a collection.
  - More specific collection interfaces (e.g., List) extend this interface.
- The Collection interface includes a variety of methods:
  - Adding objects to the collection: add(E), addAll(Collection)
  - Testing size and membership: size(), isEmpty(), contains(E), containsAll(Collection)
  - Iterating over members: iterator()

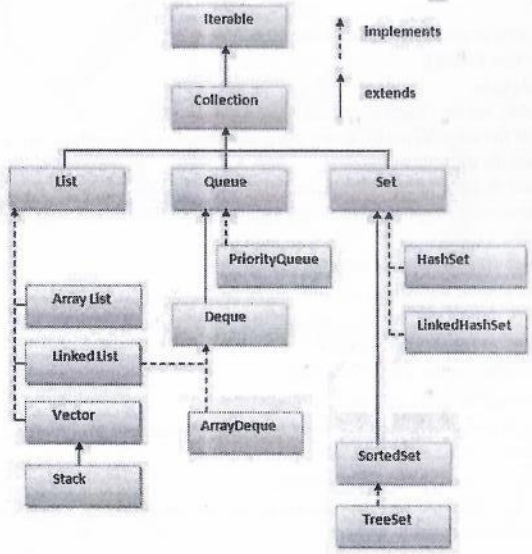
  - Removing members: remove(E), removeAll(Collection), clear(), retainAll(Collection)
  - Generating array representations: toArray(), toArray(T[])

This interface is typically used to pass collections around and manipulate them in the most generic way.

Iterating Over a Collection

- An *iterator* is an object that iterates over the objects in a collection.
- java.util.Iterator is an interface specifying the capabilities of an iterator.
  - Invoking the iterator() method on a collection returns an iterator object that implements Iterator and knows how to step through the objects in the underlying collection.
- The Iterator interface specifies the following methods:
  - hasNext() - Returns true if there are more elements in the collection; false otherwise
  - next() - Returns the next element
  - remove() - Removes from the collection the last element returned by the iterator

Collection Hierarchy



List

A List is an ordered Collection Lists may contain duplicate elements.

- ArrayList : It's Resizable-array implementation of the List interface which internally uses dynamic array. This class is roughly equivalent to Vector, except that it is unsynchronized i.e. not thread safe.
- LinkedList: Doubly-linked list implementation of the List and Deque interfaces. This is also unsynchronized. Set: A Set is a Collection that cannot contain duplicate elements.
  - HashSet : Uses hashtable (actually a HashMap instance) to store elements. It does not guarantee that the order of elements will remain constant.

- TreeSet : Its NavigableSet implementation .The elements are ordered using their natural ordering, or by a Comparator provided at set creation time. Results into Sorted set of elements.
- Map: It is a key-value pair type. A Map cannot contain duplicate keys; each key can map to at most one value.
- HashMap :The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. It does not guarantee that the order of items will remain constant over time.
- TreeMap:A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. It is unsynchronized.

About Comparable & Comparator

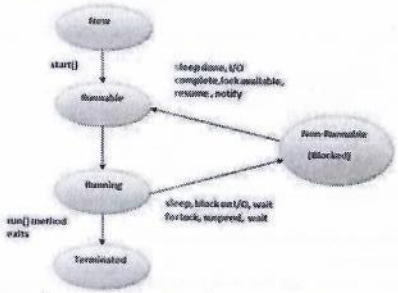
- We generally use Collections.sort() method to sort a simple array list. However if the Array List is of custom object type then in such case you have two options for sorting- comparable and comparator interfaces.
- Since Comparable is implemented by the same class whose objects are sorted so it binds code with that sorting logic which is ok in most of the cases but in case you want to have more than way of sorting your class objects you should use comparators.
-

Threading in Java

- When you want application to provide the most responsive interaction with the user, even if the application is currently doing other work we use multithreading.
- Using multiple threads of execution is one of the most powerful ways to keep your application responsive to the user and at the same time make use of the processor in between or even during user events.

Thread Life Cycle

- *New*: The thread is in new state if youcreate an instance of Thread class but before the invocation of start() method.
- *Runnable*: The thread is in runnable stateafter invocation of start() method, but the thread scheduler has not selected it to be the running thread.
- *Running:* The thread is in running state ifthe thread scheduler has selected it.
- *Non-Runnable (Blocked)* : This is the statewhen the thread is still alive, but is currently not eligible to run.
- *Terminated:* A thread is in terminated ordead state when its run() method exits.



Thread Synchronization

- Synchronization enables you to control program flow and access to shared data for concurrently executing threads. Synchronization can be achieved by mutex locks, read/write locks.
- Any code written by using synchronized block or enclosed inside synchronized method will be mutually exclusive, and can only be executed by one thread at a time.
- Synchronized keyword in Java is used to provide mutually exclusive access to a shared resource with multiple threads in Java. Important Point to be noted for synchronization.

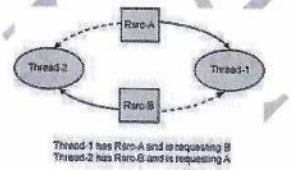
- You can use java synchronized keyword only on synchronized method or synchronized block.Whenever a thread enters into java synchronized method or blocks it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. The lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
- Java Thread acquires an object level lock when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.Java Synchronization will throw NullPointerException if object used in java synchronized block is null
- One Major disadvantage of Java synchronized keyword is that it doesn't allow concurrent read, which can potentially limit scalability.
- One more limitation of java synchronized keyword is that it can only be used to control access to a shared object within the same JVM. If you have more than one JVM and need to synchronize access to a shared file system or database, the Java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.
- Java synchronized keyword incurs a performance cost. A synchronized method in Java is very slow and can degrade performance. So use synchronization in java when it absolutely requires and consider using java synchronized block for synchronizing critical section only.
- Java synchronized block is better than java synchronized method in Java because by using synchronized block you can only lock critical section of code and avoid locking the whole method which can possibly degrade performance.

Use of join method in threadsjava.lang.Thread class provides the join() method which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program.join() method puts the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.

Deadlock

In Java, a deadlock is a situation where minimum two threads are holding the lock on some different resource, and both are waiting for other's resource to complete its task. And, none is able to leave the lock on the resource it is holding.

In given figure Thread-1 has A but need B to complete processing and similarly Thread-2 has resource Bbut need A first.



InterThread Communication using wait(), notify() and notifyAll() methods

The Object class in Java has three final methods that allow threads to communicate about the locked status of a resource.

- wait() : It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method
- notify(): It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread

can wake up. However, the lock is not actually given up until the notifier's synchronized block has completed.

notifyAll(): It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify()method above.

Generics and Collections

The generic collections **disable** the **type-casting** and there is no use of **type-casting** when it is used in generics. The generic collections are **type-safe** and checked at **compile-time**. These generic collections allow the datatypes to pass as parameters to classes. The **Compiler** is responsible for checking the **compatibility** of the types.

Syntax
class<type>, interface<type>

Type safety

Generics allows a single type of object.

```
List list = new ArrayList(); // before generics
list.add(10);
list.add("100");
List<Integer> list1 = new ArrayList<Integer>(); // adding generics
list1.add(10);
list1.add("100"); // compile-time error.
```

Type Casting

No need for type-casting while using generics.

```
List<String> list = new ArrayList<String>();
list.add("Adithya");
String str = list.get(0); // no need of type-casting
```

Compile-time

The errors are checked at **compile-time** in generics.

```
List list = new ArrayList(); // before generics
list.add(10);
list.add("100");
List<Integer> list1 = new ArrayList<Integer>(); // adding generics
list1.add(10);
list1.add("100");// compile-time error
```

Functional Interfaces

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable, ActionListener, Comparable** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called @FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

java.util.function package : Predicate, Map, Consumer, Supplier

The Functional Interface PREDICATE is defined in the java.util.function package. It improves manageability of code, helps in unit-testing them separately, and contain some methods like:

isEqual(Object targetRef) : Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

static Predicate isEqual(Object targetRef)
Returns a predicate that tests if two arguments are
equal according to Objects.equals(Object, Object).
T : the type of arguments to the predicate
Parameters:
targetRef : the object reference with which to
compare for equality, which may be null
Returns: a predicate that tests if two arguments
are equal according to Objects.equals(Object, Object)

and(Predicate other) : Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

default Predicate and(Predicate other)
Returns a composed predicate that represents a
short-circuiting logical AND of this predicate and another.
Parameters:
other: a predicate that will be logically-ANDed with this predicate
Returns : a composed predicate that represents the short-circuiting
logical AND of this predicate and the other predicate
Throws: NullPointerException - if other is null

negate() : Returns a predicate that represents the logical negation of this predicate.

default Predicate negate()
Returns:a predicate that represents the logical
negation of this predicate

or(Predicate other) : Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

default Predicate or(Predicate other)
Parameters:
other : a predicate that will be logically-ORed with this predicate
Returns:
a composed predicate that represents the short-circuiting
logical OR of this predicate and the other predicate
Throws : NullPointerException - if other is null

test(T t) : Evaluates this predicate on the given argument.boolean test(T t)

test(T t)

Parameters:
t - the input argument
Returns:
true if the input argument matches the predicate, otherwise false

Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

(*parameter1*, *parameter2*) -> *expression*

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

(*parameter1*, *parameter2*) -> { *code block* }

Using Lambda Expressions

Lambda expressions are usually passed as parameters to a function:

Example

Use a lambda expression in the ArrayList's forEach() method to print every item in the list:

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

Example

Use Java's Consumer interface to store a lambda expression in a variable:


```
import java.util.ArrayList;
import java.util.function.Consumer;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

Example

Create a method which takes a lambda expression as a parameter:

```
interface StringFunction {
  String run(String str);
}

public class Main {
  public static void main(String[] args) {
    StringFunction exclaim = (s) -> s + "!";
    StringFunction ask = (s) -> s + "?";
    printFormatted("Hello", exclaim);
    printFormatted("Hello", ask);
  }
  public static void printFormatted(String str, StringFunction format) {
    String result = format.run(str);
    System.out.println(result);
  }
}
```

Files

The File class from the java.io package, allows us to work with files.

To use the File class, create an object of the class, and specify the filename or directory name:

ExampleGet your own Java Server

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not

createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

Example

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

File created: filename.txt

Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

Example

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
```



```
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

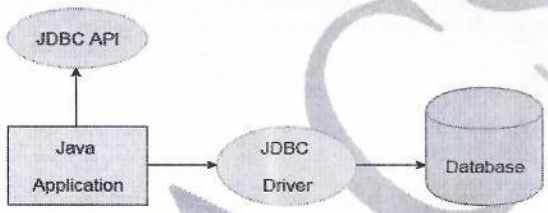
The output will be: Successfully wrote to the file.

Introduction of JDBC API

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



Creating a simple JDBC application

```
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
```