REVA UNIVERSITY
Bengaluru, India

M23DE0202 – Object Oriented Programming using JAVA

II Semester MCA Academic Year : 2024 - 2025

School of Computer Science and Applications

Pinaka Pani. R
Assistant Professor

Unit III

Object Oriented Programming using JAVA

Access of member variables

Modifier	Same Class	Same Package	Subclass (other pkg)	Other Packages
public	YES	YES	YES	YES
protected	YES	YES	YES	NO
default	YES	YES	NO	NO
private	YES	NO	NO	NO

Packages, Interfaces, Multithreading and Exception Handling

- Creation of User defined Packages, Importing Packages, Accessing Inbuilt Packages.
- Introduction to Interfaces, Features of Interfaces, Creation of Interfaces.
- Introduction to Multithreading, Life Cycle of a Thread, Creation of a Thread.
- Exception Handling, Creation of User Defined Exceptions, Understanding the keywords in Java – try, catch, throw, throws and finally, Examples on Exception Handling.

Defining a Package

A **Package** in Java is a namespace that organizes classes and interfaces.

Packages help in avoiding name conflicts, controlling access, and making it easier to locate and use classes, interfaces, and sub-packages.

A package is both a naming and a visibility control mechanism:

1) Divides the name space into disjoint subsets.

It is possible to define classes within a package that are not accessible by code outside the package.

2) Controls the visibility of classes and their members.

It is possible to define class members that are only exposed to other members of the same package.

Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

Creating a Package

- A package statement inserted as the first line of the source file:

  package myPackage;

  class MyClass1 { … }

  class MyClass2 { … }

- means that all classes in this file belong to the myPackage package.
- The package statement creates a name space where such classes are stored.
- When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

- Other files may include the same package instruction:
  1. package myPackage;
     class MyClass1 { … }
     class MyClass2 { … }
  2. package myPackage;
     class MyClass3{ … }
- A package may be distributed through several source files.

Packages and Directories

- Java uses file system directories to store packages.
- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The byte code files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Directory names must match package names exactly.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:
  1) Unix: myPackage1/myPackage2/myPackage3
  2) Windows: myPackage1\myPackage2\myPackage3
  3) Macintosh: myPackage1:myPackage2:myPackage3
- You cannot rename a package without renaming its directory!

Accessing a Package

- As packages are stored in directories, how does the Java run-time system know where to look for packages?
- Two ways:

  1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.

  2) Specify a directory path or paths by setting the CLASSPATH environment variable.

CLASSPATH Variable

- **CLASSPATH** –If the package is available in **different** directory, in that case the **compiler should be given** information regarding the **package location** by mentioning the **directory name** of the **package** in **classpath**.
- The CLASSPATH is an **environment variable** that tells the java **compiler** where to **look** for the **class files to import**. If our package exists in **e:\sub** then we need to set **classpath** as follows:
- D:java> **set CLASSPATH=e:\sub;.;%CLASSPATH%**
- **environment variable** that points to the root directory of the system's package hierarchy.
- Several **root directories** may be specified in CLASSPATH.
- Java will **search for the required packages** by looking up subsequent directories described in the **CLASSPATH** variable.

- **classpath** in Java is a parameter that tells the Java Virtual Machine (JVM) and Java compiler where to look for user-defined classes and packages when running or compiling Java programs.

Example: Package

- Save, compile and execute:
  1) call the file AccountBalance.java
  2) save the file in the directory MyPack
  3) compile; AccountBalance.class should be also in MyPack
  4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
  5) run: java MyPack.AccountBalance
- Make sure to use the package-qualified class name.

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type **long dot-separated** names.
- The import statement allows to use **classes** or **whole packages** directly.
- **Importing** of a **concrete class**:

  import myPackage1.myPackage2.myClass;
- **Importing** of **all classes within** a **package**:

  import myPackage1.myPackage2.*;

Import Statement

- The **import statement** occurs immediately after the **package** statement and **before** the **class** statement:

  package myPackage;

- import otherPackage1;otherPackage2.otherClass;

  class myClass { … }

- The Java system accepts this **import statement** by default:

  import java.lang.*;

- This package includes the **basic language functions**. Without such functions, Java is of no much use.

Example: Packages 1

- A package MyPack with one public class Balance.

  The class has two same-package variables: public constructor and a public show method.

```
package MyPack;
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n; bal = b;
}
public void show() {
if (bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}
}
```

Example: Packages 2

The importing code has access to the public class Balance of the MyPack package and its two public members:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show();
}
}
```

Java Source File

- Java source file consists of:

1) a single package instruction (optional)
2) several import statements (optional)
3) a single public class declaration (required)
4) several classes private to the package (optional)

- At the minimum, a file contains a single public class declaration.

Defining an interface

- An **interface** in Java is a reference type, **similar to a class**, that can contain only **constants, method signatures (abstract methods), default methods, static methods** etc...
- Interfaces **cannot contain any instance fields or constructors**.
- They serve as a **blueprint** for classes and are used to achieve **full abstraction, multiple inheritance.**
- Using interface, we specify **what a class must d**o, but **not how it does this.**
- An interface is defined with an **interface** keyword.

- Interface is little bit like a class, but interface is lack in instance variables. Hence, we can't create object for it.
- Interfaces are developed to support multiple inheritance.
- The methods present in interfaces are pure abstract.
- The access specifiers public, private, protected are possible with classes, but the interface uses only one specifier public.
- interfaces contains only the method declarations, no definitions.
- An interface defines, which method a class has to implement. If you want to call a method defined by an interface - you don't need to know the exact class type of an object, you only need to know that it implements a specific interface.
- A class can implement multiple interfaces.

Defining an Interface

- An interface **declaration** consists of **modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body**.
- **For example:**

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
// constant declarations double E = 2.718282;
// base of natural logarithms //
//method signatures
 void doSomething (int i, double x);
 int doSomethingElse(String s);
 }
```

- **The public access specifier indicates** that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.
- **An interface can extend other interfaces,** just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends

Differences between classes and interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.

Implementing interface

- General format:

access interface name {

type method-name1(parameter-list);

type method-name2(parameter-list);

…

type var-name1 = value1;

type var-nameM = valueM;

…

}

- **Two types of access**:

  1) **public** – interface may be used anywhere in a program

  2) **default** – interface may be used in the current package only

- Interface methods have **no body** – they end with the semicolon after the parameter list.
- They are essentially **abstract methods**.
- An interface may **include variables**, but they must be **final, static and initialized with a constant value.**
- In a public interface, all members are **implicitly public**.

Interface Implementation

- A class implements an interface if it provides a complete set of methods defined by this interface.

  1) any number of classes may implement an interface.

  2) one class may implement any number of interfaces.

- Each class is free to determine the details of its implementation.
- Implementation relation is written with the implements keyword.

Implementation Format

- General format of a class that includes the implements clause:
- Syntax:

  access class_name extends super-class implements interface1, interface2, …, interfaceN {

  …

  }

- Access is public or default.

Implementation Comments

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Applying interfaces

A Java *interface* declares a set of method signatures i.e., says what behavior exists Does not say how the behavior is implemented

i.e., does not give code for the methods

- **Does not describe any state (but may include “final” constants)**

- A concrete class that implements an interface Contains “implements *InterfaceName*” in the class declaration.
- Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface.
- An abstract class can also implement an interface.
- Can optionally have implementations of some or all interface methods.

- Interfaces and Extends both describe an “is- a” relation
- If B *implements* interface A, then B inherits the (abstract) method signatures in A
- If B *extends* class A, then B inherits everything in A,
- which can include method code and instance variables as well as abstract method signatures
- Inheritance” is sometimes used to talk about the superclass/subclass “extends” relation only

variables in interface

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:

  1) declare an interface with variables initialized to the desired values.

  2) include that interface in a class through implementation.
- As no methods are included in the interface, the class does not implement anything except importing the variables as constants.

extending interfaces

- One interface may inherit another interface.
- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
void myMethod2(…) ;
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Interface Inheritance 1

- **Consider interfaces A and B**

```
interface A {
void meth1();
void meth2();
}
```

B extends A:

```
interface B extends A {
void meth3();
}
```

Example: Interface Inheritance 2

- MyClass must implement all of A and B methods:

```
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```

Importance of Interfaces in Java:

- **Achieving Abstraction.**
- **Multiple Inheritance**.
- **Loose Coupling**: Interfaces help decouple code by separating the definition of a method from its implementation. This enables easier maintenance and scalability. Clients of the interface don't need to know the details of how a class implements its methods, only that it conforms to the interface.
- **Polymorphism:** Interfaces enable polymorphic behavior. Multiple classes can implement the same interface, allowing them to be treated uniformly. For example, different classes implementing the Runnable interface can be executed in separate threads using the same Thread class.
- **Testability.**
- **Design Flexibility**.

Collections

- **Collection<E>:** The root interface in the *collection hierarchy*.
- **Comparator<T>:** A comparison function, which imposes a *total ordering* on some collection of objects.
- **Enumeration<E>:** An object that implements the Enumeration interface generates a series of elements, one at a time.
- **EventListener:** A tagging interface that all event listener interfaces must extend.
- **Iterator<E>:** An iterator over a collection
- **List<E>**An ordered collection (also known as a *sequence*).
- **ListIterator<E>:** An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

- **Collections**: This class consists exclusively of static methods that operate on or return collections
- **Currency**: Represents a currency.
- **Date**: The class Date represents a specific instant in time, with millisecond precision.
- **Dictionary<K,V>**: The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
- **EventObject**: The root class from which all event state objects shall be derived.

- **GregorianCalendar**: GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
- **HashMap<K,V>**: Hash table based implementation of the Map interface.
- **HashSet<E>**: This class implements the Set interface, backed by a hash table (actually a HashMap instance)
- .**Hashtable<K,V>**: This class implements a hashtable, which maps keys to values.
- **LinkedList<E>**: Linked list implementation of the List interface
- **Locale**: A Locale object represents a specific geographical, political, or cultural region.
- **Observable**: This class represents an observable object, or "data" in the model-view paradigm
- **Properties**: The Properties class represents a persistent set of properties.

- **Random**: An instance of this class is used to generate a stream of pseudorandom numbers.
- **ResourceBundle**: Resource bundles contain locale-specific objects.
- **SimpleTimeZone**: SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
- **Stack<E>**: The Stack class represents a last-in-first-out (LIFO) stack of objects.
- **StringTokenizer**: The string tokenizer class allows an application to break a string into tokens.
- **TimeZone**: TimeZone represents a time zone offset, and also figures out daylight savings.
- **TreeMap<K,V>**: A Red-Black tree based NavigableMap implementation.
- **TreeSet<E>**: A NavigableSet implementation based on a TreeMap.**UUID**A class that represents an immutable universally unique identifier (UUID).
- **Vector<E>**: The Vector class implements a growable array of objects

Exception Summary

- **EmptyStackException:** Thrown by methods in the Stack class to indicate that the stack is empty.
- **InputMismatchException:** Thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.
- **InvalidPropertiesFormatException:** Thrown to indicate that an operation could not complete because the input did not conform to the appropriate XML document type for a collection of properties, as per the Properties specification.
- **NoSuchElementException:** Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.
- **TooManyListenersException:** The TooManyListenersException Exception is used as part of the Java Event model to annotate and implement a unicast special case of a multicast Event Source.
- **UnknownFormatConversionException:** Unchecked exception thrown when an unknown conversion is given.

Multithreading / Concurrency

- **Thread:** single sequential flow of control within a program
- **Single-threaded program can handle one task at any time.**
- **Multitasking allows single processor to run several concurrent threads.**
- **Most modern operating systems support multitasking.**
- A thread represents execution of statements or processing the statements. The process or execution is called a Thread.
- Internally ‘main’ thread is running by the JVM.

Java Threads

- Threads allows a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.

- Creating a Thread
- There are two ways to create a thread.

- It can be created by extending the Thread class and overriding its run() method:

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Another way to create a thread is to implement the Runnable interface:

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Running Threads

- If the class **extends** the **Thread** class, the thread can be run by creating an instance of the class and call its **start()** method.

- If the class **implements** the **Runnable** interface, the thread can be run by passing an instance of the class to a **Thread** object's constructor and then calling the thread's **start()** method.

Differences between "extending" and "implementing" Threads:

The major difference is that when a class **extends** the Thread class, you cannot extend any other class, but by implementing the **Runnable** interface, it is possible to **extend** from another class as well.

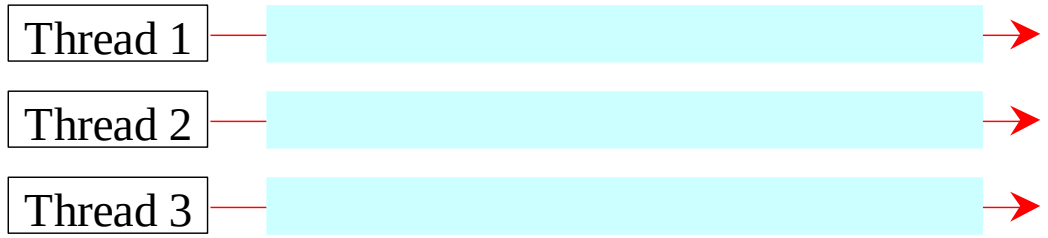
Ex: class MyClass extends OtherClass implements Runnable.

Advantages of Multithreading

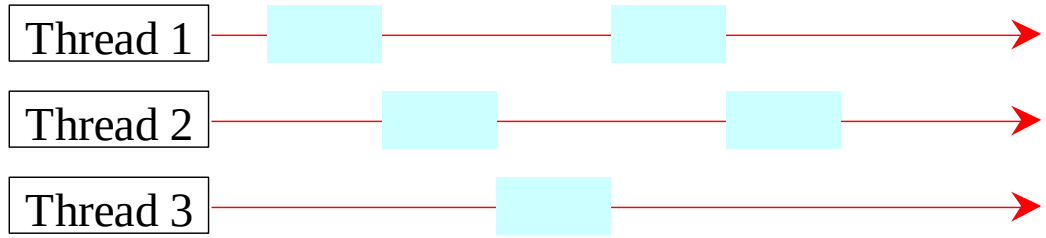
- Reactive systems – constantly monitoring
- More responsive to user input – GUI application can interrupt a time-consuming task
- Server can handle multiple clients simultaneously
- Can take advantage of parallel processing

Threads Concept

Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



Threads in Java

Creating threads in Java:

- Extend java.lang.Thread class

OR

- Implement java.lang.Runnable interface

Threads in Java

Creating threads in Java:

- Extend java.lang.Thread class
  - run() method must be overridden (similar to main method of sequential program).
  - run() is called when execution of the thread begins.
  - A thread terminates when run() returns.
  - start() method invokes run().
  - Calling run() does not create a new thread.

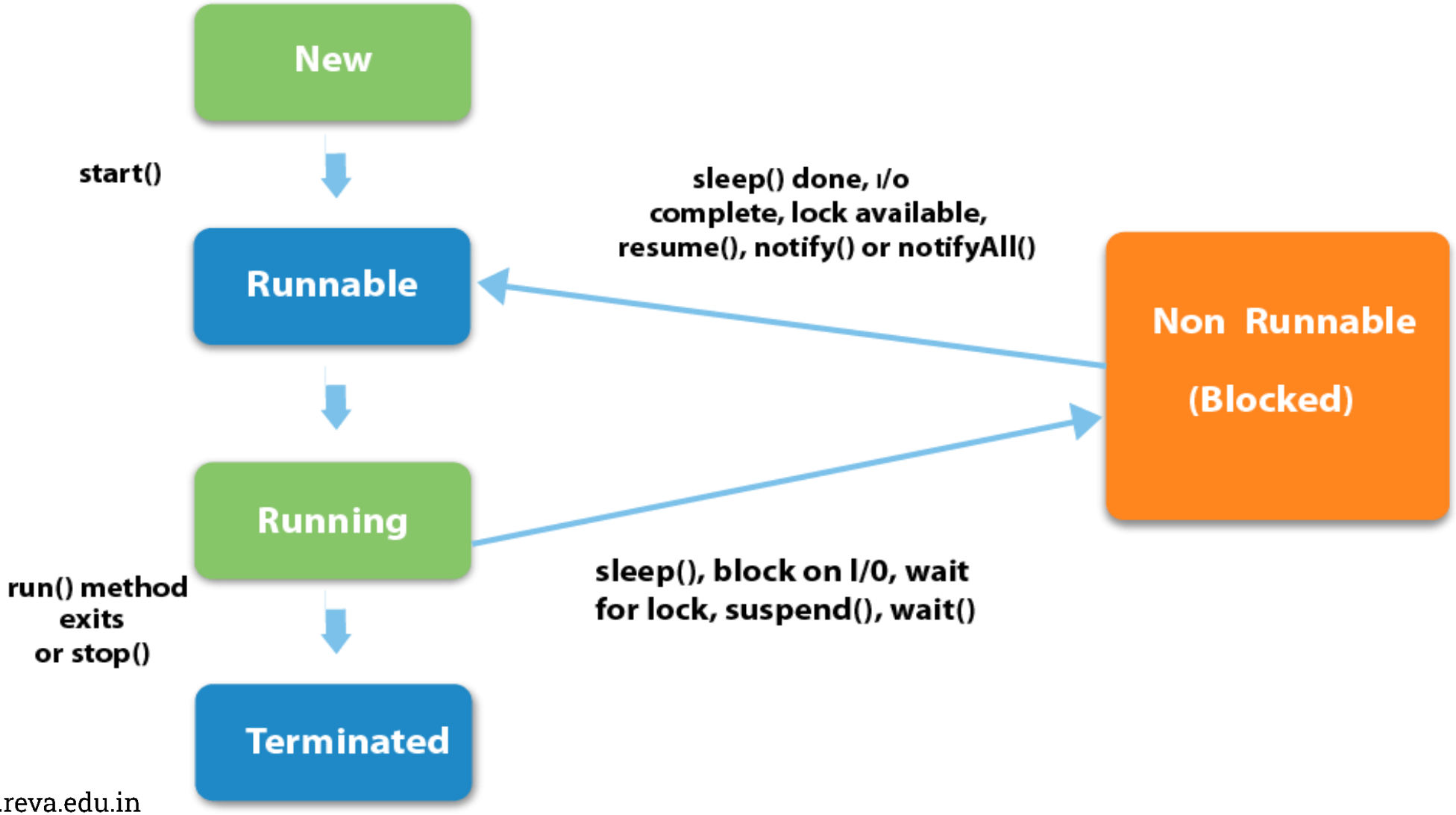
- Implement java.lang.Runnable interface

Thread lifecycle

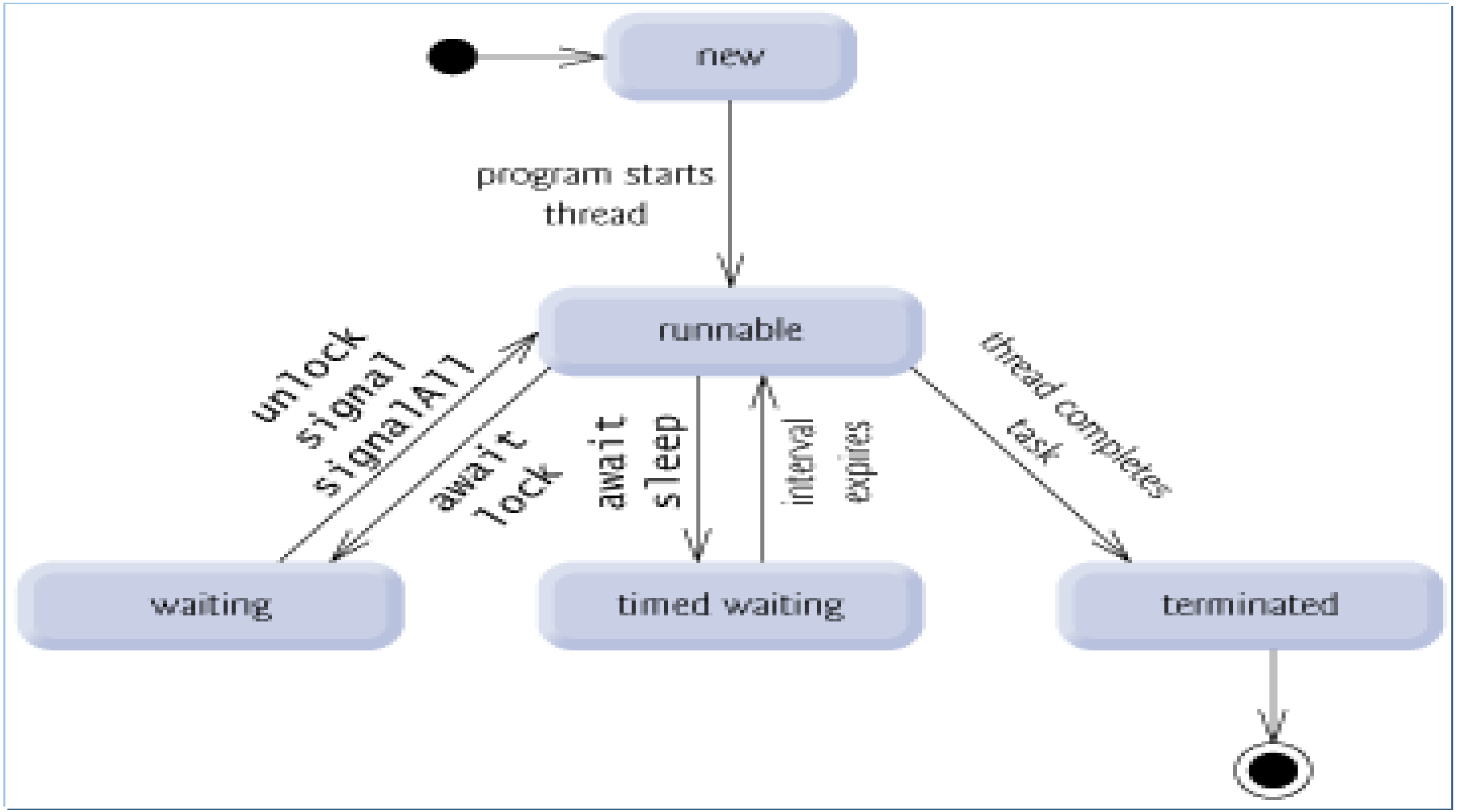
- The thread lifecycle represents the different stages a thread goes through from **creation to termination**.

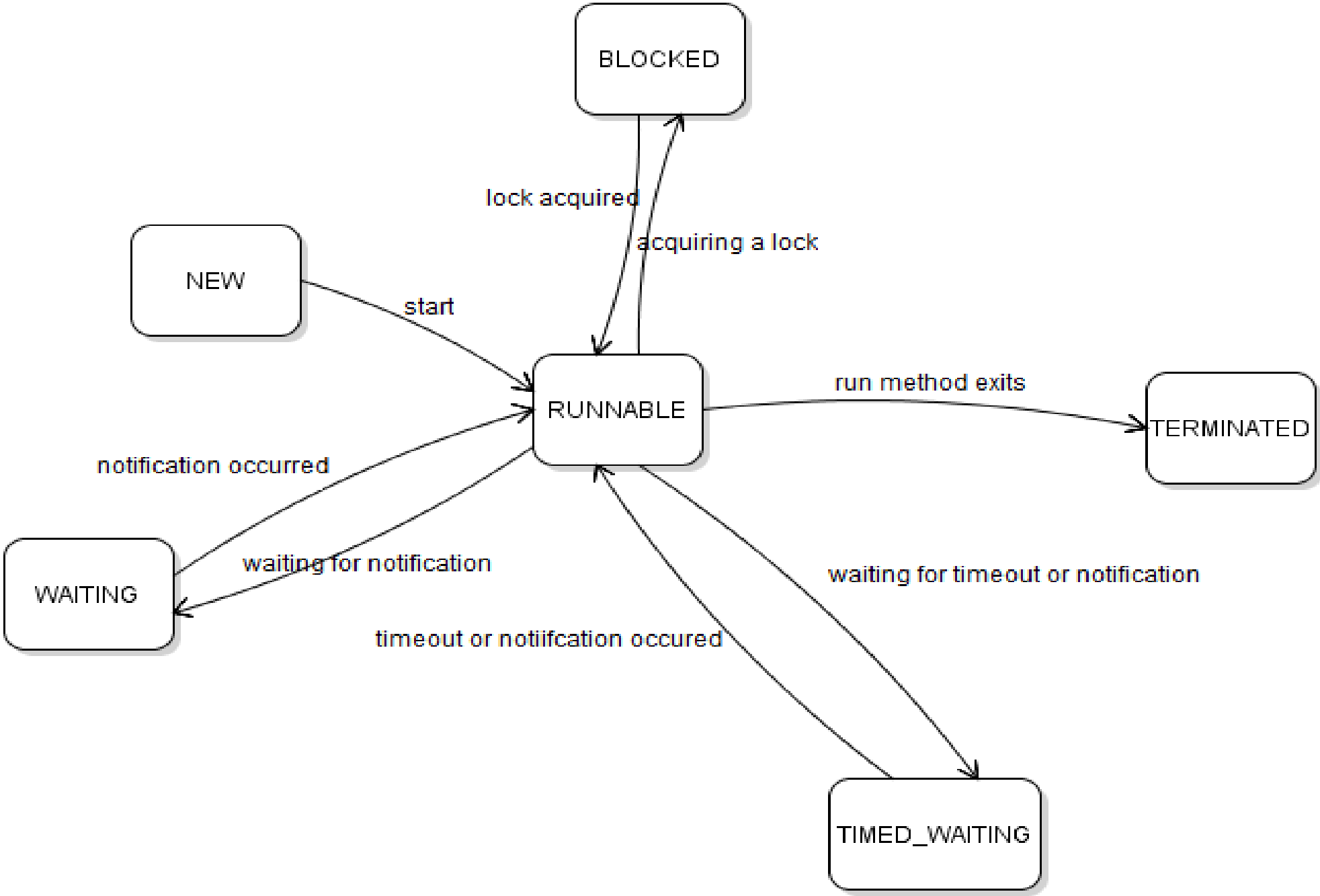
- Each thread in Java passes through **multiple states**.
- Understanding these states helps in managing and optimizing multithreaded applications.

States in the lifecycle of a Thread



Thread States




BLOCKED
lock acquired
acquiring a lock
NEW
start
RUNNABLE
run method exits
TERMINATED
notification occurred
WAITING
waiting for notification
waiting for timeout or notification
timeout or notiifcation occured
TIMED_WAITING

These enum constants are defined in the Thread.State enum. Let me explain each state in details.

NEW: when a thread is created but has not executed (the start() method has not been invoked), it is in the new state.

RUNNABLE: when the start() method has been invoked, the thread enters the runnable state, and its run() method is executing. Note that the thread can come back to runnable state from another state (waiting, blocked), but it may not be picked immediately by the thread scheduler, hence the term “runnable”, not running.

BLOCKED: when a thread tries to acquire an intrinsic lock (not a lock in the java.util.concurrent package) that is currently held by another thread, it becomes blocked. When all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold the lock, the thread becomes unblocked and enters the runnable state.

WAITING: a thread enters this state if it waits to be notified by another thread, which is the result of calling Object.wait() or Thread.join(). The thread also enters waiting state if it waits for a Lock or Condition in the java.util.concurrent package. When another thread calls Object‘s notify()/notifyAll() or Condition’s signal()/signalAll(), the thread comes back to the runnable state.

TIMED_WAITING: a thread enters this state if a method with timeout parameter is called: sleep(), wait(), join(), Lock.tryLock() and Condition.await(). The thread exits this state if the timeout expires or the appropriate notification has been received.

TERMINATED: a thread enters terminated state when it has completed execution. The thread terminates for one of two reasons:

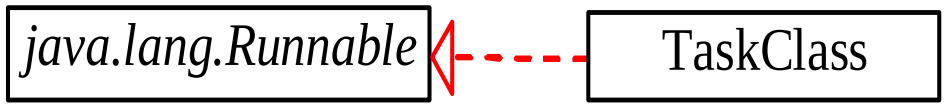
+ the **run()** method exits normally.
+ the **run()** method exits abruptly due to a uncaught exception occurs.

Thread termination

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

Creating Tasks and Threads



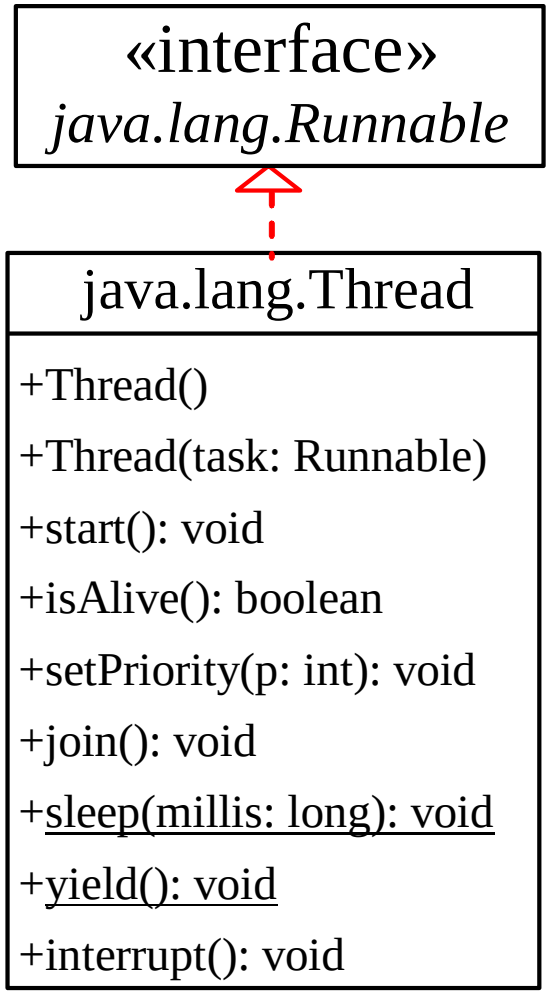

```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```



Creates a default thread.

Creates a thread for a specified task.

Starts the thread that causes the run() method to be invoked by the JVM.

Tests whether the thread is currently running.

Sets priority p (ranging from 1 to 10) for this thread.

Waits for this thread to finish.

Puts the runnable object to sleep for a specified time in milliseconds.

Causes this thread to temporarily pause and allow other threads to execute.

Interrupts this thread.

wait()

- The **wait**() method causes the current thread to **wait** until another thread invokes the **notify**() or **notifyAll**() methods for that object.
- The **notify**() method wakes up a single thread that is **waiting** on that object's monitor.
- The **notifyAll**() method wakes up all threads that are **waiting** on that object's monitor.

Continued…

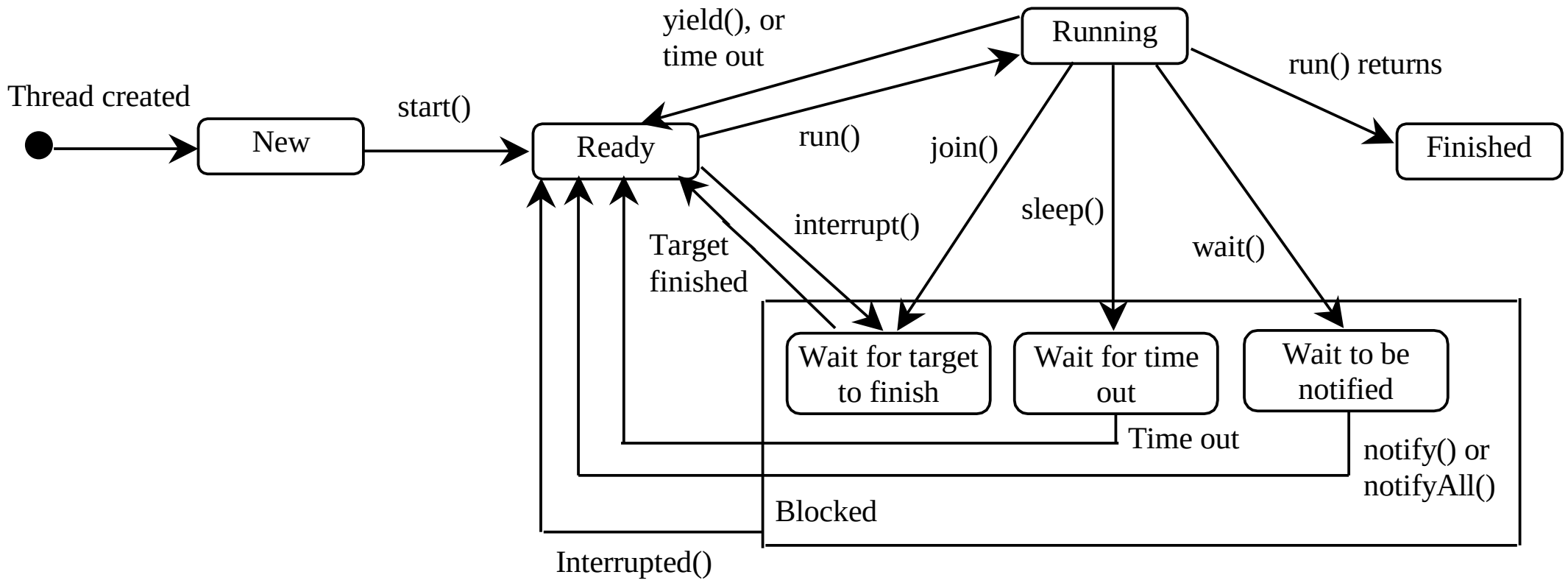
- The threads can communicate with each other through **wait(), notify()** and **notifyAll()** methods in Java.
- These are **final** methods defined in the **Object** class and can be called only from within a **synchronized** context.
- The **wait()** method causes the current thread to wait until another thread invokes the **notify()** or **notifyAll()** methods for that object.

Continued…

- The **notify()** method **wakes up a single thread** that is waiting on that object's monitor.
- The **notifyAll()** method **wakes up all threads** that are waiting on that object's monitor.
- A thread waits on an object's monitor by calling one of the **wait()** method.
- These methods can throw **IllegalMonitorStateException** if the current thread is not the owner of the object's monitor.

Thread States

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



Thread methods

isAlive()
- method used to find out the state of a thread.
- returns true: thread is in the Ready, Blocked, or Running state
- returns false: thread is new and has not started or if it is finished.

interrupt()
- If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

- The isInterrupt() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

NOTE:

- The Thread class also contains the stop(), suspend(), and resume() methods.
- As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe.
- You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

Thread Priority

- Each thread is assigned a default priority of Thread.NORM_PRIORITY (constant of 5). You can reset the priority using setPriority(int priority).
- Some constants for priorities include
- Thread.MIN_PRIORITY
- Thread.MAX_PRIORITY
- Thread.NORM_PRIORITY
- By default, a thread has the priority level of the thread that created it.

wait(), notify(), and notifyAll()

- Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

- The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Daemon thread

- Is a special type of thread that runs in the background and supports other non-daemon (user) threads.
- It is often used for low-priority tasks that do not affect the main application, like **garbage collection, monitoring, or housekeeping tasks that should not block the application from exiting.**
- **Runs in Background:** Daemon threads run in the background, performing tasks that support the main program without interfering with the application's core **functions.**

- **Terminates Automatically**: Daemon threads are automatically terminated by the Java Virtual Machine (JVM) when all non-daemon threads (also known as user threads) have completed execution.
- The JVM does not wait for daemon threads to finish their tasks before shutting down, so daemon threads should not perform critical or essential operations that must complete.

- **Use Cases:** They are often used for background support tasks, such as: Garbage collection, Thread pooling, Monitoring or logging, Scheduling and background maintenance

Concepts of exception handling

Exceptions

- Exception is an abnormal condition that arises when executing a program.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:
  1) provides syntactic mechanisms to signal, detect and handle errors
  2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
  3) brings run-time error management into object-oriented programming

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java?

Exception is an abnormal condition.

- An exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java?

Exception is an abnormal condition.

- An exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.

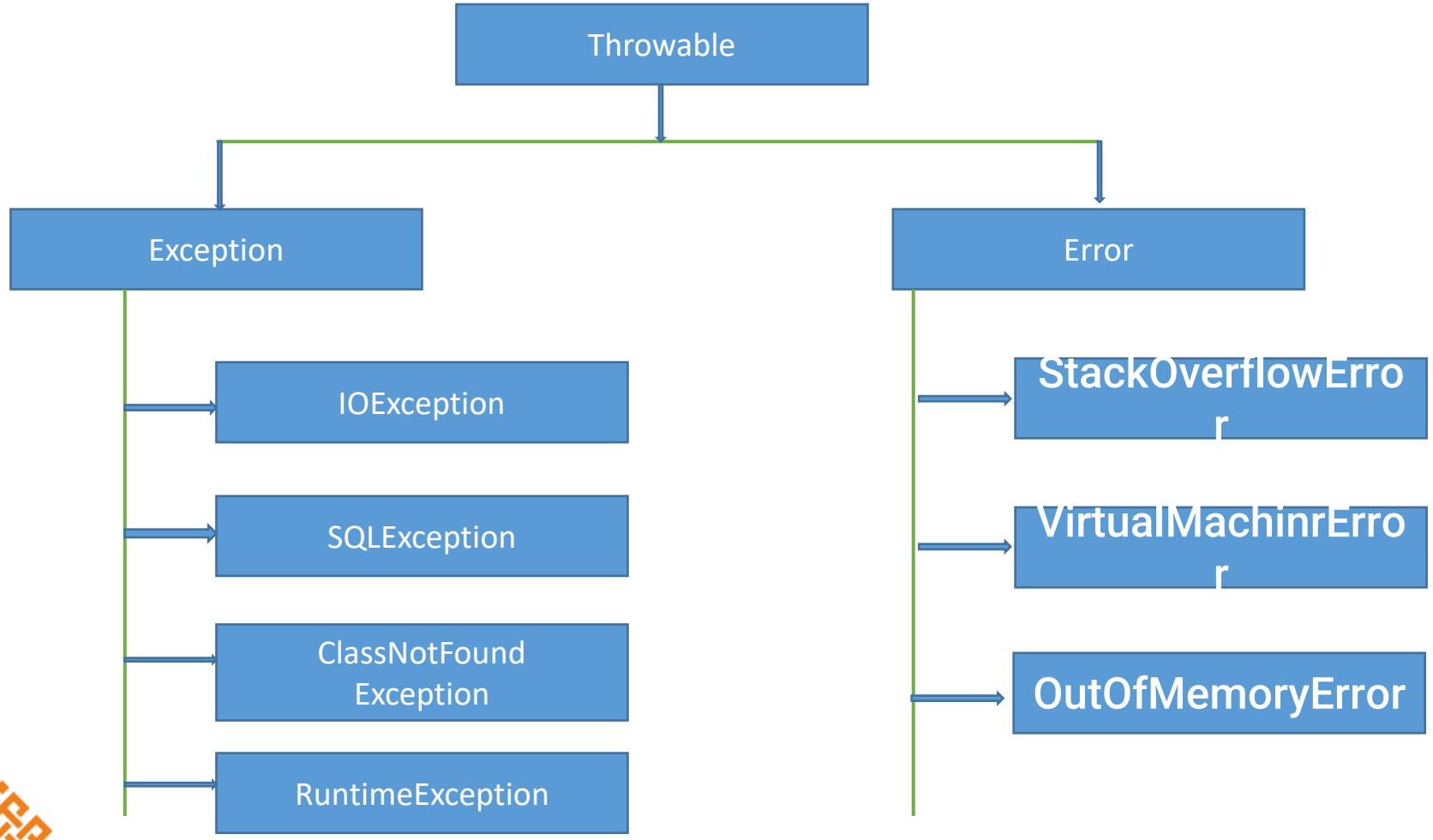
Exception Handling in Java

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc…

Advantage of Exception Handling

- To maintain the normal flow of the application.
- An exception normally disrupts the normal flow of the application that is why we use exception handling.

Hierarchy of Java Exception classes



Types of Java Exceptions

QUIZ

1) When does Exceptions in Java arises in code sequence?

A. Run Time

B. Compilation Time

C. Can Occur Any Time

D. None of the mention

Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

Difference between Checked and Unchecked Exceptions

2) Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions.
- e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Differences between throw and throws :

Aspect	throw	throws
Definition	Used to explicitly throw an exception within a method.	Declares that a method might throw one or more exceptions.
Purpose	To generate an exception during runtime.	To inform the compiler and callers about potential exceptions.
Syntax	throw new ExceptionType("message");	void methodName() throws ExceptionType1, ExceptionType2 {}
Placement	Inside a method or block of code.	In the method declaration after the parameter list.
Action	Directly triggers the exception.	Acts as a declaration, not a trigger.
Scope	Affects only the current execution flow.	Affects the method's contract with its caller.

Java Exception Handling Example

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

2) A scenario where NullPointerException occurs

3) A scenario where NumberFormatException occurs

4) A scenario where ArrayIndexOutOfBoundsException occurs

QUIZ

1) Which of these keywords must be used to monitor for exceptions?

A. try

B. finally

C. throw

D. catch

ANSWER: A

Java try block

- It is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute.
- it is recommended not to keeping the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Java try block

Syntax of Java try-catch

```
try{
//code that may throw an exc
eption
}catch(Exception_class_Name
 ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an except
ion
}finally{}
```

Java catch block

- It is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., Exception) o the generated exception type.
- The good approach is to declare the generated type of exception.
- The catch block must be used after the try block only.

- You can use multiple catch block with a single try block.

QUIZ

1) Which of these keywords must be used to handle the exception thrown by try block?

A. try

B. finally

C. throw

D. catch

ANSWER: D

Java Multi-catch block

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

Points to remember

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

Java catch block

- It is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., Exception) o the generated exception type.
- The good approach is to declare the generated type of exception.
- The catch block must be used after the try block only.

- You can use multiple catch block with a single try block.

QUIZ

1) Which of these keywords is used to manually throw an exception?

A. try

B. finally

C. throw

D. catch

ANSWER: C

Java Nested try block

- The try block within a try block is known as nested try block in java.

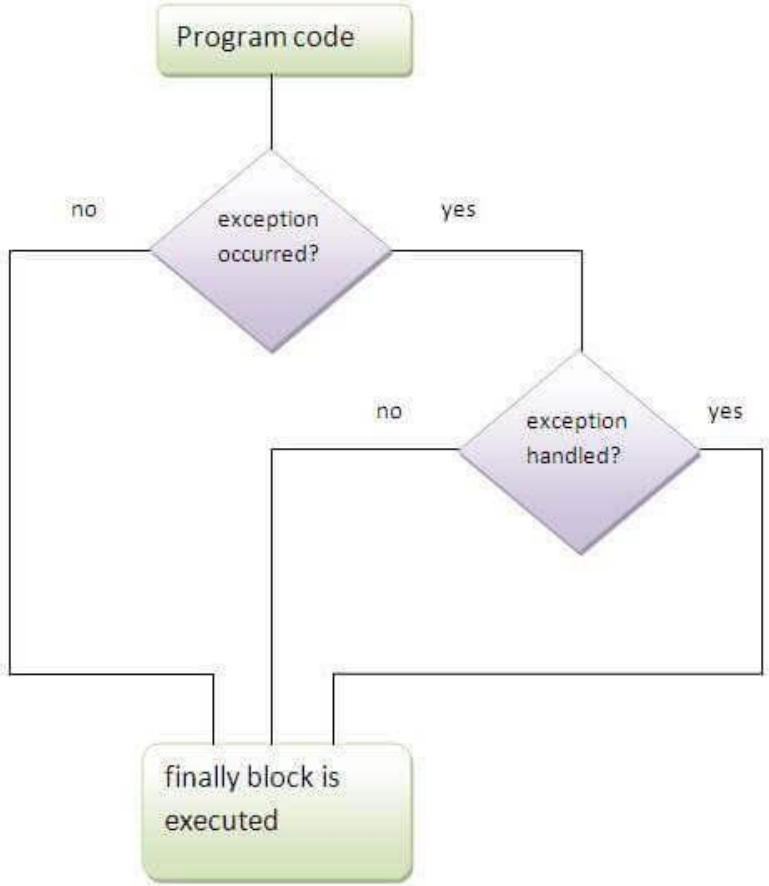
Why use nested try block

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.
- In such cases, exception handlers have to be nested.

```
....
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

Java finally block

- It is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.



Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

- **Rule:** For each try block there can be zero or more catch blocks, but only one finally block.
- **Note:** The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw keyword

- The Java **throw** keyword is used to explicitly throw an exception.
- We can throw either **checked or uncheked exception** in java by throw keyword.
- The **throw** keyword is mainly used to **throw** custom exception.

The syntax of java throw keyword

- throw exception;
- throw new IOException("sorry device error);

java throw keyword example

```
public class TestThrow1{
   static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
   }
   public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

QUIZ

```
class exception_handling
{
public static void main(String args[])
{
try
{
int a, b;
b = 0;
a = 5 / b;
System.out.print("A");
}
catch(ArithmeticException e)
{
System.out.print("B");
}
finally
{
System.out.print("C");
}
}
}
```

a) A

b) B

c) AC

d) BC

ANSWER: D

QUIZ

1. Which of these keywords is not a part of exception handling?

A. try

B. Finally

C. Thrown

D. Catch

ANSWER: C