Java Programming

UNIT - III

by

Mr. K. Srikar Goud
Asst. Professor
Department of Information Technology

Exception Handling

CONTENTS

- Overview of Exceptions, errors
- Keywords
- try and catch block
- Multiple catch block
- Nested try
- finally block
- throw keyword
- Exception Propagation
- throws keyword

Overview of Exceptions, errors

What is Error?

- **error** is a bug in a **program** that causes it to operate incorrectly.

What is Exception in Java

- **Dictionary Meaning:** Exception is an abnormal condition.

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

What is Exception Handling?

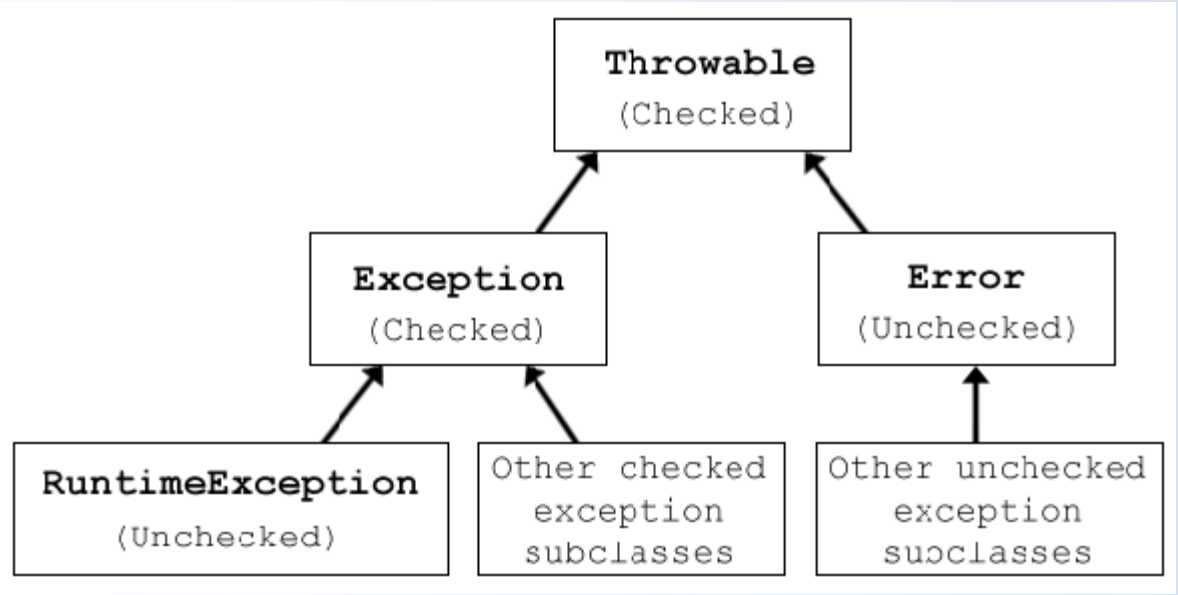
- Exception Handling is a mechanism to handle runtime errors such as
  - ClassNotFoundException,
  - IOException
  - SQLException
  - RemoteException etc.

Java Exception Keywords

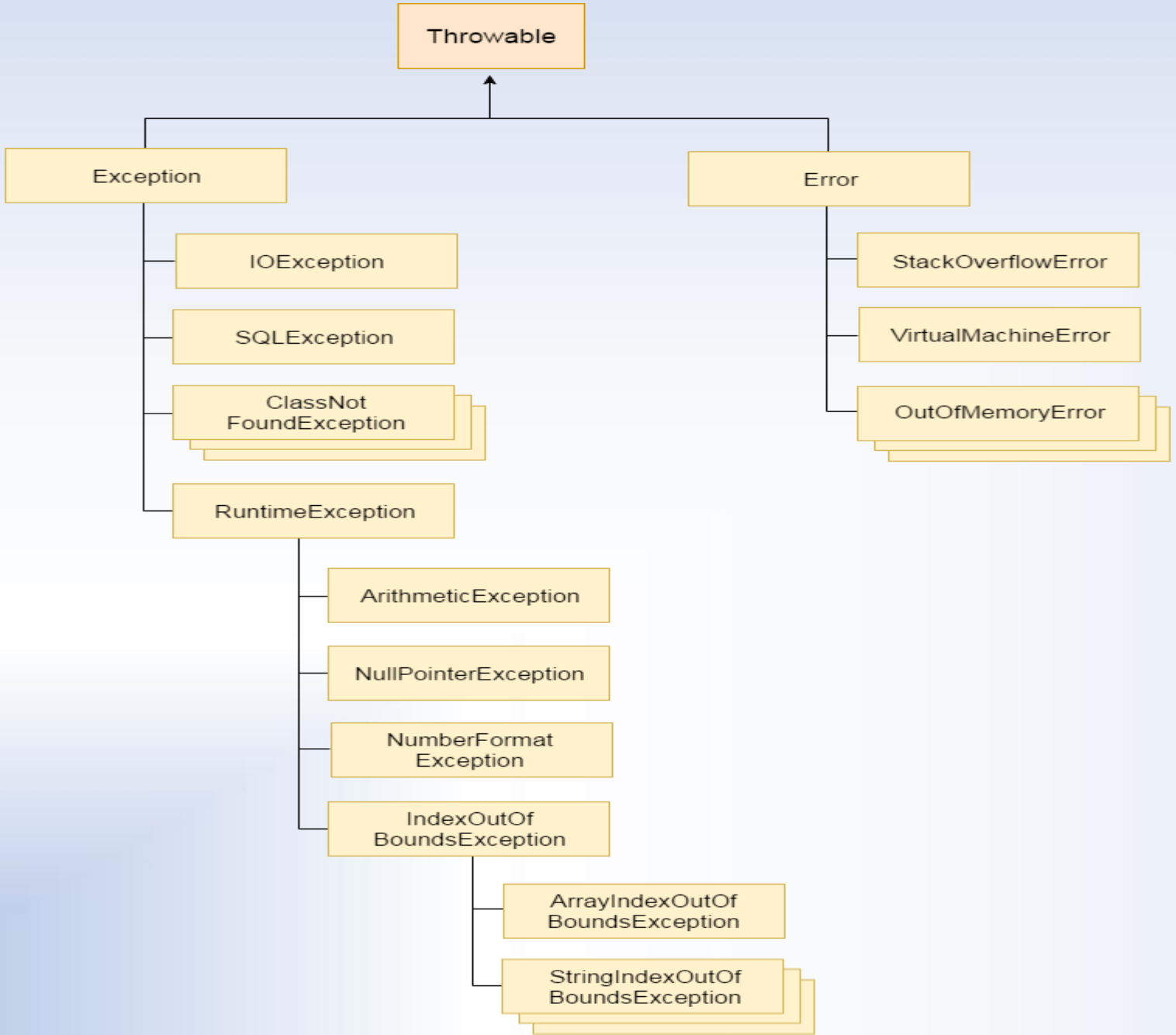
Keyword	Description
Try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which 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.

Types of Java 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.
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.



- Hierarchy of Java Exception classes



try-catch block to handle the exception.

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

Java **catch** block is used to handle the Exception by declaring the type of exception within the parameter.

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

Output:

```
Exception in thread main
java.lang.ArithmeticException:/ by zero
rest of the code...
```

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.
- 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.

Example of java multi-catch block.

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

Output:

Arithmetic Exception occurs
rest of the code

Nested try block

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

Syntax:

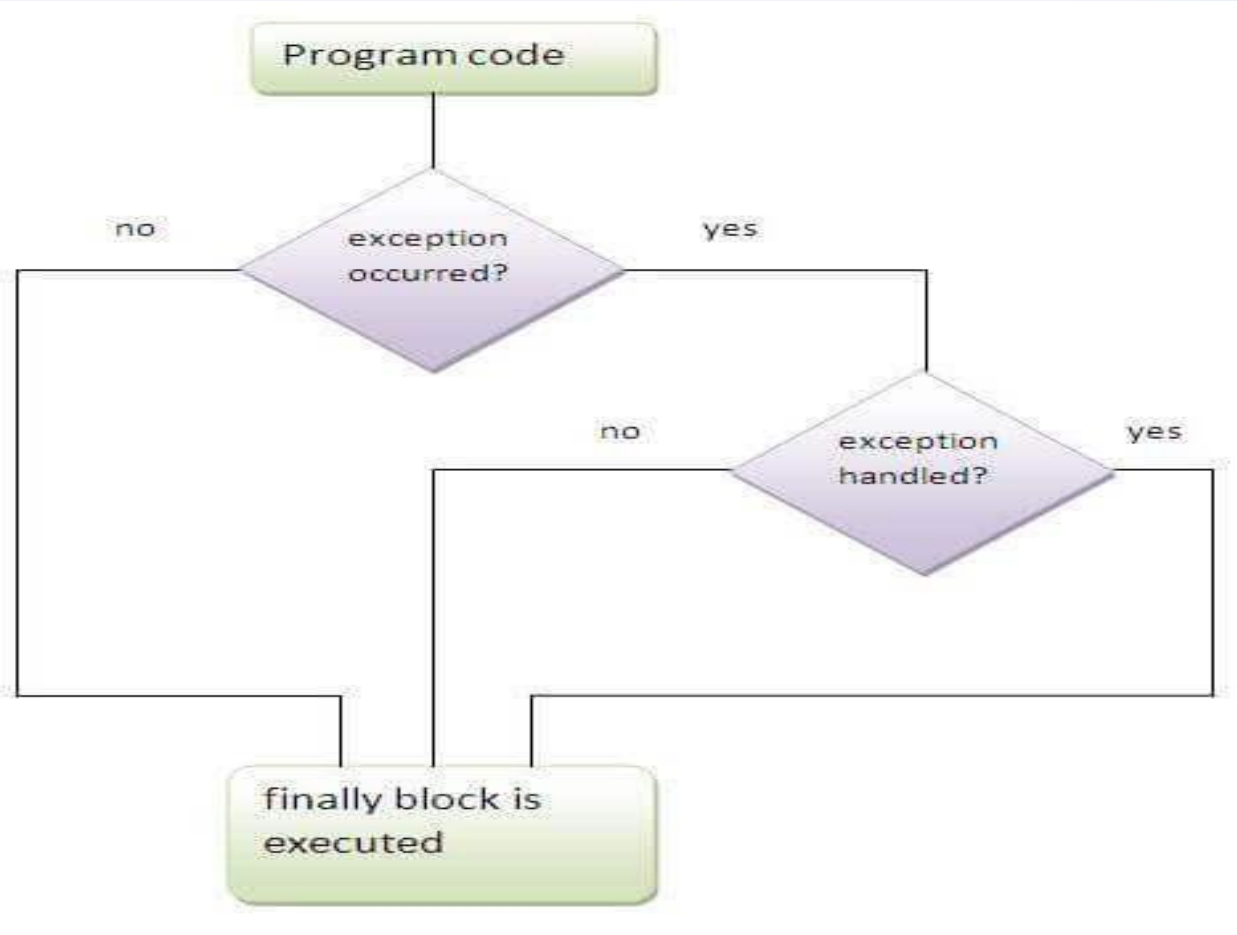
```
....
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

Nested try example

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

Java finally block

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



Case 1: where **exception doesn't occur**.

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:5

finally block is always executed

rest of the code...

Case 2: exception occurs and not handled.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output: finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3: exception occurs and handled.

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

Output: Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...

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 is, syntax - **throw** exception;
- throw IOException syntax - **throw new** IOException("sorry device error);

throw keyword example

Example: validate method that takes integer value as a parameter. If the age is less than 18, throw an ArithmeticException otherwise print a message welcome to vote.

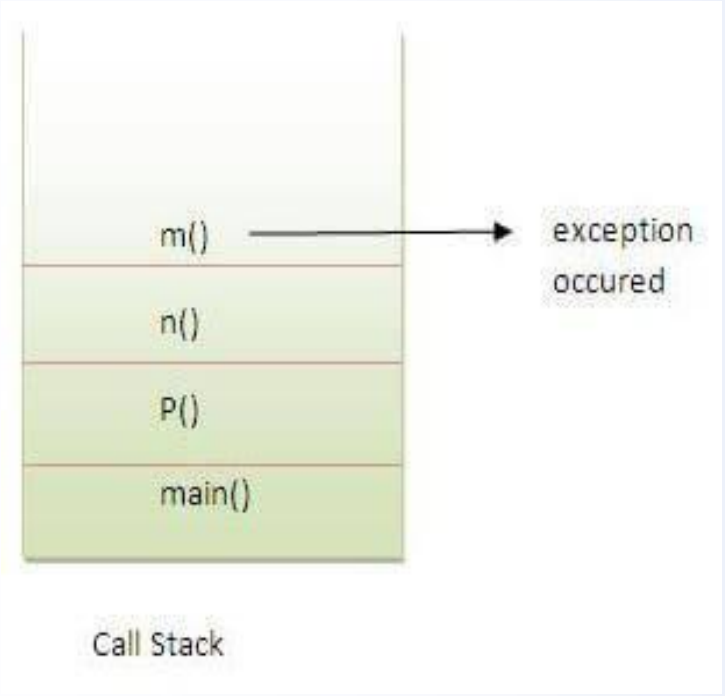
```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
 }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

- An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method
- If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.
- This is called exception propagation.



Program of Exception Propagation

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:exception handled

normal flow...

throws keyword

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

return_type method_name() **throws** exception_class_name{

//method code

}

Java throws example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

- Difference between throw and throws in Java

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Difference between final,finally and finalize in Java

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Are you able to answer these questions?

- What is the difference between checked and unchecked exceptions?
- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when finally block is not executed?
- What is exception propagation?
- What is the difference between throw and throws keyword?

Multi Threading

Differences between multi threading and multitasking

Multi-Tasking

- Two kinds of multi-tasking:
  1) process-based multi-tasking
  2) thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.
- Processes are heavyweight tasks:
  1) that require their own address space
  2) inter-process communication is expensive and limited
  3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
- several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:

  1) they share the same address space

  2) they cooperatively share the same process

  3) inter-thread communication is inexpensive

  4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:

  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

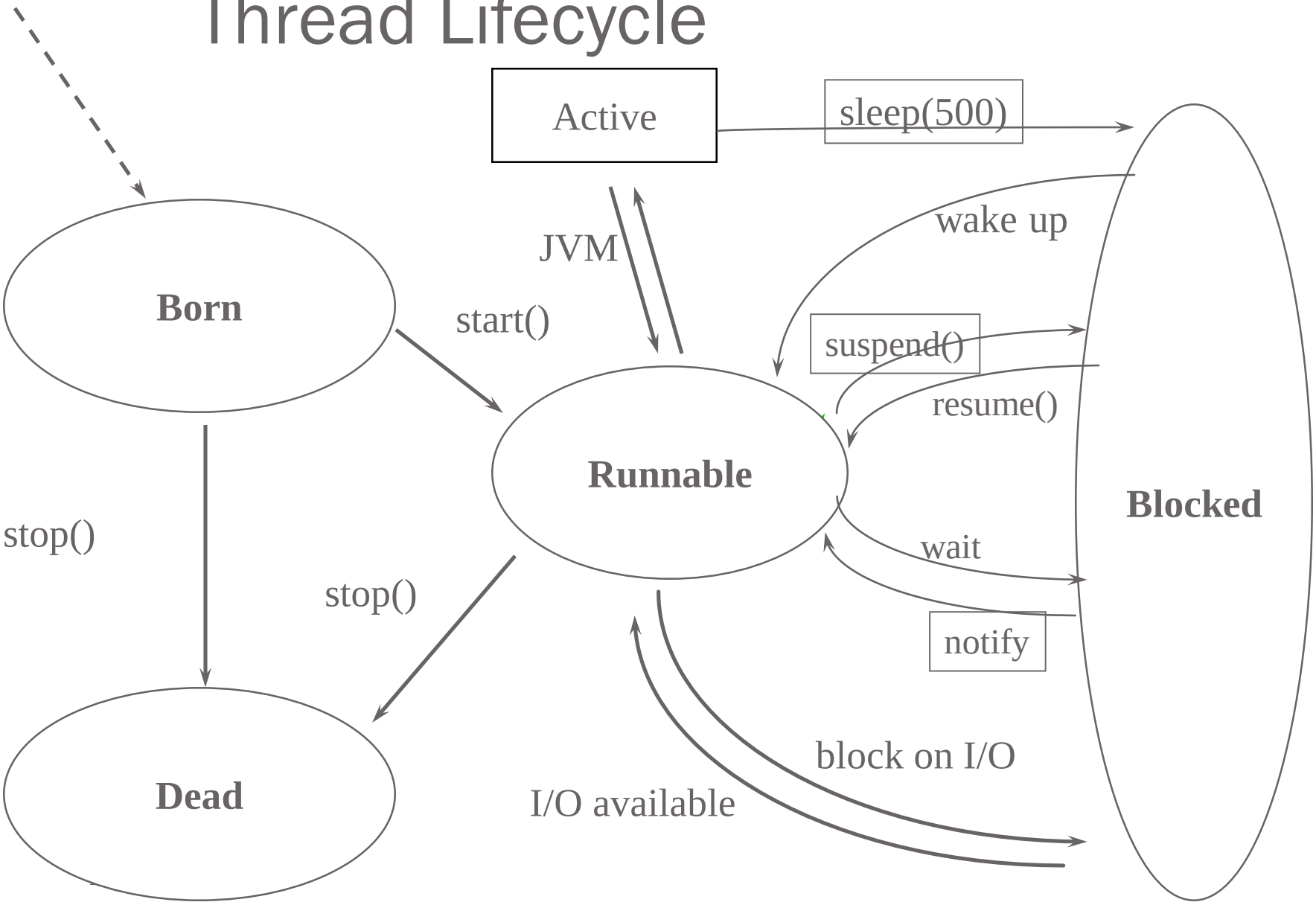
  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU

  3) of course, user input is much slower than the computer

Thread Lifecycle

- Thread exist in several states:
  1) ready to run
  2) running
  3) a running thread can be suspended
  4) a suspended thread can be resumed
  5) a thread can be blocked when waiting for a resource
  6) a thread can be terminated
- Once terminated, a thread cannot be resumed.

Thread Lifecycle



- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Creating Threads

- To create a new thread a program will:

  1) extend the Thread class, or

  2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

- Start: a thread by calling start its run method
- Sleep: suspend a thread for a period of time
- Run: entry-point for a thread
- Join: wait for a thread to terminate
- isAlive: determine if a thread is still running
- getPriority: obtain a thread's priority
- getName: obtain a thread's name

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:

  1) create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

  public void run()

  2) instantiate a Thread object within that class, a possible constructor is:

  Thread(Runnable threadOb, String threadName)

  3) call the start method on this object (start calls run):

  void start()

Example: New Thread 1

- A class NewThread that implements Runnable:

```
class NewThread implements Runnable {
Thread t;
//Creating and starting a new thread. Passing this to the
// Thread constructor – the new thread will call this
// object's run method:
NewThread() {
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
```

Example: New Thread 2

```
//This is the entry point for the newly created thread – a five-iterations loop
//with a half-second pause between the iterations all within try/catch:
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

Example: New Thread 3

```
class ThreadDemo {
public static void main(String args[]) {
//A new thread is created as an object of
// NewThread:
new NewThread();
//After calling the NewThread start method,
// control returns here.
```

Example: New Thread 4

```
//Both threads (new and main) continue concurrently.
//Here is the loop for the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

New Thread: Extend Thread

- The second way to create a new thread:

  1) create a new class that extends Thread

  2) create an instance of that class

- Thread provides both run and start methods:

  1) the extending class must override run

  2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:

```
class NewThread extends Thread {
//Create a new thread by calling the Thread's
// constructor and start method:
NewThread() {
super("Demo Thread");
System.out.println("Child thread: " + this);
start();
}
```

Example: New Thread 2

NewThread overrides the Thread's run method:

```
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

Example: New Thread 3

```
class ExtendThread {
public static void main(String args[]) {
//After a new thread is created:
new NewThread();
//the new and main threads continue
//concurrently…
```

Example: New Thread 4

```
//This is the loop of the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.
- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:

  1) classes can define so-called synchronized methods

  2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called

  3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Synchronization

- Language keyword: `synchronized`
- Takes out a monitor lock on an object
  - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a "force field" around the locked object so no other threads can enter
  - Actually, it only blocks access to other synchronizing threads

Daemon Threads

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- For example, you can start or suspend all the threads within a group with a single method call.
- Java thread groups are implemented by the "ThreadGroup" class in the java.lang package.
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created

The ThreadGroup Class

- The "ThreadGroup" class manages groups of threads for Java applications.
- A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- ThreadGroups can contain not only threads but also other ThreadGroups.
- The top-most thread group in a Java application is the thread group named main.
- You can create threads and thread groups in the main group.
- You can also create threads and thread groups in subgroups of main.

Creating a Thread Explicitly in a Group

- A thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread.
- The Thread class has three constructors that let you set a new thread's group:

  public Thread(ThreadGroup *group*, Runnable *target*) public Thread(ThreadGroup *group*, String *name*) public Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group.

  For example:

  ```
  ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
  Thread myThread = new Thread(myThreadGroup, "a thread for my group");
  ```