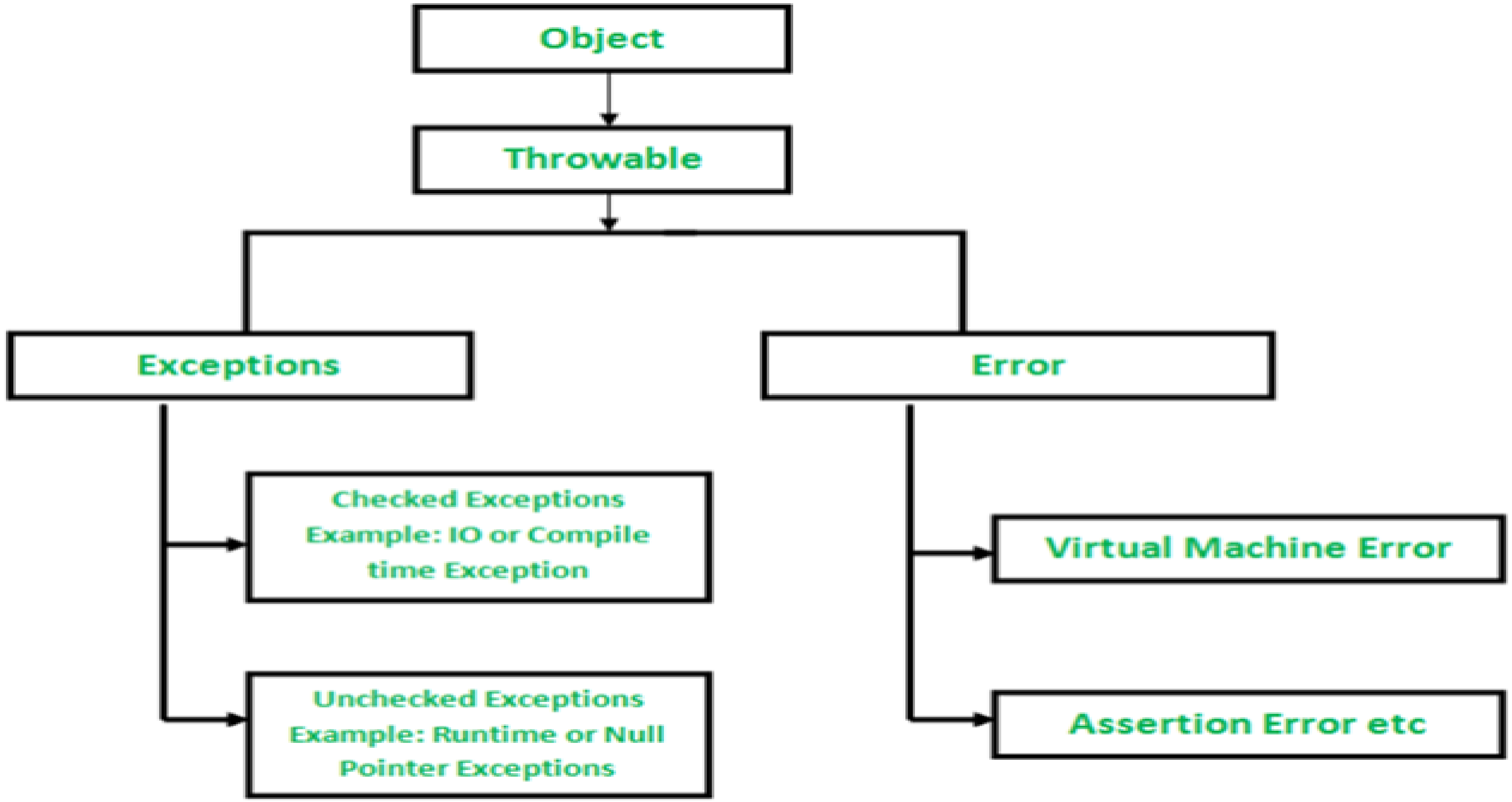
Unit 2: Concurrent Programming

Exception Keywords - Try, catch, finally, throw, throws, Creating User defined Exceptions, Working with Thread class and the Runnable interface, Thread priorities, File handling with java, File stream, File connection methods, JDBC architecture, Types of drivers, Java.sql package, Establishing connectivity and working with connection interface

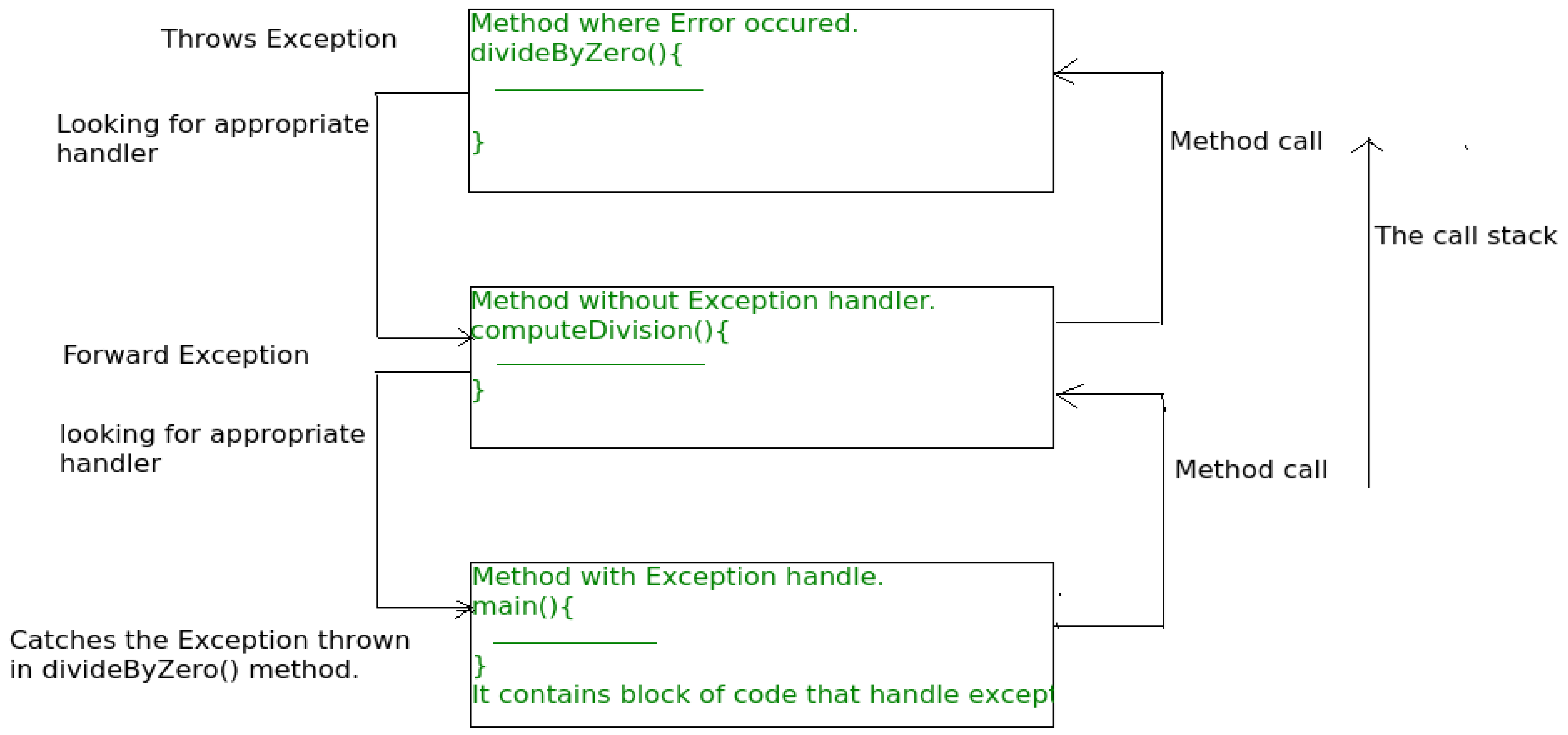

Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Definition

- **Error:** An Error indicates serious problem that a reasonable application should not try to catch. Compile time – syntax and semantic
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Default Exception Handling

- Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM).
- The exception object contains name and description of the exception, and current state of the program where exception has occurred.
- Creating the Exception Object and handling it to the run-time system is throwing an Exception.
- There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called Call Stack.
- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called Exception handler.
- The run-time system starts searching from the method in which exception occurred proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn’t have found the appropriate handler then run-time system handover the Exception Object to default exception handler, which is part of run-time system. This handler prints the exception information in the following format and terminates program abnormally.


Throws Exception
Looking for appropriate handler
Forward Exception
looking for appropriate handler
Catches the Exception thrown in divideByZero() method.
Method where Error occured.
divideByZero(){
}
Method without Exception handler.
computeDivision(){
}
Method with Exception handle.
main(){
}
It contains block of code that handle except
Method call
Method call
The call stack

The call stack and searching the call stack for exception handler.

Exception handling

- An *exception* is an abnormal condition that arises in a code sequence at run time. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- **Customized Exception Handling:** Java exception handling is managed via five keywords **try**, **catch**, **throw**, **throws**, and **finally**. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.
- This is the general form of an exception-handling block:

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

Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for Debugging.
- Doing so provides two benefits.
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.

- Displaying a Description of an Exception
- Throwable overrides the toString( ) method (defined by Object) so that it returns a string containing a description of the exception.

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

Nested try Statements

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

throw

- However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:
- throw *ThrowableInstance*;
- Here, *ThrowableInstance* must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
- using a parameter in a catch clause, or creating one with the new operator.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

- Java’s builtin run-time exceptions have at least two constructors:
  - one with no parameter and one that takes a string parameter.
  - When the second form is used, the argument specifies a string that describes the exception.
  - This string is displayed when the object is used as an argument to print( ) or println( ).
  - It can also be obtained by a call to getMessage( ), which is defined by Throwable.

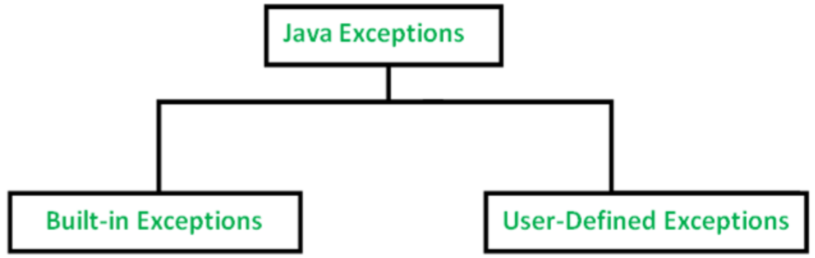
throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:
- *type method-name*(*parameter-list*) throws *exception-list*
- {
- // body of method
- }
- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

throw	throws
Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code.	Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code.
Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple exception with throw keyword.	On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used.
With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw.	On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name.
Syntax wise throw keyword is followed by the instance variable.	On other hand syntax wise throws keyword is followed by exception class names.
In order to use throw keyword we should know that throw keyword is used within the method.	On other hand throws keyword is used with the method signature.

finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.
- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.



- The circumstances that prevent execution of the code in a finally block are:
  - The death of a Thread
  - Using of the System exit() method.
  - Due to an exception arising in the finally block.

Finally and Close()

- **close()** statement is used to close all the open streams in a program. Its a good practice to use close() inside finally block.
- Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

User-Defined Exceptions

the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.
Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
    {          super(str);       }
```

- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

```
MyException me = new MyException(“Exception details”);
     throw me;
```

Using Exceptions

- Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program’s logic.
- One last point: Java’s exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

Checked vs Unchecked Exceptions

- Checked: are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
  In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Here are the few other Checked Exceptions –

- SQLException
- IOException
- ClassNotFoundException
- InvocationTargetException

Here are the few unchecked exception classes:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException
- NumberFormatException

Should we make our exceptions checked or unchecked?

- *If a client can reasonably be expected to recover from an exception, make it a checked exception.*
- *If a client cannot do anything to recover from the exception, make it an unchecked exception.*

Multithreading

Java thread model, Life Cycle of Thread, Working with Thread class and the Runnable interface, Thread priorities

Thread

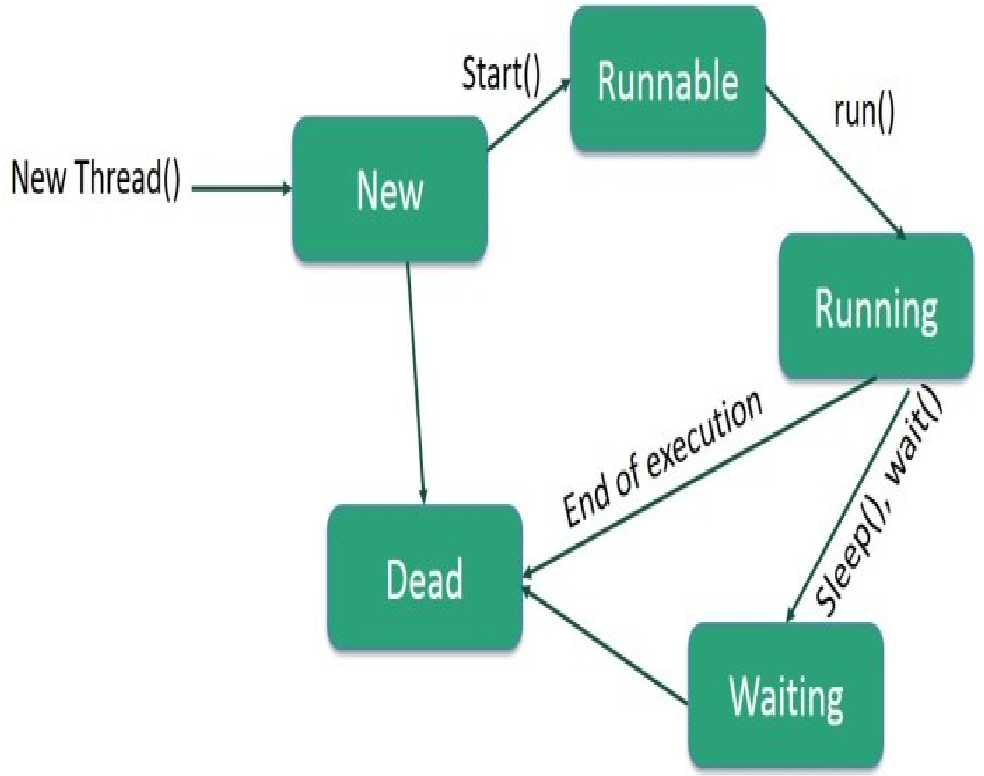
- Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. there are two distinct types of multitasking: process based and thread-based.
- A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a *thread-based* multitasking environment, the thread is the smallest unit of dis-patchable code. This means that a single program can perform two or more tasks simultaneously.
- Process-based multitasking deals with the "big picture," and thread-based multitasking handles the details. Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces.

Interthread Communication

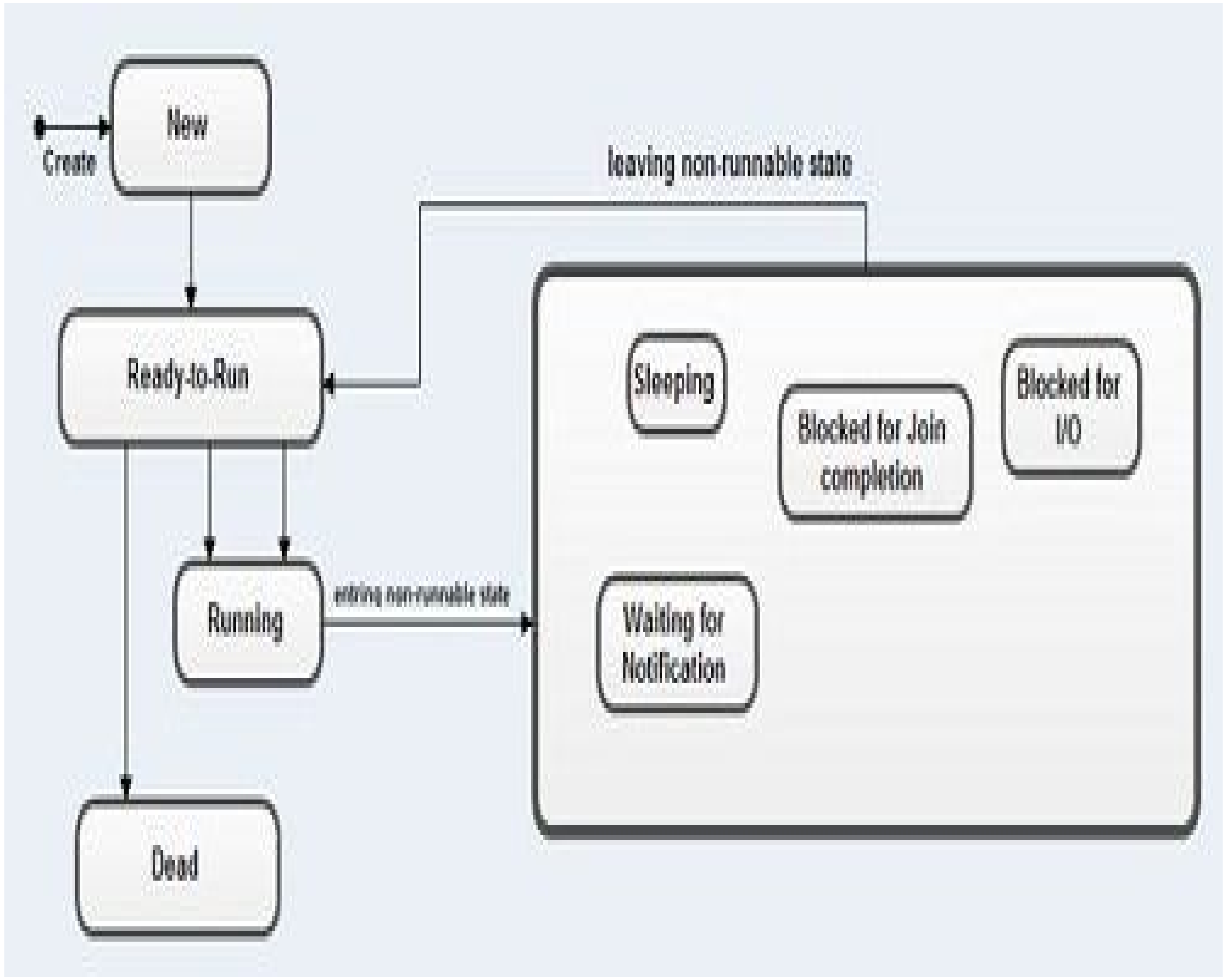
- multithreading replaces event loop programming by dividing your tasks into discrete, logical units.
- Threads also provide a secondary benefit: they do away with polling.
- Polling is usually implemented by a loop that is used to check some condition repeatedly.
- Once the condition is true, appropriate action is taken. This wastes CPU time.
- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.
- These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

Inter-process communication Vs Thread

- Inter-process communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- Single-threaded systems use an approach called an *event loop* with *polling.*



New Thread()
New
Start()
Runnable
run()
Running
End of execution
sleep(), wait()
Dead
Waiting



Create
New
Ready-to-Run
leaving non-runnable state
Sleeping
Blocked for Join completion
Blocked for I/O
Running
entering non-runnable state
Waiting for Notification
Dead

stages of the life cycle

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

context switch

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. A higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.*
- *A thread can voluntarily surrender control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not y ield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*
- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Messaging

- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java’s messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.
- Java’s multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. To create a new thread, your program will either extend Thread or implement the Runnable interface.

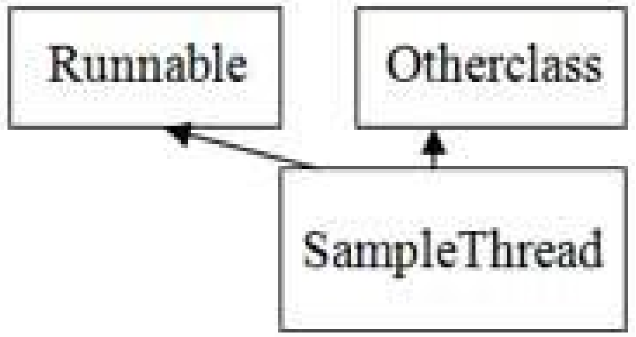
The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:
    - It is the thread from which other “child” threads will be spawned.
    - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
- *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, t is again output. This time, the new name of the thread is displayed.

Creating a Thread

- In the most general sense, you create a thread by instantiating an object of type Thread.
- Java defines two ways in which this can be accomplished:
- You can implement the Runnable interface: Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this: public void run( )
  - Inside run( ), you will define the code that constitutes the new thread.
  - It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.
- You can extend the Thread class, itself.

```
OtherClass implements Runnable
{
 ......
 public void run ()
 {
   //code to be run when this thread is started
 }
}
```



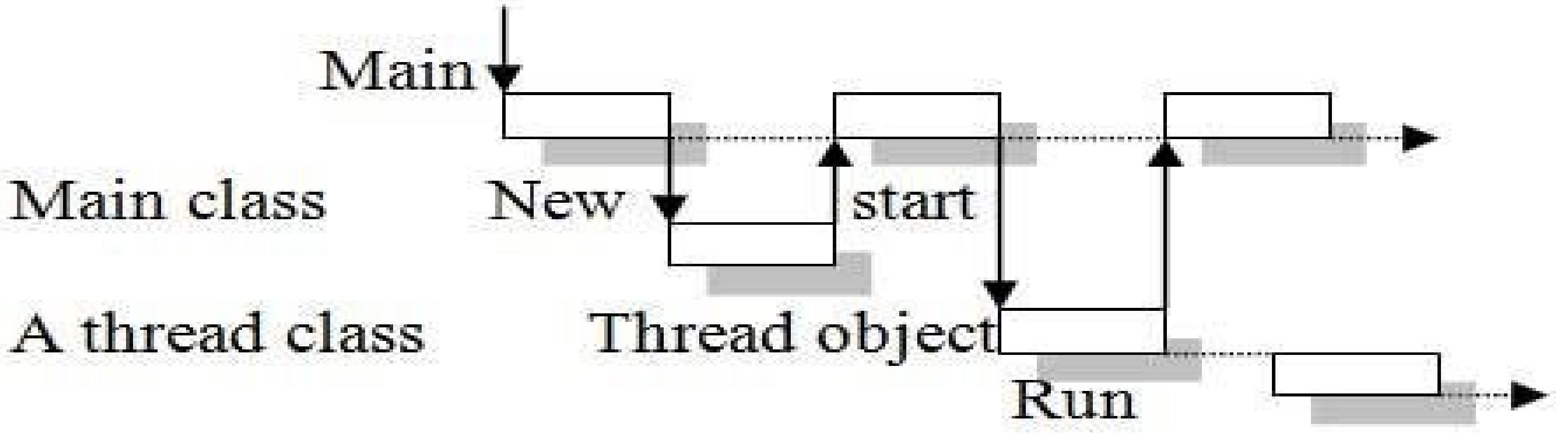


Fig. Running the 'main' thread and newly created thread.

Choosing an Approach

- The Thread class defines several methods that can be overridden by a derived class.
- Of these methods, the only one that *must* be overridden is run( ).
- This is, of course, the same method required when you implement Runnable.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- **Thread Class vs Runnable Interface**
  1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

  2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Thread Methods

- Using isAlive( ) and join( )
- Two ways exist to determine whether a thread has finished.
- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
- Join()method waits until the thread on which it is called terminates.

Thread Priority

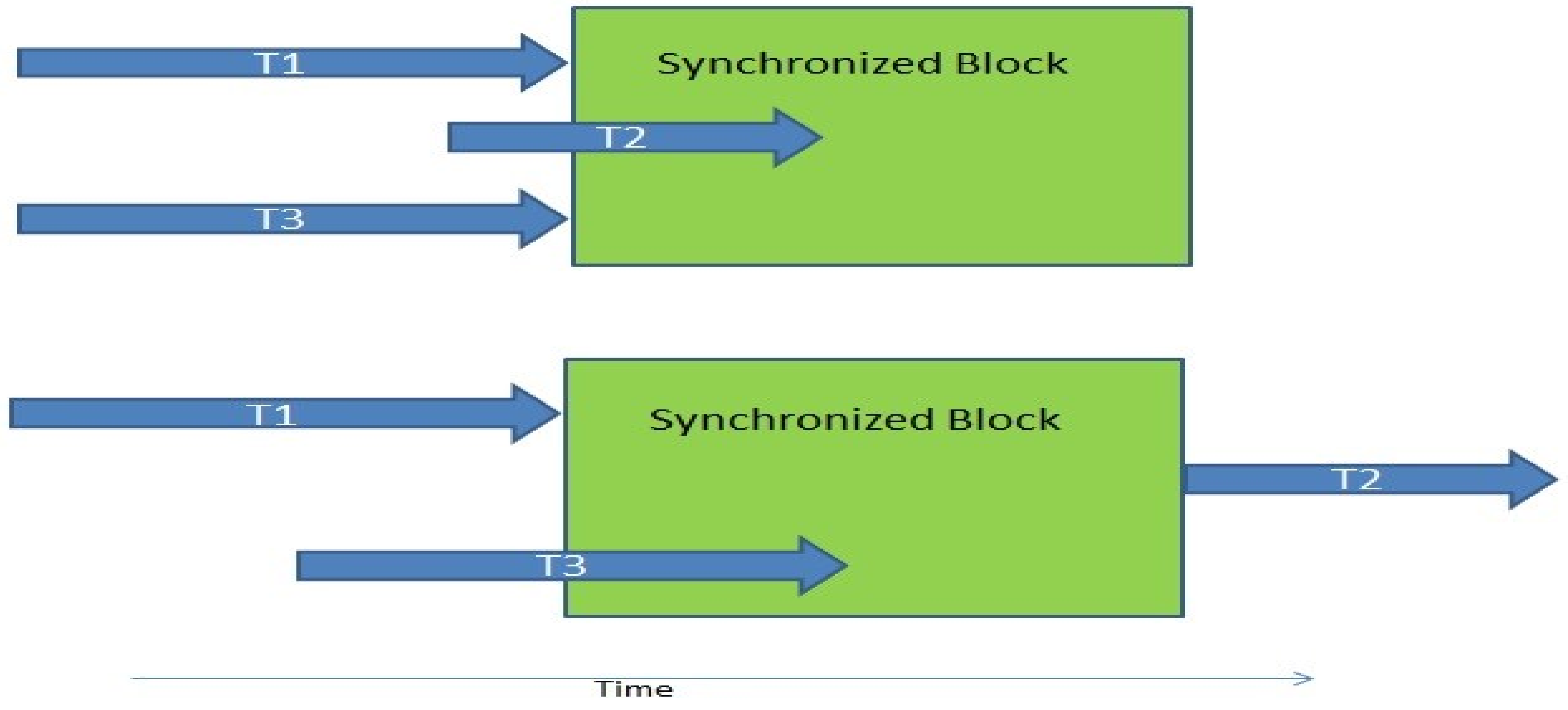
- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes, it will preempt the lower priority thread.
- In theory, threads of equal priority should get equal access to the CPU.
- Java is designed to work in a wide range of environments. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a non-preemptive operating system.
- In practice, even in non-preemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run.

Priority of a Thread (Thread Priority)

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY
```

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.



In Synchronized Block, other threads will have to wait when one thread is in

Advantages of Java Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.

S.N.	Modifier and Type	Method	Description
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

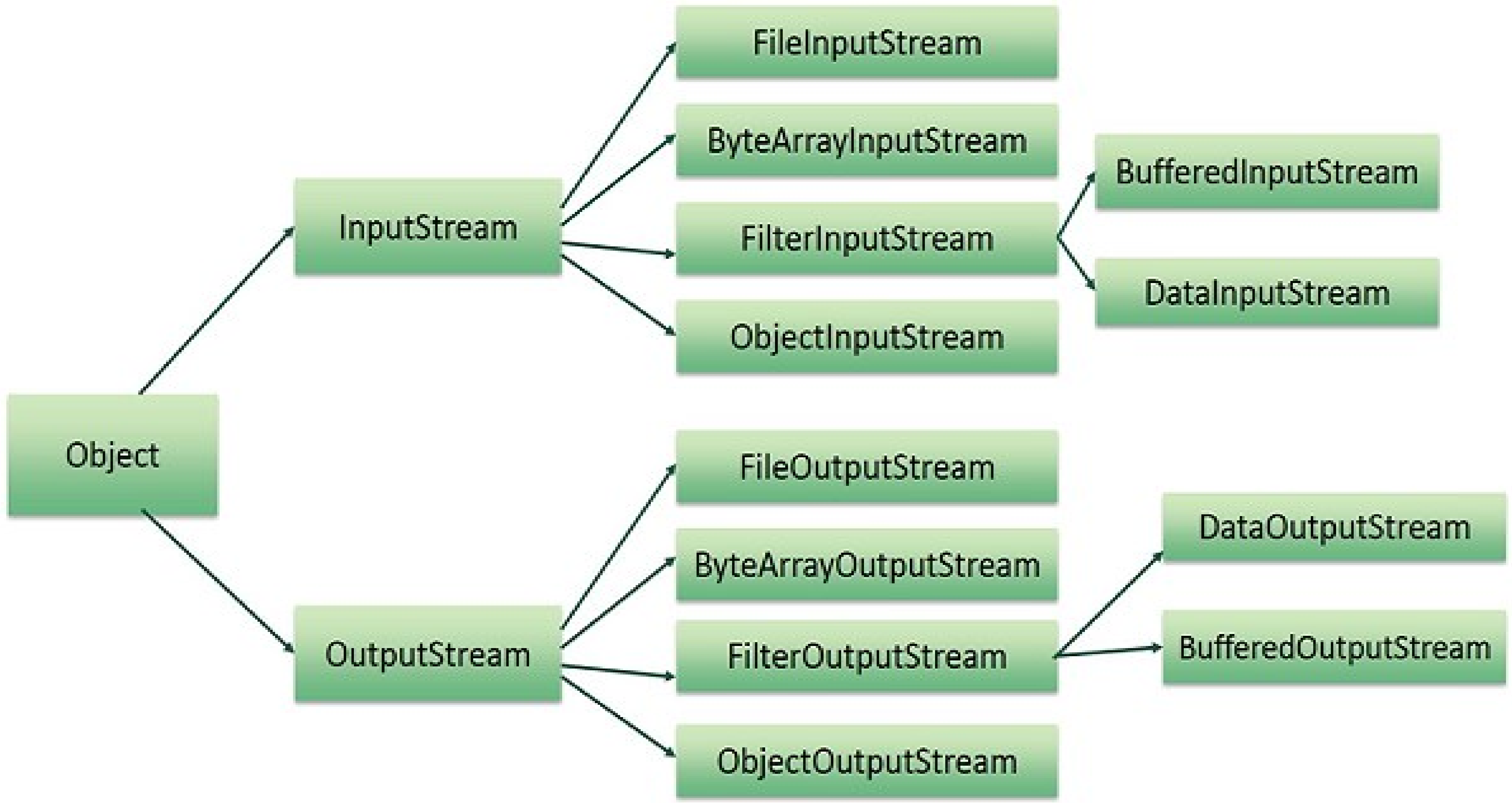
What if we call run() method directly instead start() method

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```java
class TestCallRun1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

File handling

Input streams and Output streams, FileInputStream and FileOutputStream, Binary and Character streams, Buffered Reader/ Writer, Object serialization and Deserialization


Object
InputStream
OutputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream

Stream

A stream is linked to a physical device by the Java I/O. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io.package.

- System.out: standard output stream
- System.in: standard input stream
- System.err: standard error stream

Difference between input/output Streams and Readers/Writers

- The major difference between these is that the input/output stream classes read/write **byte stream** data. Whereas the Reader/Writer classes **handle characters**.
- The methods of input/output stream classes **accept byte array as parameter** whereas the Reader/Writer classes accept **character array as parameter**.
- The Reader/Writer classes handles all the **Unicode characters**, comes handy for internalization, comparatively efficient that input/output streams.
- Therefore, until you deal with binary data like images it is recommended to use Reader/Writer classes.
- read( ) and write( ), which, respectively, read and write bytes of data. Both methods are declared as abstract inside InputStream and OutputStream. They are overridden by derived stream classes.

When to use Character Stream over Byte Stream?

- In Java, characters are stored using **Unicode conventions**. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream?

- Byte **oriented reads byte by byte**. A byte stream is suitable for processing raw data like binary files.

- FileStream
- Java FileOutputStream is an output stream used for writing data to a file.
- If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

- Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

- Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. For adding the buffer in an OutputStream, use the BufferedOutputStream class.
- Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.
- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

- Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class

- BufferedReader reads a couple of characters from the specified stream and stores it in a buffer. This makes input faster.
- InputStreamReader reads only one character from specified stream and remaining characters still remain in the stream.

```
class NewClass{
public static void main(String args[]) throws InterruptedException, IOException{
 BufferedReader isr = new BufferedReader(new InputStreamReader(System.in));
Scanner sc = new Scanner(System.in);
System.out.println("B.R. - "+(char)isr.read());
System.out.println("Scanner - " + sc.nextLine());
 }}
```

- Both *BufferedReade*r and *BufferedWriter* in java are classified as buffered I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.
- For unbuffered I/O stream, each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive. This means that instead of using *BufferedReader*, one can use the *Scanner* class, and instead of using *BufferedWriter*, one can use *PrintWriter.*
- *BufferedReader* is a class in Java that reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, lines and arrays. The buffer size may be specified. If not, the default size, which is predefined, may be used.

```
FileReader reader = new FileReader(“MyFile.txt”);
BufferedReader bufferedReader = new BufferedReader(reader);
```

- will buffer the input from the specified file. Without buffering, each invocation of *read()* or *readLine()* could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.
- A *BufferedWriter* on the other hand is a java class that writes text to a character-output stream, while buffering characters so as to provide for the efficient writing of single characters, strings and arrays.
- A *newLine()* method is provided, which uses the platform’s own notion of line separator as defined by the system property line.separator. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.

- Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.
- BufferedReader has a buffer, but by itself needs to take the characters from another Reader, given to the constructor using the decorator pattern common in java.io. The most common is the Java SDK InputStreamReader which converts characters from an InputStream which provides the bytes.

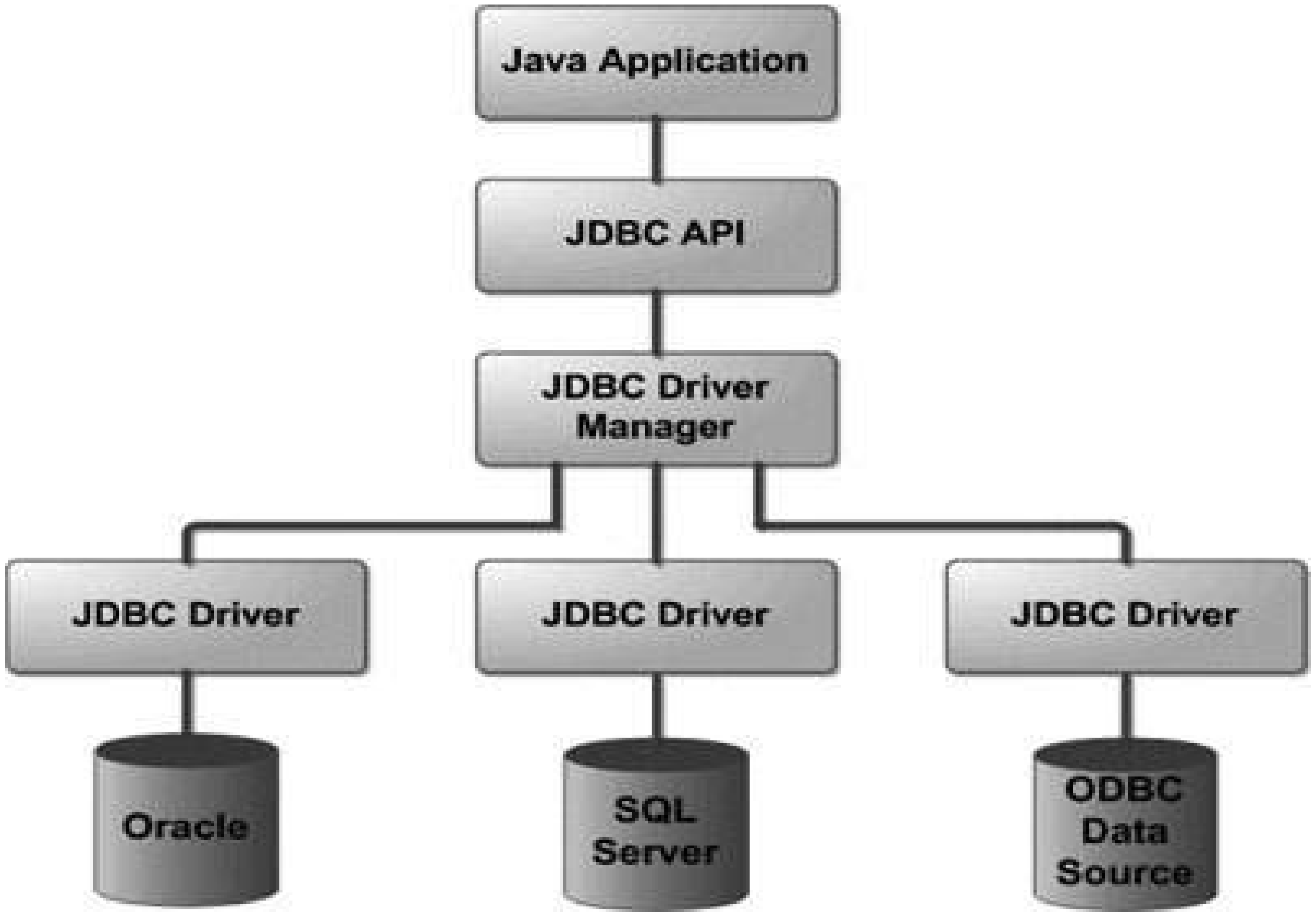
  The buffer allows a useful feature of BufferedReader, reading lines separated by an end-of-line delimiter.

  BufferedWriter does the opposite and writes, though it does not provide a "write line" method.
- BufferedWriter is almost similar to FileWriter but it uses internal buffer to write data into File. So if the number of write operations are more, the actual IO operations are less and performance is better. You should use BufferedWriter when number of write operations are more.

Unit 4: **Database Programming**

JDBC architecture, Types of drivers, Java.sql package , Establishing connectivity and working with connection interface, Working with statement interface, Working with PreparedStatement interface , Working with ResultSet interface,Working with ResultSetMetaData interface.

JDBC Architecture



We can use JDBC API to handle database using Java program and can perform the following activities:

- Connect to the database
- Execute queries and update statements to the database
- Retrieve the result received from the database.

Types of JDBC architecture

- **Two-tier model:** A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.
  The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the user’s machine acts as a client and the machine having the data source running acts as the server.
- **Three-tier model:** In this, the user’s queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.
  This type of model is found very useful by management information system directors.

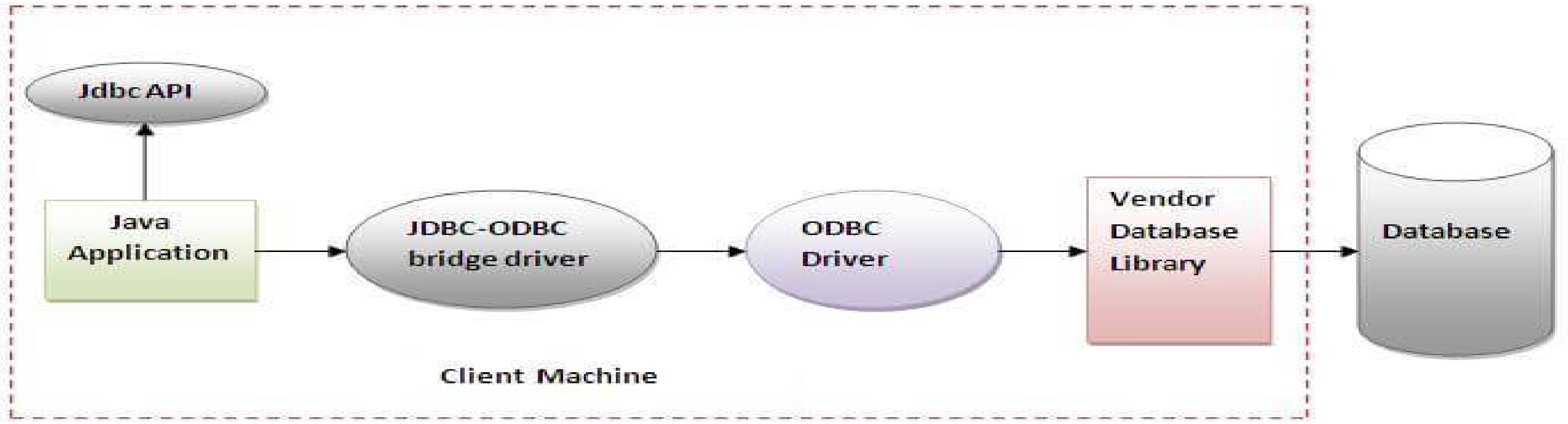
Components of JDBC

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

JDBC driver

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



Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.
- **In Java 8, the JDBC-ODBC Bridge has been removed**

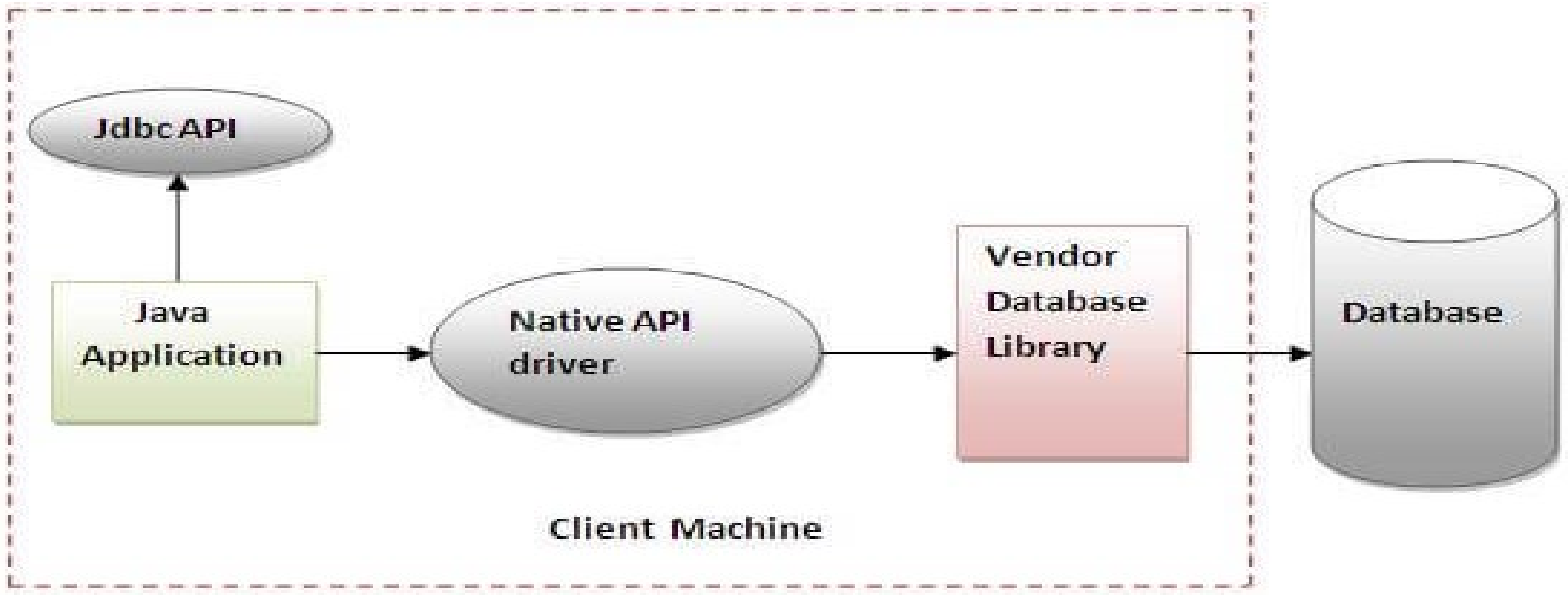
2. Native-API driver (partially java driver)

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.



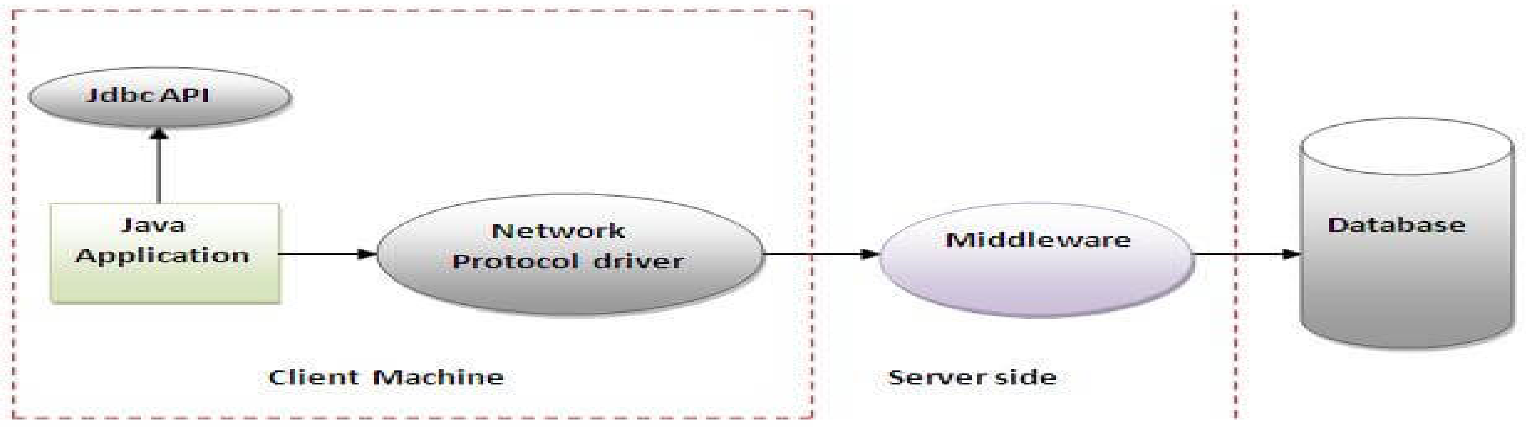
3. Network Protocol driver (fully java driver)

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier



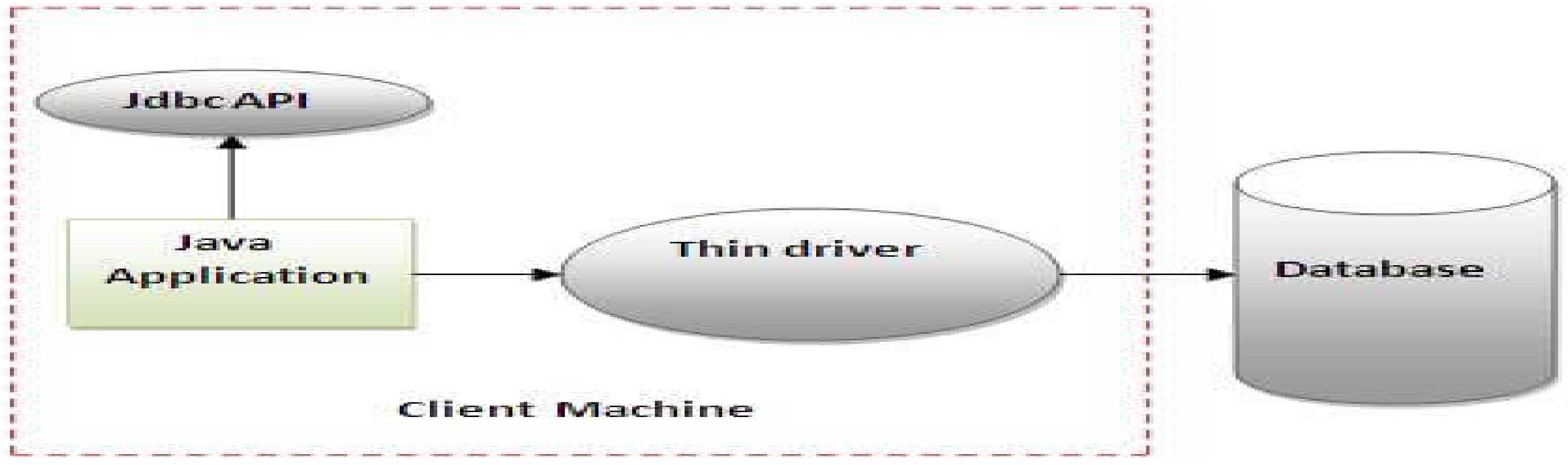
4. Thin driver (fully java driver)

Advantage:

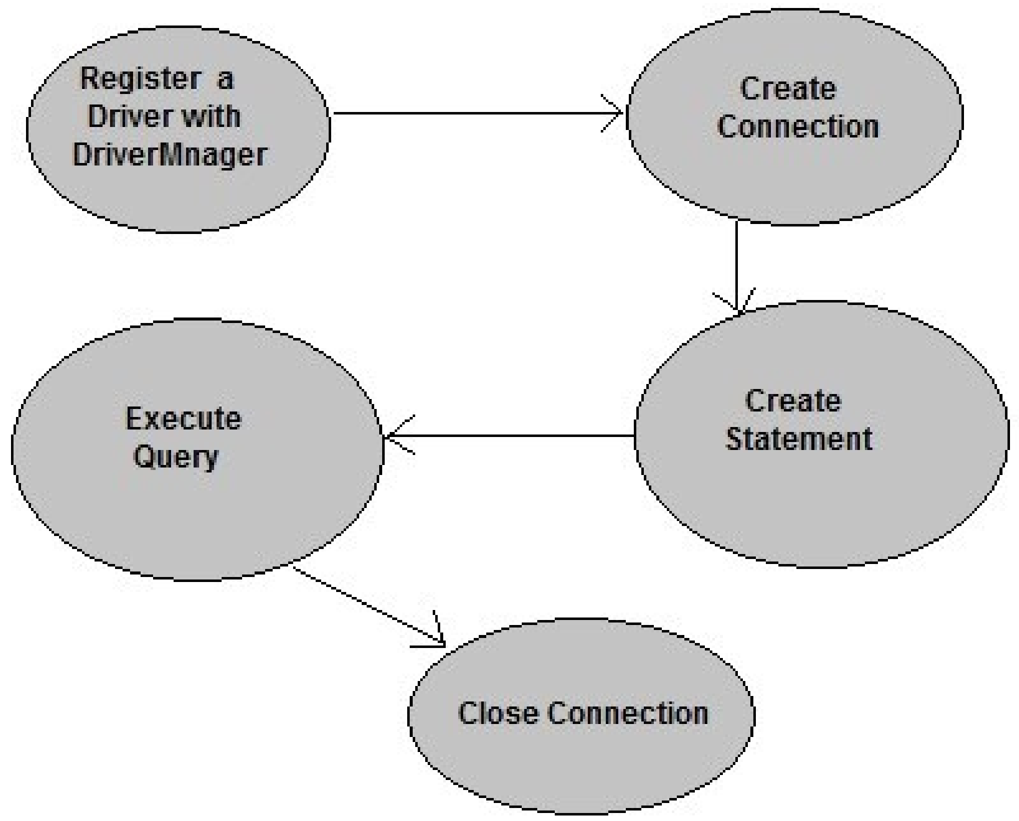
- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.



- Register the Driver
- Create a Connection
- Create SQL Statement
- Execute SQL Statement
- Closing the connection



Database Connectivity with 5 Steps

1. Register the Driver class

- The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class

public static void forName(String className) **throws** ClassNotFoundException

Class.forName("oracle.jdbc.driver.OracleDriver");

- DriverManager.registerDriver(): DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time. example uses DriverManager.registerDriver()to register the Oracle driver

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())

- The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

2. Create connection

- A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.
- The **getConnection()** method of DriverManager class is used to establish connection with the database.

public static Connection getConnection(String url)**throws** SQLException

public static Connection getConnection(String url, String name, String password) **throws** SQLException

{Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system","password");}

- user – username from which your sql command prompt can be accessed.
  password – password from which your sql command prompt can be accessed.
- con: is a reference to Connection interface.
  url : Uniform Resource Locator. It can be created as follows:
- String url = “ jdbc:oracle:thin:@localhost:1521:xe”
- @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider.

3. Create SQL Statement.

Statement: Used to implement simple SQL statements with no parameters
You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set.
You need a Connection object to create a Statement Object

Statement is an interface under java.sql package and my understanding is that it is not possible to create an instance of an interface in Java

The driver's implementation of Connection has an implementation of the createStatement method that returns the driver's implementation of Statement.

There are three different kinds of statements

a. Create statement

- The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database

```
public Statement createStatement()throws SQLException

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:152
1:xe","system","oracle");

Statement stmt=con.createStatement();

int result=stmt.executeUpdate("delete from emp765 where id=33");

System.out.println(result+" records affected");
```

b. PreparedStatement(Extends Statement)

It is used for precompiling SQL statements that might contain input parameters. This statement gives you the flexibility of supplying arguments dynamically

```
String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL)
```

- passing parameter (?) for the values will be set by calling the setter methods of PreparedStatement
- **Improves performance**: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.
- The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

public PreparedStatement prepareStatement(String query)**throws** SQLException{}

```
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");
int i=stmt.executeUpdate();
System.out.println(i+" records inserted");
```

c. CallableStatement: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters

```
 DROP PROCEDURE IF EXISTS `EMP`.`getEmpName`
CREATE PROCEDURE `EMP`.`getEmpName` (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
  SELECT first INTO EMP_FIRST FROM Employees WHERE ID = EMP_ID;
END
```

- Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three

```
String SQL = "{call getEmpName (?, ?)}";
cstmt = conn.prepareCall (SQL);
```

- Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.
- If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.
- When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return

Commonly used methods of Connection interface

1) public Statement createStatement(): **creates a statement object that can be used to execute SQL queries.**

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): **Creates a Statement object that will generate ResultSet objects with the given type and concurrency**

3) public void setAutoCommit(boolean status): **is used to set the commit status. By default it is true.**

4) public void commit(): **saves the changes made since the previous commit/rollback permanent.**

5) public void rollback(): **Drops all changes made since the previous commit/rollback**

6) public void close(): **closes the connection and Releases a JDBC resources immediately**
Statement interface

4. Execute SQL Statement

After creating statement, now execute using executeQuery() method of Statement interface. This method is used to execute SQL statements. Syntax of the method is given below.

```
public ResultSet executeQuery(String query) throws SQLException
```

Example:

```
ResultSet rs=s.executeQuery("select * from user");
  while(rs.next())
  {
   System.out.println(rs.getString(1)+" "+rs.getString(2));
 }
```

Statement Interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

- Commonly used methods of Statement interface:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands

ResultSet interface

- The ResultSet interface can be three categories −
- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

Type of ResultSet

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Concurrency of ResultSet

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

```
Statement stmt = conn.createStatement( ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
```

createStatement(int RSType, int RSConcurrency);
prepareStatement(String SQL, int RSType, int RSConcurrency);
prepareCall(String sql, int RSType, int RSConcurrency);

If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Execute queries

- To execute a query, call an execute method such as the following:
- execute: Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
- executeQuery: 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.
- executeUpdate: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

Viewing a Result Set

- The ResultSet interface contains dozens of methods for getting the data of the current row. There is a get method for each of the possible data types, and each get method has two versions −
- One that takes in a column name.
- One that takes in a column index.

S.N.	Methods & Description
1	public int getInt(String columnName) throws SQLException Returns the int in the current row in the column named columnName.
2	public int getInt(int columnIndex) throws SQLException Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a Result Set

- The ResultSet interface contains a collection of update methods for updating the data of a result set.
  - One that takes in a column name.
  - One that takes in a column index.

S.N.	Methods & Description
1	public void updateString(int columnIndex, String s) throws SQLException Changes the String in the specified column to the value of s.
2	public void updateString(String columnName, String s) throws SQLException Similar to the previous method, except that the column is specified by its name instead of its index.

5. Close connection

- Close connection By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection
- **public void** close()**throws** SQLException

Example

- con.close();

ResultSet interface

- The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row. **By default, ResultSet object can be moved forward only and it is not updatable**
- But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:
- Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIV E,ResultSet.CONCUR_UPDATABLE);