Programming in Java

Dr. Nyein Aye Maung Maung
Dr. Eng (Ritsumeikan University, Japan)
Lecturer
Computer Engineering and Information Technology Dept.
Yangon Technological University

Course Schedule

Week	Topics
Week 1	Overview of JAVA, Data Types, Variables and Arrays
Week 2	Operators and Control Statements
Week 3	Classes and A closer look at Methods and Classes
Week 4	Inheritance, Polymorphism, Abstraction and Encapsulation
Week 5	Packages and Interfaces
Week 6	Exception Handling and Multi-threaded Programming
Week 7	String Handling
Week 8	Exploring java.lang and More utilities classes
Week 9	Java Collections Framework
Week 10	Java I/O
Week 11	Basic Graphical User Interface
Week 12	Event Handling
Week 13	Database Programming
Week 14	Applet and Networking

Lecture 6

Exception Handling and Multithreading

Outline of Class (Lecture 5)

- Exception Handling
  - try-catch and finally
  - throwing exceptions
  - Exception as objects
  - Creating new exception classes
- Multithreading

Lecture Objectives

- To get an overview of exceptions and exception handling
- To explore the advantages of using exception handling
- To develop applications with exception handling
- To develop task classes by implementing the Runnable interface
- To create threads to run tasks using the Thread class

Topic 1

Exception Handling

Exceptions

- An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution.
- Exception handling enables a program to deal with exceptional situations and continue its normal execution.
  - **Error** – An error is a problem which is serious and any program shouldn't face it.
  - **Exception** – An exception is a problem that is not that serious and a program may or may not want it.
    - Example exceptions
      - A requested file cannot be found
      - An array index is out of bounds
      - A network link failed
- Two benefits of handling exceptions
  - allow to fix the error
  - prevent the program from automatically terminating

Why Use Exceptions?

- A simple quotient program

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
    (number1 / number2));
  }
}
```

Second number=0 → a runtime error would occur, because you can not divide an integer by **0**.

Why Use Exceptions?

- Quotient program with method to handle divide by zero error.

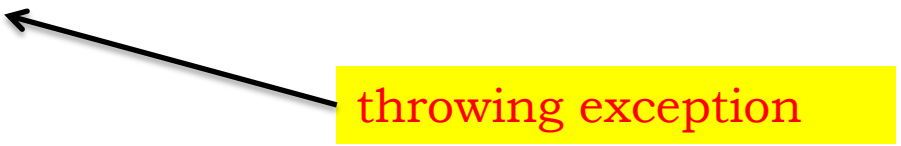
```
import java.util.Scanner;
public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }
    return number1 / number2;
  }
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

If **number2** is **0**, it cannot return a value, so the program is terminated

Why Use Exceptions?

- Quotient program with exception

```
import java.util.Scanner;
public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");
    return number1 / number2;
  }
   public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
```

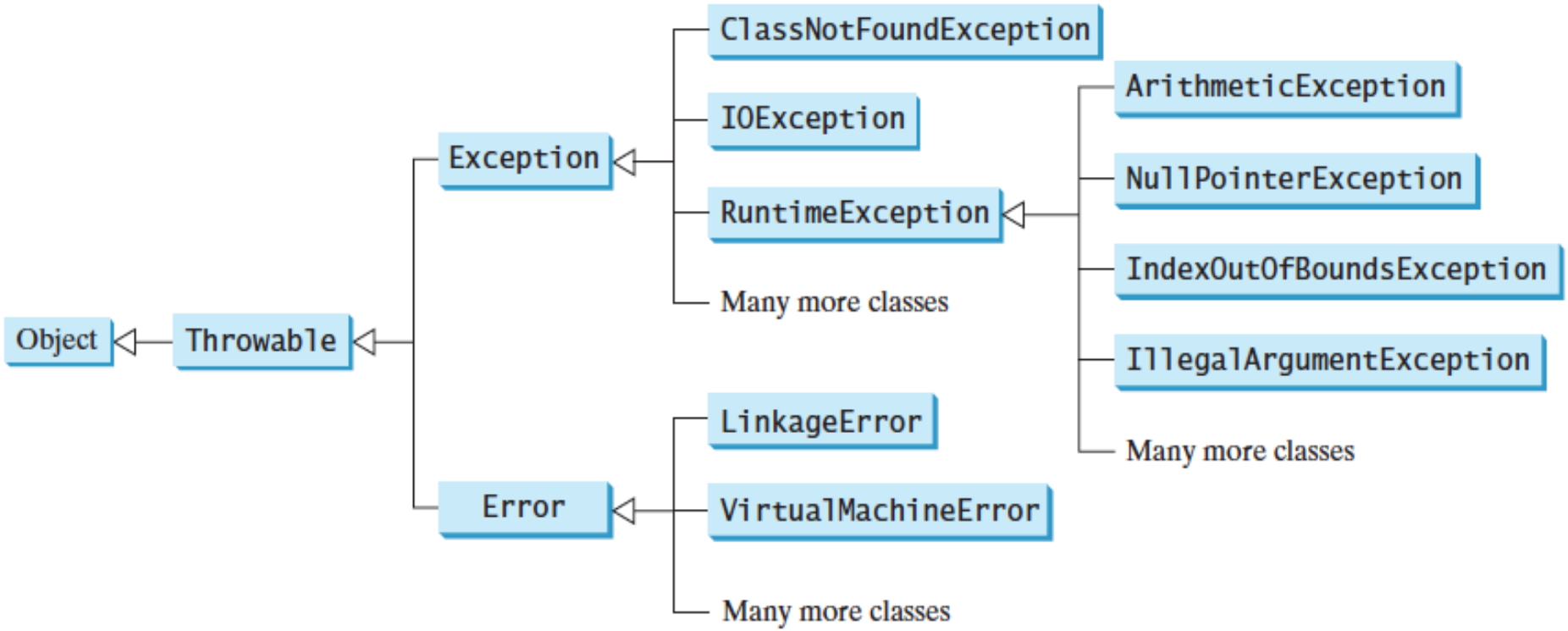


```
try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "+ result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

handle exception

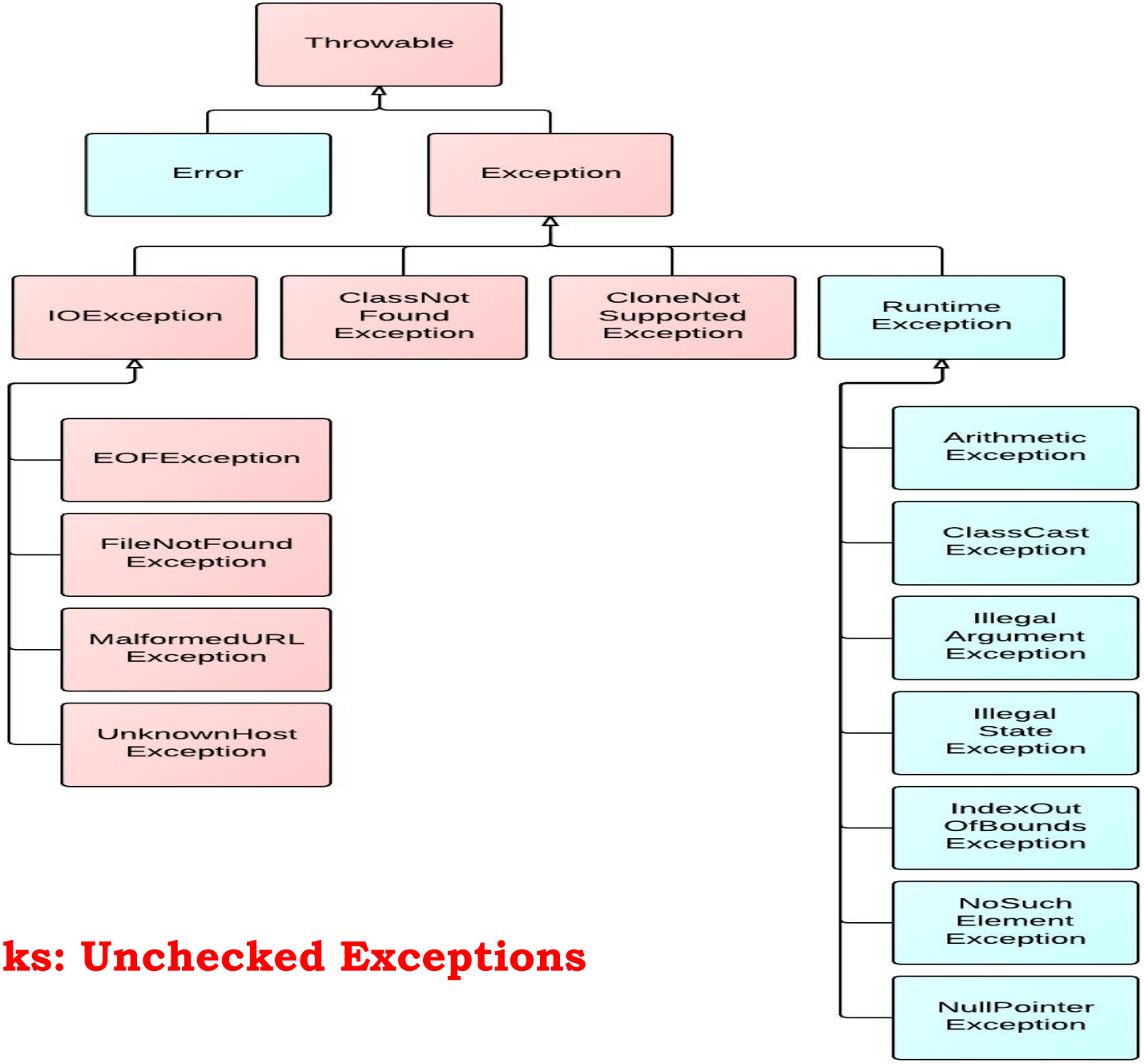
Program continues even after exception was thrown.

Exception Classes



Exception Types

1. Unchecked Exceptions
   - RuntimeException, Error, and their subclasses are known as unchecked exceptions.
2. Checked Exceptions
   - 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 using try-catch block or it must specify the exception using *throws* keyword



Green blocks: Unchecked Exceptions

Unchecked Runtime Exceptions

Type	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Common Scenarios of Java Exceptions

```
int a=50/0;//ArithmeticException
```

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

```
String s=null;
System.out.println(s.length());//NullPointerException
```

```
String s="abc";
System.out.println(s.charAt(3));//StringIndexOutOfBounds
Exception
```

Checked Runtime Exceptions

Type	Description
ClassNotFoundException	Class not found
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface
IllegalAccessException	Access to a class is denied
SQLException	Can occur both in the driver and the database
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file
DataAccessException	Something went wrong while executing a SQL statement from jOOQ

Exception Handling Keywords

Keyword	Description
try	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.

Handling Exception using try-catch

- Enclose the code to monitor inside a try block
- immediately following the try block, include a catch clause that specifies the exception type that you wish to catch

```
try{
        //code that may throw exception
}catch(Exception_class_Name ref)
{
        //Code to perform when an exception occurs
}
```

Example 6.1. Demonstrate try-catch

```
import java.util.Random;
class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();
        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

Multiple catch Clauses

```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

Example 6.2. Demonstrate multiple catch statements

```
// Demonstrate multiple catch statements.
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

a=0

a>0

java MultiCatch → a=0

java MultiCatch test → a=1

Nested try Statements

- The try block within a try block is known as nested try block in java.
- 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)
{
}
```

Example 6.2. Demonstrate nested try statements

```
class NestTry {
public static void main(String args[]) {
    try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        try { // nested try block
        if(a==1) a = a/(a-a);
        if(a==2) {
        int c[] = { 1 };
        c[42] = 99;
        }
        } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
        }
        } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
        }
        }
        }
```

no command-line argu → a divide-by-zero exception.

one command-line arg → a divide-by-zero exception

two command-line args→ an out-of-bounds exception

finally Statement

- **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.

```
try
{
    statements
    resource-acquisition statements
} // end try
catch ( AKindOfException exception1 )
{
    exception-handling statements
} // end catch
catch ( AnotherKindOfException exception2 )
{
    exception-handling statements
} // end catch
finally
{
    statements
    resource-release statements
} // end finally
```

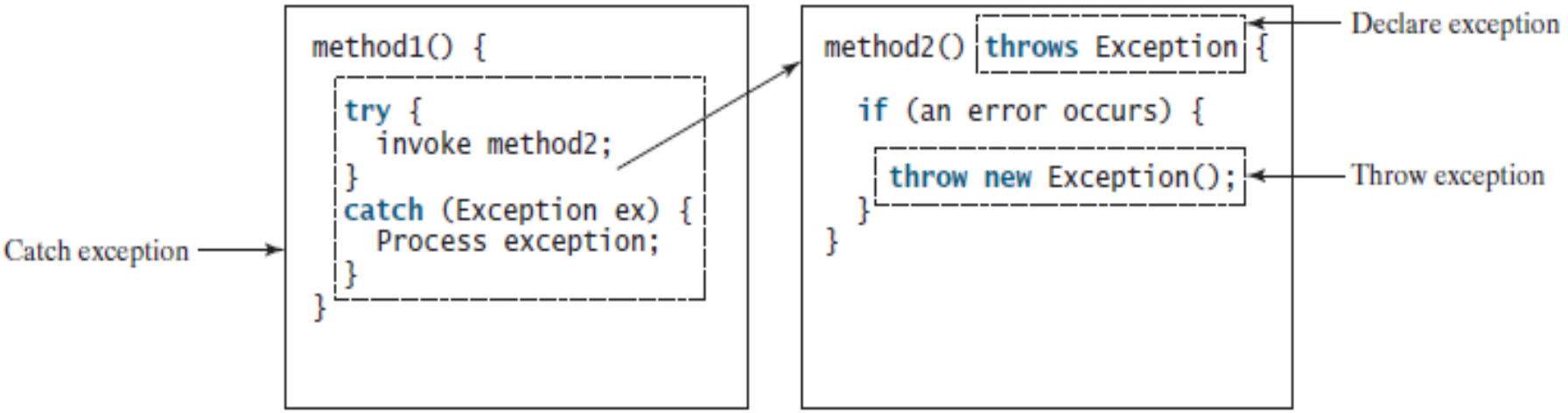
Unreachable Exceptions

- When using multiple catch statements, it is important to remember that **exception subclasses must come before any of their superclasses**

```
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
        System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
        ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException e) { // ERROR - unreachable
        System.out.println("This is never reached.");
        }
    }
}
```

Exception Handling Model

- Java's exception-handling model is based on three operations
  - **declaring** an exception
  - **throwing** an exception , and
  - **catching** an exception



Declaring Exceptions

- Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions .
- Which exception should be declared
  - checked exception only, because:
  - unchecked Exception: under your control so correct your code.
  - error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.
- Use the throws keyword in the method header

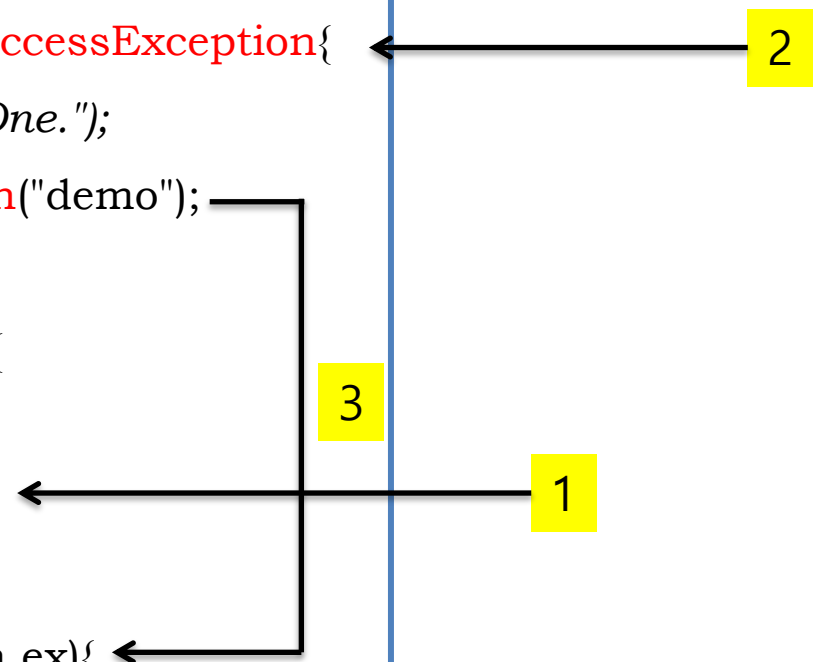
```
public void myMethod() throws IOException
```

or

```
public void myMethod() throws Exception1, Exception2, ..., ExceptionN
```

Example 6.3. Demonstrate declaring exceptions using 'throws' keyword

```
class ThrowsDemo {
        static void throwOne() throws IllegalAccessException{
            System.out.println("Inside throwOne.");
            throw new IllegalAccessException("demo");
            }
        public static void main(String args[]) {
                try{
                        throwOne();
                }
                catch(IllegalAccessException ex){
                        System.out.println("Caught"+ex);

                }
}}
```



Output

Inside throwOne.
Caughtjava.lang.IllegalAccessException: demo

Throwing Exceptions

- It is possible to throw an exception explicitly, using the **throw** statement

  throw ThrowableInstance;

- 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
- Flow of execution stops immediately after the throw statement
  - check nearest enclosing try block to see if it has a catch statement that matches the type of exception
    - found → transfer control to that statement
    - else → keep inspecting next enclosing try statement
      - No matching → default exception handler halts the program

```
IllegalArgumentException ex =new
IllegalArgumentException("Wrong Argument");
throw ex;
```

- Example (throw statement)

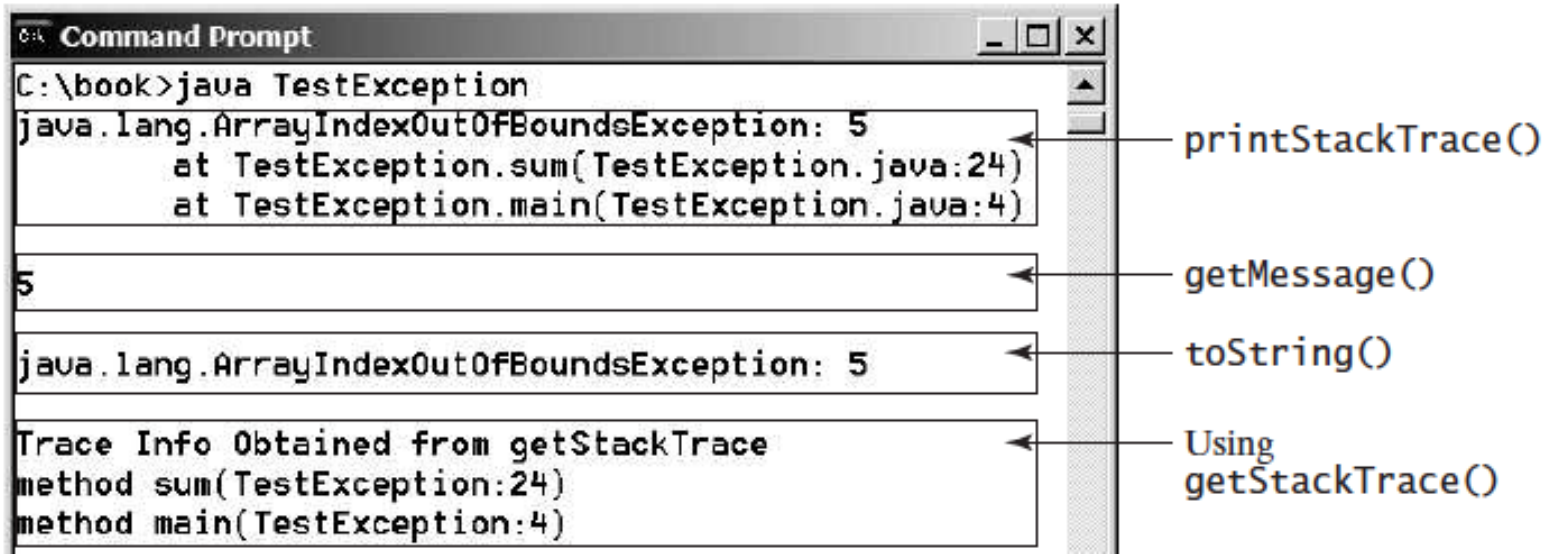
```
/** Set a new radius */
  public void setRadius(double newRadius) throws IllegalArgumentException
{
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
  }
```

Throw and Throws

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.

Getting Information from Exceptions

`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `":"` (a colon and a space); (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.



Defining Custom Exception Classes

- If you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class
- Can define a custom exception class by extending the java.lang.Exception class

Example 6.4. Write a program that lest users of age above 18 to vote. Write a method validate(int age) that checks whether the use is valid to vote or not. If not valid, it will throw an exception. Design your own exception class named InvalidAgeException.

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
     try{
     validate(13);
     }catch(Exception m){System.out.println("Exception occured: "+m);}

     System.out.println("rest of the code...");
  }
}
```

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

Output:Exception occured: InvalidAgeException:not valid rest of the code…

Time for Quiz

1. What is the output of the following statements?

```
public class Test {
public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        try {
        System.out.println(1/0);
        }
        catch (Exception ex) {
                System.out.println(ex);
        }
}}}
```

2. What is the output of the following statements?

```
public class Test {
public static void main(String[] args) {
        try {
        for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
        }
        }
        catch (Exception ex) {
                System.out.println(ex);
        }
}
}
```

3. What exception will happen when executing the following codes?

```
public class Test {
        public static void main(String[] args) {
            try {
            int[] list = new int[10];
            System.out.println("list[10] is " + list[10]);
            }
            catch (ArithmeticException ex) {
            System.out.println("ArithmeticException");
            }
            catch (RuntimeException ex) {
            System.out.println("RuntimeException");
            }
            catch (Exception ex) {
            System.out.println("Exception");
            }
        }
}
```

4. What exception will happen when executing the following codes?

```
public class Test {
    public static void main(String[] args) {
        try {
        method();
        System.out.println("After the method call");
        }
        catch (ArithmeticException ex) {
        System.out.println("ArithmeticException");
        }
        catch (RuntimeException ex) {
        System.out.println("RuntimeException");
        }
        catch (Exception e) {
        System.out.println("Exception");
        }
    }
    static void method() throws Exception {
            System.out.println(1 / 0);
            }
}
```

5. What is the output of the following codes?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      String s ="abc";
      System.out.println(s.charAt(3));
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
    }
  }
}
```

6. What is the output of the following codes?

```
public class MyClass {
   public static void main(String[] args) {
      int k=0;
      try {
                  int i = 5/k;
          } catch (ArithmeticException e) {
                  System.out.println("1");
          } catch (RuntimeException e) {
                  System.out.println("2");
                  return;
          } catch (Exception e) {
                  System.out.println("3");
          } finally {
                  System.out.println("4");   }
      System.out.println("5");
   }
}
```

Part II

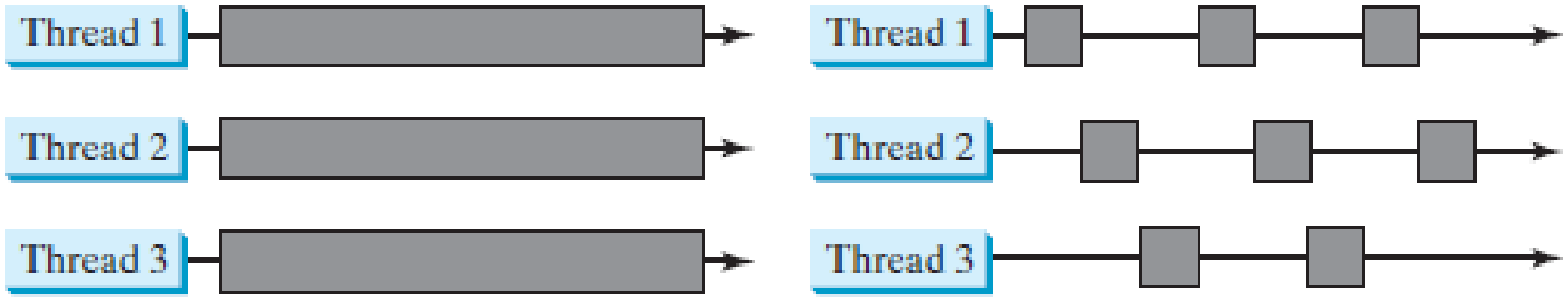
Multithreading

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:
- Process-based Multitasking (Multiprocessing)
  - Each process has an address in memory. In other words, each process allocates a separate memory area.
  - A process is heavyweight.
  - Cost of communication between the process is high.
  - Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.
- Thread-based Multitasking (Multithreading)
  - Threads share the same address space.
  - A thread is lightweight.
  - Cost of communication between the thread is low.

Multithreading

- **Multithreading in java is** a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multithreading enables multiple tasks in a program to be executed concurrently.
- A thread provides the mechanism for running a task.



Multiple threads on multiple CPUs

Multiple threads share a single CPU

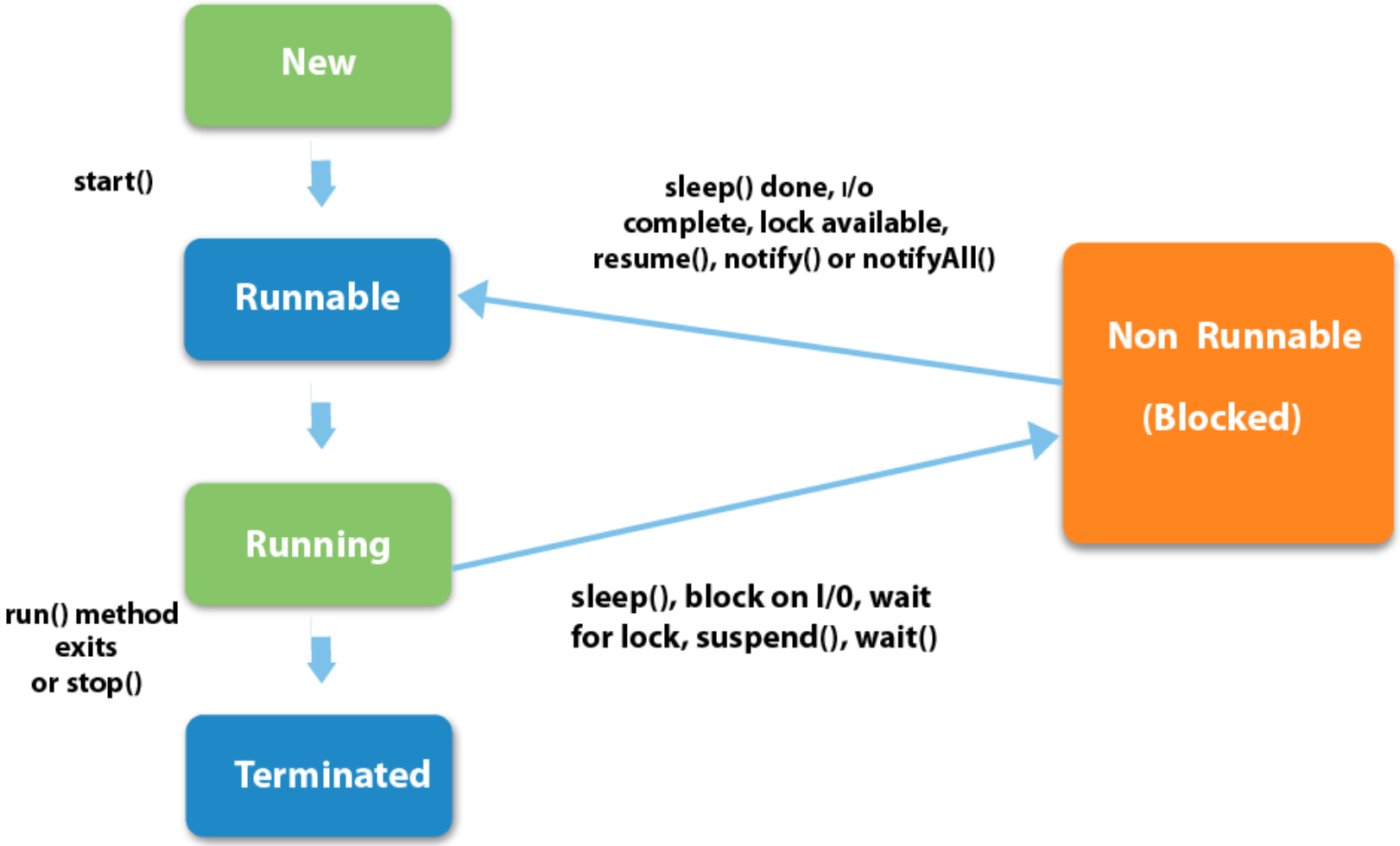
Advantages of Multithreading

- 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.

Life cycle of a Thread

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

Life cycle of a Thread



Creating Tasks and Threads

- Two ways:
  1. By implementing **Runnable interface**.
  2. By extending Thread class
- Create a thread by *instantiating an object of type Thread*
- A task class must implement the Runnable interface. *A task must be run from a thread.*

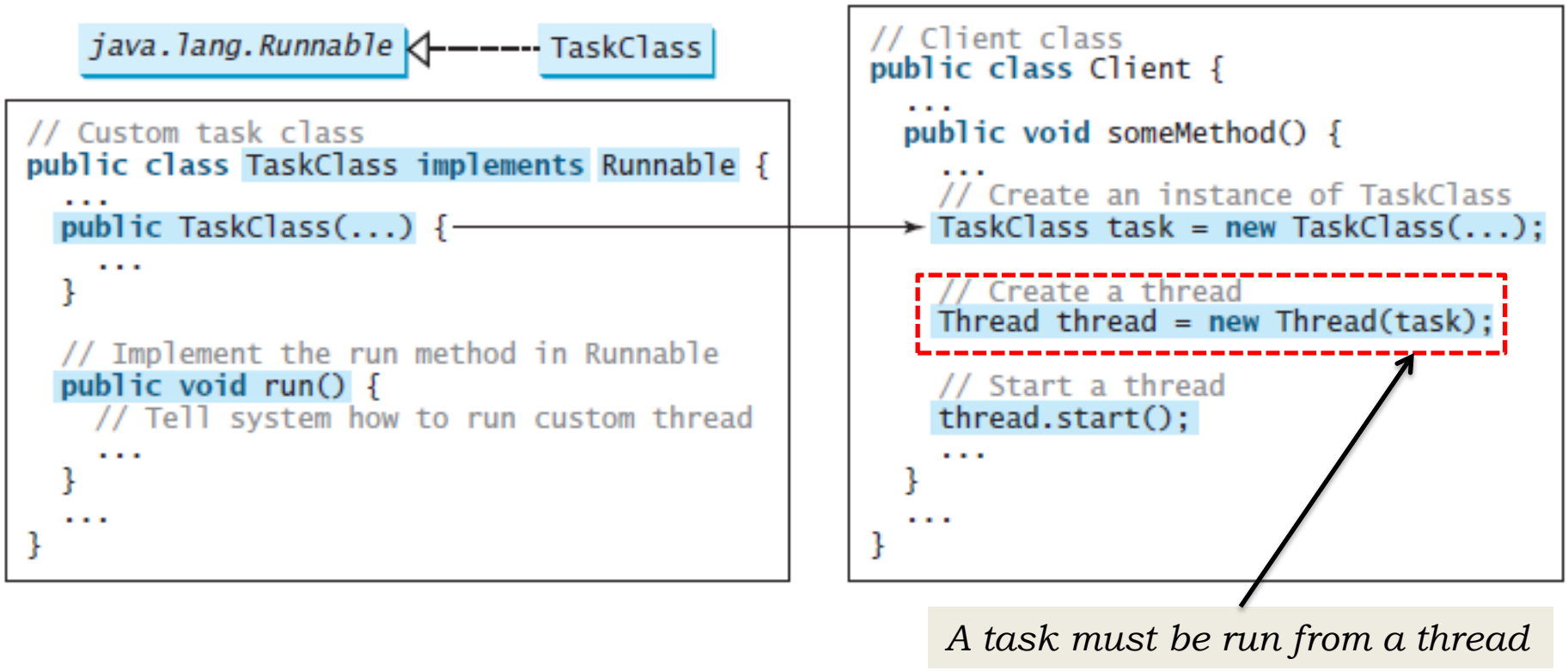
Creating Tasks and Threads

(1) Implementing Runnable interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread
- .Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Creating Tasks and Threads



Define a task class by implementing the Runnable interface

Creating Tasks and Threads

(2) Thread Class

- The **Thread** class contains the constructors for creating threads for tasks and the methods for controlling threads
- **Commonly used Constructors**
  - Thread()
  - Thread(String name)
  - Thread(task:Runnable r)
  - Thread(task:Runnable r,String name)
- **Commonly used methods**
  - **public void run():** is used to perform action for a thread.
  - **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
  - **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
  - **public void join():** waits for a thread to die.
  - **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.

- **Commonly used methods (continue)**
  - **public String getName():** returns the name of the thread.
  - **public void setName(String name):** changes the name of the thread.
  - **public Thread currentThread():** returns the reference of currently executing thread.
  - **public int getId():** returns the id of the thread.
  - **public boolean isAlive():** tests if the thread is alive.
  - **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
  - **public void stop():** is used to stop the thread(depricated).
  - **public void interrupt():** interrupts the thread.
  - **public void yield( ):** Causes a thread to pause temporarily and allow other threads to execute.

Creating Tasks and Threads


```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

Default: **Thread** class implements **Runnable**

Define a thread class by extending the Thread class

Example 6.5. Creating multiple threads by implementing Runnable Interface

```
class MyThread implements Runnable {
        String name;
        Thread t;
            MyThread (String thread){
            name = thread;
            t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
        }
         public void run() {
         try {
              for(int i = 5; i > 0; i--) {
              System.out.println(name + ": " + i);
               Thread.sleep(1000);
        }
        }catch (InterruptedException e) {
              System.out.println(name + "Interrupted");
        }
              System.out.println(name + " exiting.");
        }
}
```

```
class MultiThread {
public static void main(String args[]) {
      new MyThread("One");
      new MyThread("Two");
      new NewThread("Three");
try {
      Thread.sleep(10000);
} catch (InterruptedException e) {
       System.out.println("Main thread
Interrupted");
}
       System.out.println("Main thread
exiting.");
       }
}
```

Output

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Example 6.6. Creating multiple threads **using Thread Class**

```
class MultithreadingDemo extends Thread
{
        public void run()
        {
        try
        {
        System.out.println ("Thread " +
        Thread.currentThread().getId() +
        " is running");
        }
        catch (Exception e)
        {
        System.out.println ("Exception is caught");
        }
        }
}
```

```
public class Multithread
{
public static void main(String[] args)
{
int n = 8; // Number of threads
for (int i=0; i<8; i++)
{
MultithreadingDemo object =
new MultithreadingDemo();
object.start();
}
}
}
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run
  - **public int getPriority():** returns the priority of the thread.
  - **public int setPriority(int priority):** changes the priority of the thread.

Example 6.7. Demonstrate two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a nonpreemptive platform. One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
class clicker implements Runnable {
        long click = 0;
        Thread t;
        private volatile boolean running = true;
        public clicker(int p) {
                t = new Thread(this);
                t.setPriority(p);
        }
        public void run() {
                while (running) {
                click++;
                }
        }
        public void stop() {
                running = false;
        }
        public void start() {
                t.start();
        }}
```

```
class HiLoPri {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
        lo.start();
        hi.start();
        try {
        Thread.sleep(10000);
        } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
        }
        lo.stop();
        hi.stop();
        // Wait for child threads to terminate.
        try {
        hi.t.join();
        lo.t.join();
        } catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
        }
        System.out.println("Low-priority thread: " + lo.click);
        System.out.println("High-priority thread: " + hi.click);
    }
}
```

```
Low-priority thread: 4408112
High-priority thread: 589626904
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called ***synchronization***
- Key to synchronization is the concept of the **monitor** (also called a *semaphore*).
  - A *monitor* is an object that is used as a mutually exclusive lock, or ***mutex***.
  - Only one thread can ***own*** a monitor at a given time.
  - When a thread acquires a lock, it is said to have ***entered*** the monitor.
  - All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor.
  - These other threads are said to be ***waiting*** for the monitor.

Why Synchronization?

Example 6.8. Sample multithreaded program without synchronization.

Output is inconsistent.

```
class Table{
//method not synchronized
void printTable(int n){
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){
              System.out.println(e);
              }
   }
   }
}
```

```
class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
        this.t=t;
        }
        public void run(){
        t.printTable(5);
        }

}
```

```
class MyThread2 extends Thread{
      Table t;
      MyThread2(Table t){
      this.t=t;
      }
      public void run(){
      t.printTable(100);
      }
}

class TestSynchronization1{
      public static void main(String args[]){
      Table obj = new Table();//only one object
      MyThread1 t1=new MyThread1(obj);
      MyThread2 t2=new MyThread2(obj);
      t1.start();
      t2.start();
      }
}
```

Output:
5
100
10
200
15
300
20
400
25
500

Synchronized Method

- If you declare any method as synchronized, it is known as synchronized method.
  - Add **synchronized** keyword
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

- Modify printTable method in Example 6.8 to be synchronized method.

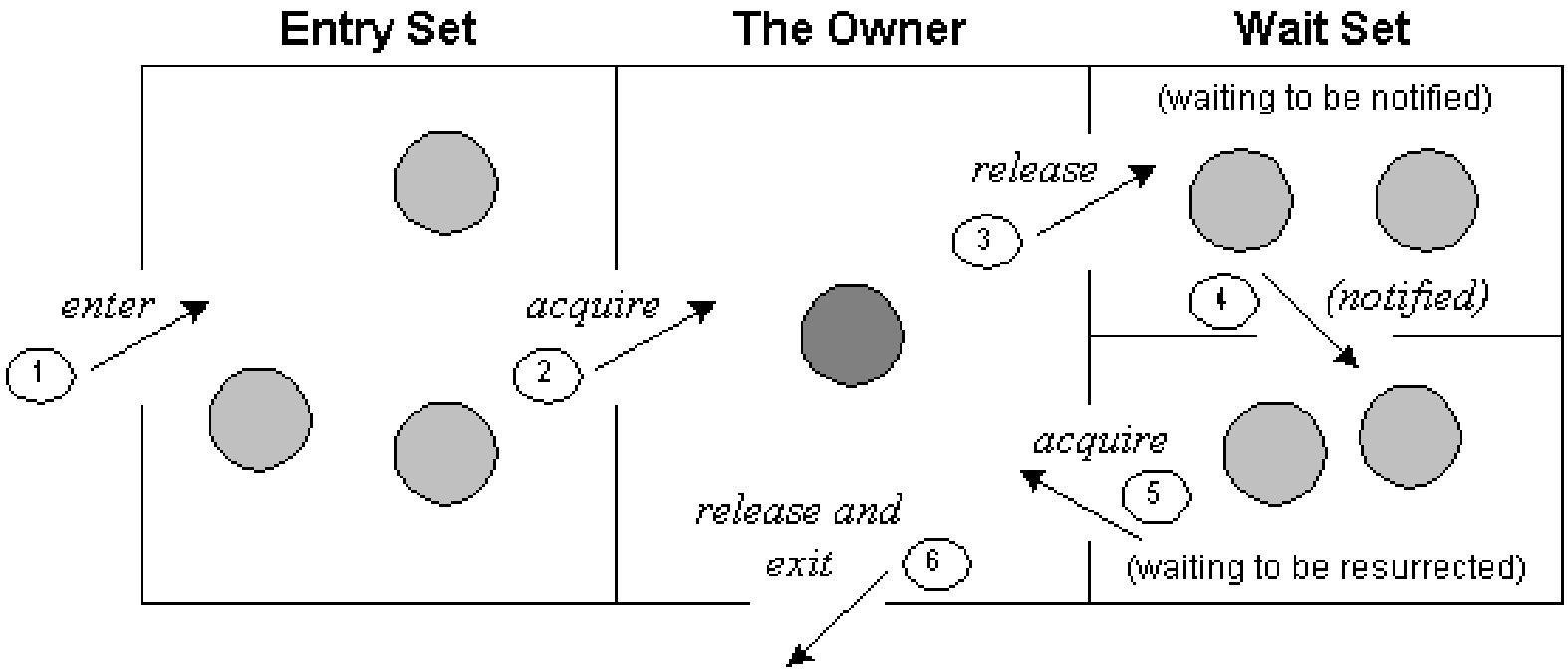
```
class Table{
//synchronized method
 synchronized void printTable(int n){
          for(int i=1;i<=5;i++){
              System.out.println(n*i);
              try{
               Thread.sleep(400);
              }catch(Exception e){System.out.println(e);}
            }

 }
}
```

Output
5
10
15
20
25
100
200
300
400
500

Output becomes consistent after synchronization.

Inter-thread communication

- Allowing synchronized threads to communicate with each other
- Is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed
- Three main methods
  - void wait() / void wait(long timeout)
    - Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed
  - notify()
    - Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened (arbitrary manner)
  - notifyAll()
    - Wakes up all threads that are waiting on this object's monitor



1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Example 6.9. Demonstrating inter-thread communication

```
class Customer{
      int amount=10000;
      synchronized void withdraw(int amount){
      System.out.println("going to withdraw...");
      if(this.amount<amount){
      System.out.println("Less balance; waiting for deposit...");
      try{wait();}catch(Exception e){}
      }
      this.amount-=amount;
      System.out.println("withdraw completed...");
      }
      synchronized void deposit(int amount){
      System.out.println("going to deposit...");
      this.amount+=amount;
      System.out.println("deposit completed... ");
      notify();
}
}
```

```
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output

going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed

Quiz on Multithreading

1. What is wrong in the following two programs? Correct the errors.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Test task = new Test();
    new Thread(task).start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(a)

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Thread t = new Thread(this);
    t.start();
    t.start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(b)

2. What is wrong in the following code?

```
synchronized (object1) {
try {
while (!condition) object2.wait();
}
catch (InterruptedException ex) {
}
}
```

Next Week Lecture

- String Handling

Thank you!