Java: Exception Handling

Objectives

- Write code to handle exceptions,
- Classify different types of exceptions
- Create user defined exceptions

Table of Content

Exceptions	Delegation - throws
Handling exceptions in the code	Partial Delegation, Chained Exceptions
Flow after exception	Re-throw
Multiple catches	Exception Wrapping
Catch all exception	Printing Stack Trace
Exception class	StackTraceElement
throw	Overriding and Exception
Throwing an unchecked exception	finally – Why another keyword needed?
Throwing a checked exception	

Exception

- Abnormal Event that disrupts the normal flow of program execution

Exception Recovery

- At times it is recoverable from an exception but may not be the case always

When can it occur?

- Abnormality do occur when the program is running

E.g.

  - User enters a text in place of an integer
  - A File is not present in the expected folder

- What if it is not handled?

  - Program may be terminated
  - May lock useful resources like file handles/ database / network connections
  - May cause data integrity issues as well

Recall : Encounters of runtime errors

- How many times have you bumped into runtime error so far?
- Can you name a few runtime errors that you have encountered?

- **`NullPointerException,`**

  `ArrayIndexOutOfBoundsException`

  →ClassesAndMethods-Part 1

- **`NoClassDefFoundError`** → Packages

- **`ClassCastException`** → Inheritance

- **`CloneNotSupportedException`** → Interface
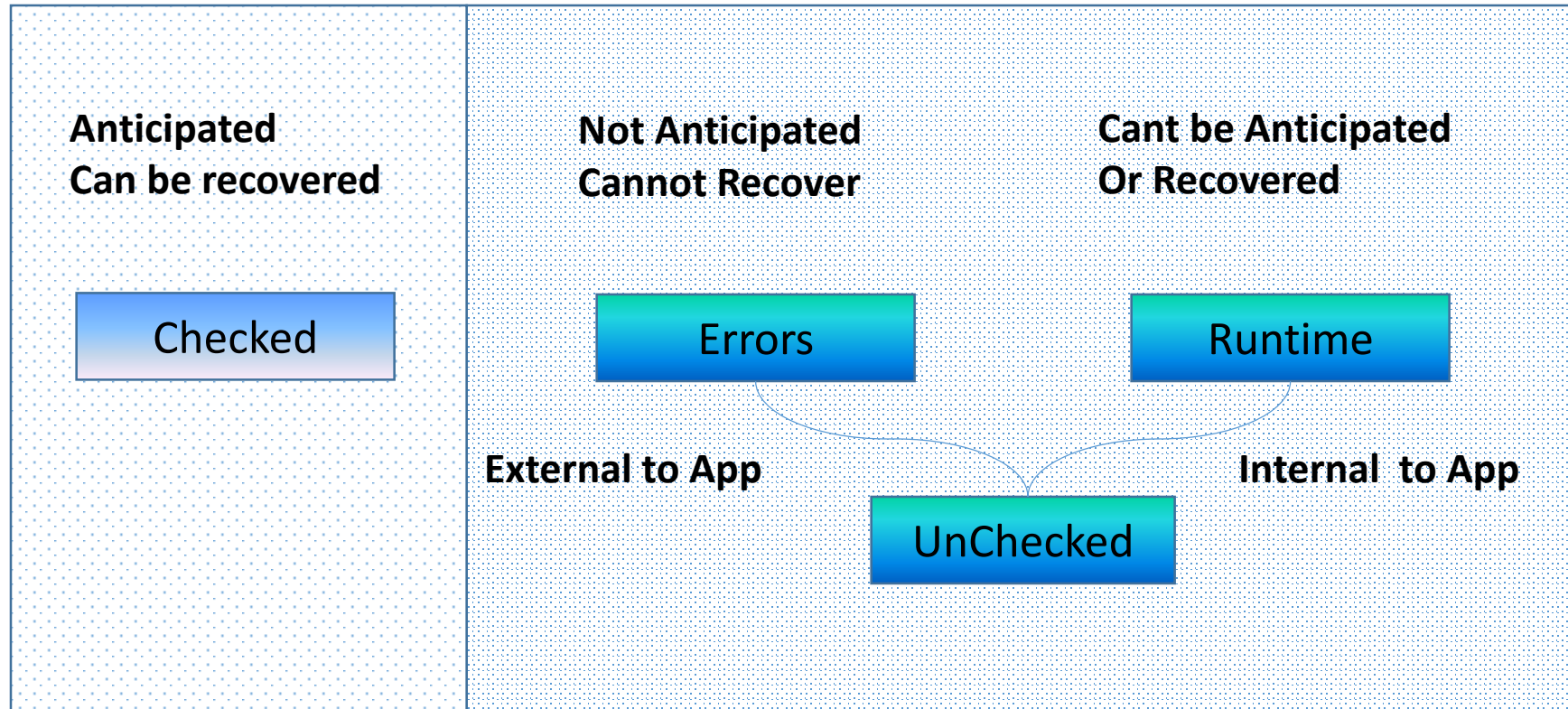
Common Exception scenarios

- Attempt to divide an integer by zero

- Attempt to call a method using a reference that is null.

- Attempting to open a nonexistent file for reading.
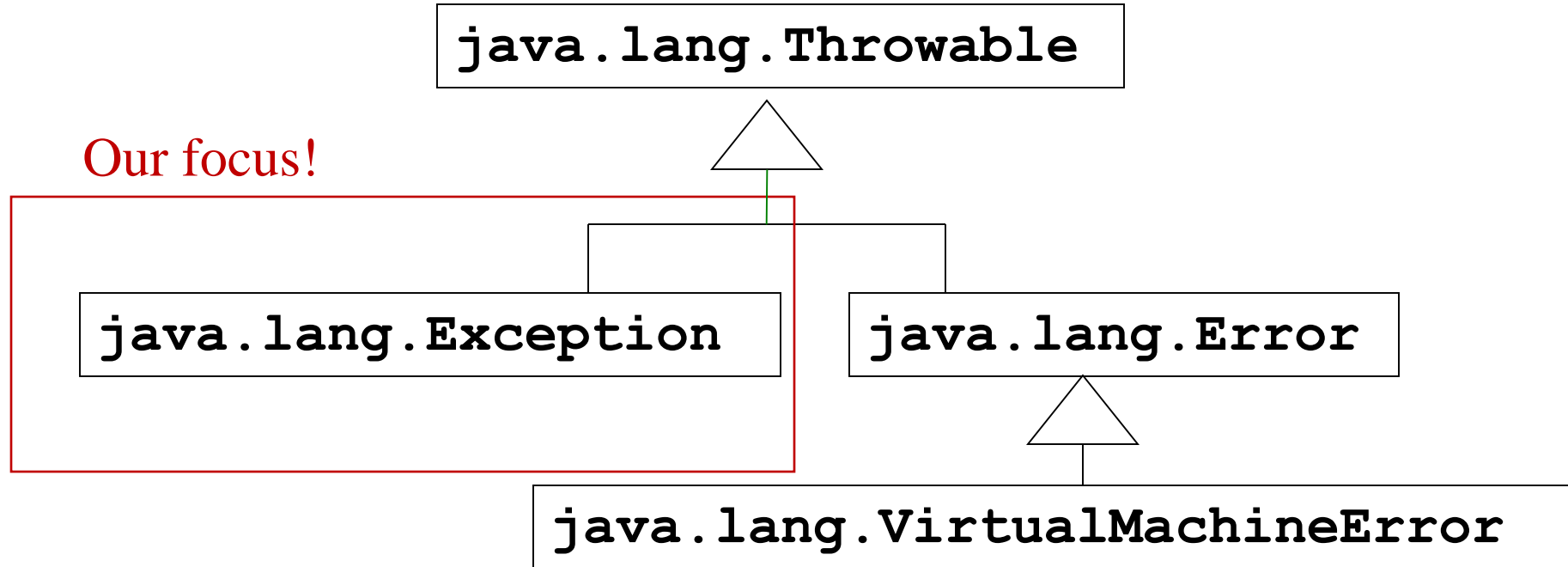
- JVM running out of memory.

Exception handling, required?

- To recover from the error conditions.
- To give users friendly, relevant messages when something goes wrong.
- To conduct certain critical tasks such as "save work" or "close open files/sockets" in case critical error leads to abnormal termination.
- To allow programs to terminate gracefully or operate in degraded mode.

Exception Categories

Anticipated Can be recovered

Not Anticipated Cannot Recover

Cant be Anticipated Or Recovered

Checked

Errors

Runtime

External to App

Internal to App

UnChecked

Exception Hierarchy

```
                    ┌──────────────────────────┐
                    │  java.lang.Throwable     │
                    └──────────────────────────┘
                                 △
   Our focus!                    │
 ┌─────────────────────────────────────┐
 │  ┌──────────────────────────┐   ┌──────────────────────────┐
 │  │  java.lang.Exception     │   │  java.lang.Error         │
 │  └──────────────────────────┘   └──────────────────────────┘
 └─────────────────────────────────────┘         △
                                                 │
                         ┌──────────────────────────────────────┐
                         │  java.lang.VirtualMachineError        │
                         └──────────────────────────────────────┘
```

- **Throwable** class is super class for all exceptions in Java. When an exception occurs in a method, an object of **Throwable** type is thrown.

- The 2 subclasses –**Error** and **Exception**

Exception Class Hierarchy
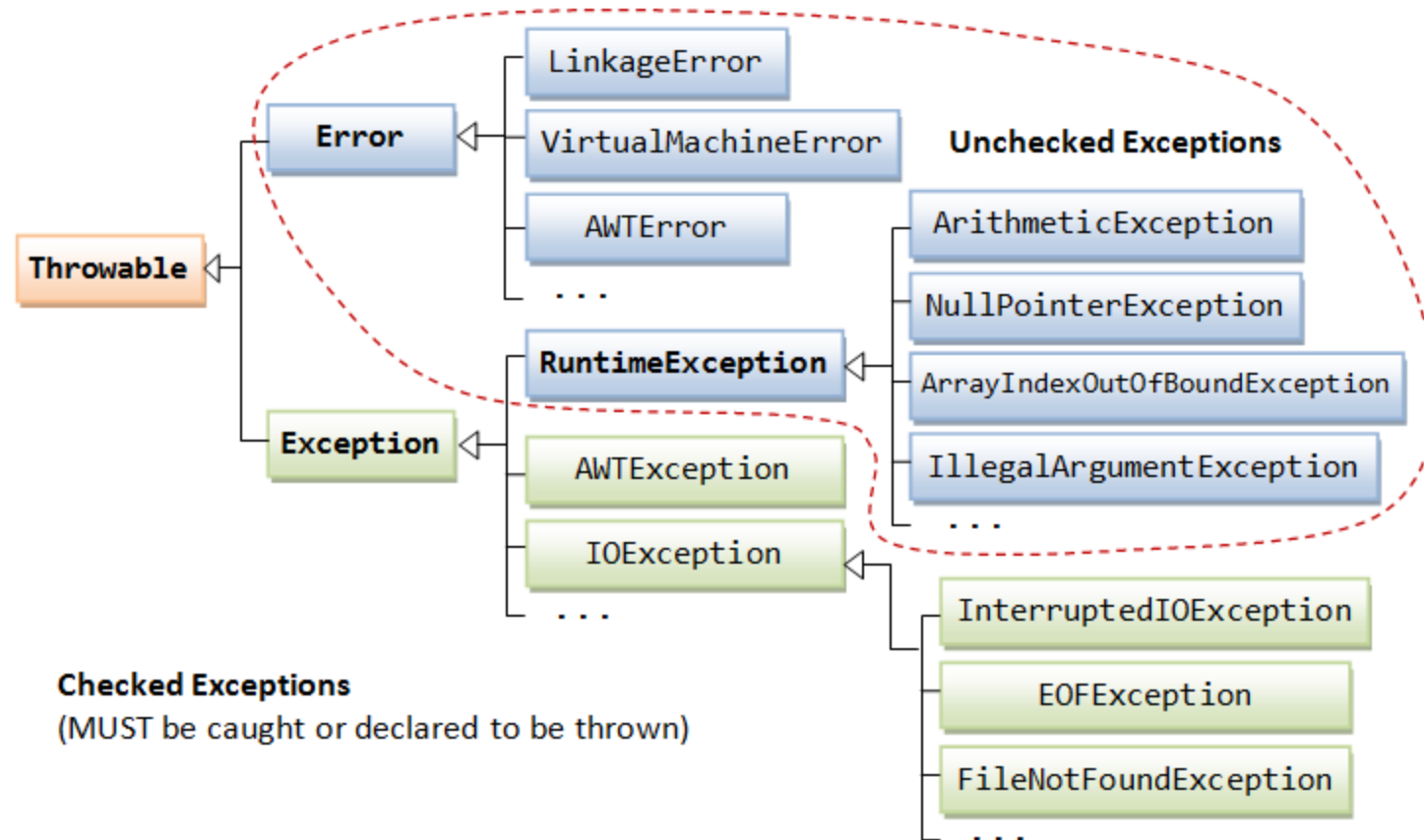
Types of exception

- Two types

Unchecked exceptions or runtime exception

- Compiler does not enforce code to be written to handle exception.
- All unhandled (uncaught) unchecked exceptions are handled by JVM.
- Two classifications
  - Subclass of `Exception` called `RuntimeException` and all its subclasses are unchecked exception
  - `Error` class and it subclasses: These exceptions are external to the application and application cannot anticipate errors like virtual memory errors, JVM errors etc.

Checked exceptions or compiler-enforced exceptions

Requires programmer to explicitly write code to handle exception otherwise compiler will throw an error
All the classes that do not inherit from `RuntimeException` are checked exception
Examples: `IOException, SQLException, CloneNotSupportedException`

java.lang.Error

unchecked exception

java.lang.Exception

unchecked exception

checked exception

java.lang.RuntimeException

java.io.IOException

java.sql.SQLException

java.lang.ArithmeticException

java.lang.NullPointerException

many more classes …

many more classes …
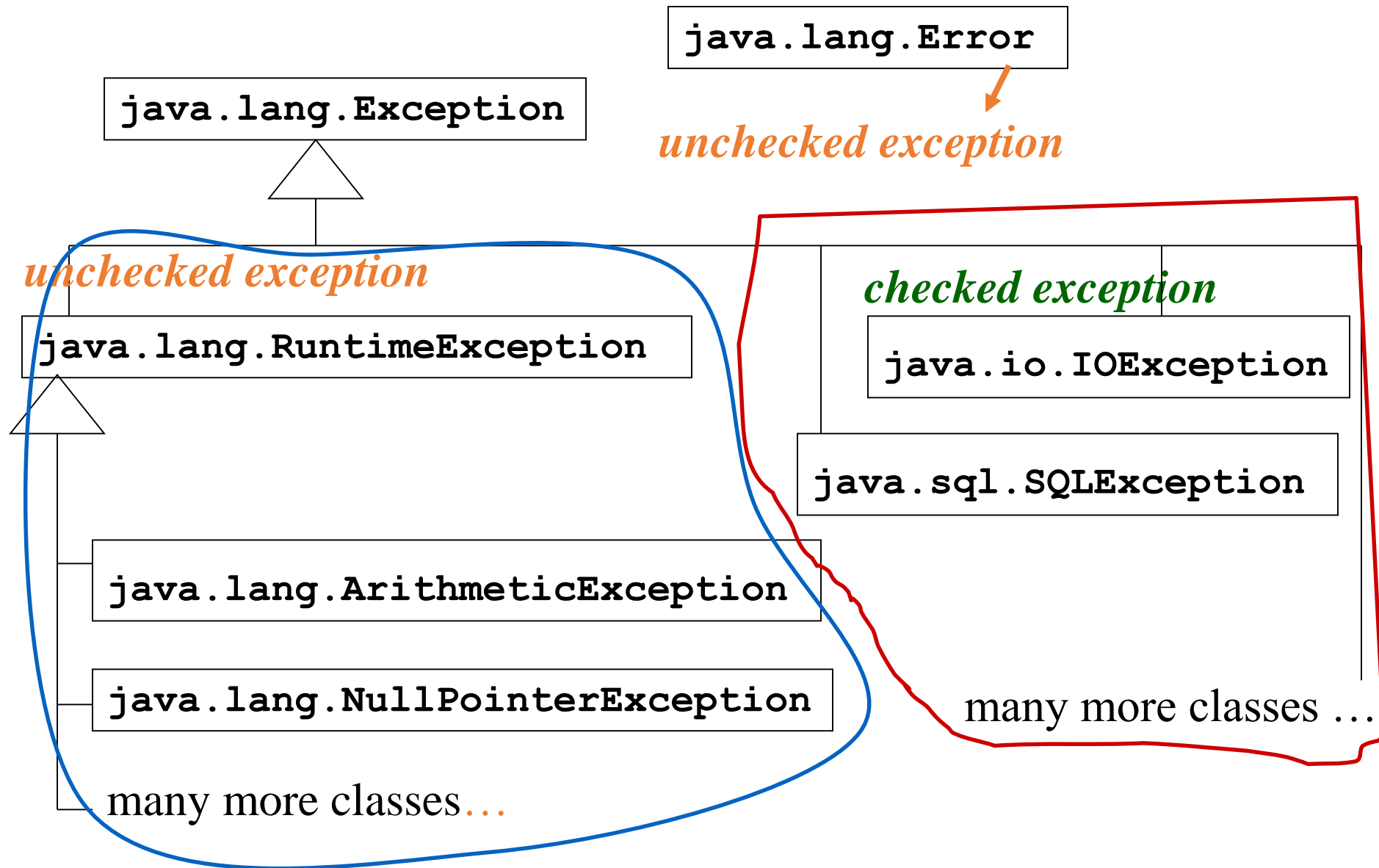
Exception Handling Keywords

- try
- catch
- finally
- throws and
- throw

Syntax

Exception handling block:

```
try{

…

}
catch(ExceptionType1 e){ …}
[catch(ExceptionType2 e){ …}]
finally { ... }
```

Code that may throw exception

Test your understanding

```
public class NullException {

static String s;

    public static void main(String[] a){

    System.out.println(s.length());

    }

}
```

What will happen on execution of this code?

Handling exceptions in the code

```java
public class NullException {
    static String s;
    public static void main(String[] a){
        try{
          System.out.println(s.length());
        }
        catch(NullPointerException n){
        System.out.print("String not initialized");
        }
    }
}
```

Activity

```
public class NullException {
    public static void main(String[] a){

        System.out.println(a.length());

        System.out.println(a[0]);

        }

}
```

Handle appropriate exception for the given the code .

Exception rules

- A method can throw an exception when something fails at runtime

- An exception is always an object of type Exception

- Compiler does not pay attention to exceptions that are runtime (Runtime doesn't have to be declared or wrapped in try-catch)

- All Exceptions the compiler cares about are Checked Exceptions

- A method throws an exception with the keyword **throw** followed by a new Exception object

- Method that throws an exception with keyword **throw** must announce it using **throws** declaration

Exception rules
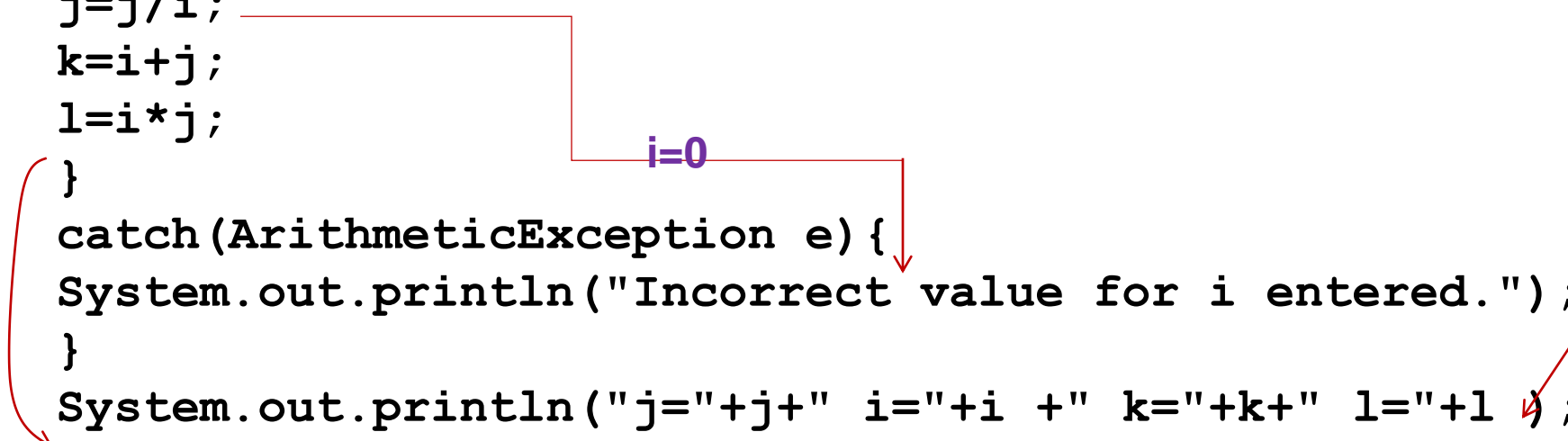
- If your code calls a risky method, wrap the call in a try-catch OR declare using throws keyword

- One method will catch what another method throws. An exception is always thrown back to the caller

Flow after exception

- When exception occurs, the statements after exception are skipped and the control goes to the matching catch block. After the catch block, the control go to the statement next to all the other catch blocks.

- The example in the next slide catches an **ArithmeticException**.

- An **ArithmeticException** is a unchecked exception that is thrown when an attempt is made to divide by 0.

```java
public class A{
public static void main(String[] args){
   int j=10;
   int k=0;
   int l=0;
   java.util.Scanner scan= new
       java.util.Scanner(System.in);
   int i= scan.nextInt();

   try{
   j=j/i;
   k=i+j;
   l=i*j;
   }
   catch(ArithmeticException e){
   System.out.println("Incorrect value for i entered.");
   }
   System.out.println("j="+j+" i="+i +" k="+k+" l="+l );
   }}
```

i=0

Multiple catches

```java
public class NoArgument{
public static void main(String[] args){
    try{
        int j=10/args.length;
        System.out.println(j);
        System.out.println(args[1]);

    }
    catch(ArithmeticException e){
    System.out.println("command line arguments not
    entered");
    }
    catch(ArrayIndexOutOfBoundsException a){
    System.out.println("command line 2nd arguments not
    entered");
    }
}
}
```

? *In Path 1, why is **args.length** not throwing **NullPointerException** ?*

Execution paths

Exceptions paths

       Path 1:

                  java NoArgument

       Result:

                  command line arguments not entered

       Path 2:

                  java NoArgument 1

       Result:

                  10

                  command line 2nd arguments not entered

Normal path

       Path 3:

                  java NoArgument X Y

       Result:

                  5

                  Y

Catch all exception

```
try{
        int j=10/args.length;

        System.out.println(args[j]);

        System.out.println(args[-1]); }

catch(ArithmeticException e){

System.out.println("command line arguments not

entered");

}

catch(Exception a){

System.out.println("Some error occurred that

caused the application to terminate");

        }
```

Test your understanding

```java
public class MulTry {
public static void main(String[] s) {
try{
        try{
        String n[]= new String[s.length-1];
        System.out.println(n[0]);        }
    catch(ArrayIndexOutOfBoundsException ae){
    System.out.println("Out of bounds");}
}
catch(NegativeArraySizeException ne){
System.out.println("Array size cannot be negative");}
}}
```

Can you guess what will happen when we execute this code with no arguments?

Exercise

- *In the Calculator program(Slide no 36 Classes and Methods part 2), modify the program such that the numbers are taken as input from the user. Handle the appropriate exceptions.*

  Hint: Use InputMismatchException, Arithmetic Exception

  (15 mins)

Activity

- **`Integer.parseInt(String s)`** is to convert string into int. Find out exception will be thrown if string passed to Integer.parseInt is not an int.

Exercise

- *A comma separated list containing pairs of topic name, time in hours (Java 14, JEE 10, JME 12) will be entered in the command line arguments. If a day consists of 8 hours, list out the topics that will be covered day-wise. Catch all the possible exceptions.*

(30 mins)

Exception class

- **Exception** object is checked exception.

- Constructors:
  ```
  public Exception()
  public Exception(String message)
  ```

- Important methods:
  ```
  public String getMessage()
  public void printStackTrace()
  StackTraceElement getStackTrace
  ```

Inherited from **Throwable**
Coming up

throw

- So far all the exceptions that were caught were thrown by the runtime system.

- If a method needs to throw an exception explicitly, it can do so by using **throw** keyword.

  - Syntax:
    - **throw new <SomeClassThatInheritsFromThrowable>**
      Or
    - **throw <SomeExceptionObject>**

Throwing an unchecked exception

```java
public abstract class Person{
    public void setName(String name){
    if(name==null)
    throw new RuntimeException("Invalid
    name");
    else   this.name=name;
}
…
}
class Test {
public static void main(String args[]){
new student.Student("X").setName(null);
}
}
```

On execution:

Exception in thread "main" **java.lang.RuntimeException**: Invalid name

Throwing a checked exception

```
public void setName(String name){
    if(name==null)
throw new Exception("Invalid name");
    else this.name=name; }
```

A compilation error occurs:
```
    Unreported exception
    java.lang.Exception;
     must be caught or declared to be thrown
```

```
public void setName(String name){
try{
  if(name==null)
  throw new Exception("Invalid name");
  else      this.name=name;      }
}catch(Exception e){
System.out.println(e.getMessage());   }
```

Tell me why?

We can achieve the same thing that the previous example does without writing any exception handlers.

```
public void setName(String name){
try{       if(name==null)
System.out.println("Invalid Name");
  else this.name=name;        }}
```

Why do we need exception handlers?

We need handlers for handling runtime exceptions.

But the reason for having exception handler is deeper than just a technical

requirement. We need such handlers

1. To separate normal business logic code and error handling code .

2. To take advantages of common error handlers.

3. To delegate.

Delegation - `throws`
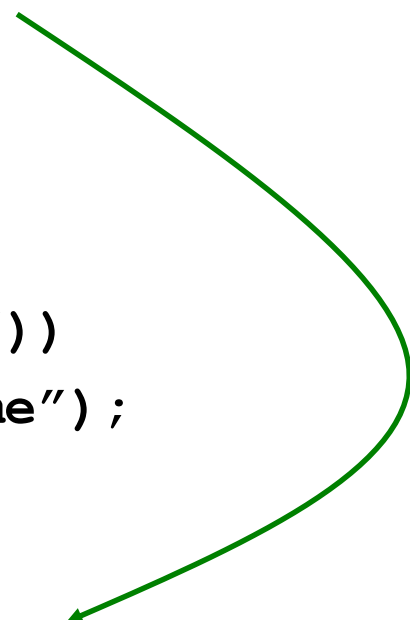
- A method that does not want to handle exceptions can delegate this to the calling method by using **`throws`** keyword with the method declaration.

- The throws list can contain all the exceptions a method throws and does not want to handle.

- The exception classes in the exception list can either exact match or be automatically convertible to the exception object thrown by the code.

- **Syntax:**
  `<method declaration statement> throws`
  `<ExceptionList>``{ ...}`

Example - `throws`

```
public abstract class Person{
public void setName(String name)
throws Exception{
if(name==null)
throw new Exception("Invalid name");
else
this.name=name;    }
…
}
```

Any method that calls `setName()` methods must handle this exception.

```
class Test {
public static void main(String args[]){
try{
new student.Student("X").setName(null);
}catch(Exception e){
if(e.getMessage().equals("Invalid name"))
System.out.println("Invalid student name");
}
}}
```

The main method must handle the exception since it is calling the `setName()` method or ..

```
public static void main() throws Exception{
new student.Student("X").setName("XX");
}
```

In this case, calling method delegates the exception
handling to the JVM. It prints:
Exception in thread "main" `java.lang.Exception`: Invalid
name

Exercise

- *In the previous exercise in slide 18, modify the program such that the exceptions are not handled in the add, diff, mul and div methods. The exception handling should be delegated to the caller method.*

(15 mins)

Partial Delegation, Chained Exceptions

- A large application may throw an exception whose root cause may be somewhere deep inside.

- The methods that handle the exception deep inside may partially handle the original exception and delegate the rest of the handling to the called method. This delegation can be done by either re-throwing
  - the same exception object
  - A totally new exception object
  - A wrapped exception object (Exception Wrapping)

- In effect, this leads to one exception causing another exception and so on. Such type of exception occurrences are called Chained Exceptions .

Re-throw

```java
public class Rethrow {
public static void method1(String s) throws Exception{
    if(s.equals("Hello"))System.out.println(s);
    else
    try{
    throw new Exception("expecting hello");
    }catch(Exception ee){
    System.out.println("caught "+ee);
    throw ee;
}}

public static void main(String s[]){
    try{method1(s[0]);
    }catch(Exception e){
    System.out.println("Exception
                        Raised:"+e.getMessage());}
}}
```

Exception Wrapping

- In case of partial delegation , when an exception is caught in the catch handler, a new exception object can be thrown that can contain the old exception object inside it so that information about the old exception is not lost. Encapsulating the old exception object inside the new one is exception wrapping.

```
try{
//  some io code
 } catch (IOException e) {
 throw new Exception("some text", e); }
```

- Built-in exceptions has constructors that can take a "cause" parameter for this purpose.
  - `Exception(String message, Throwable cause)`
  - `Exception(Throwable cause)`
- The **getCause()** method will return the wrapped exception object.

Printing Stack Trace

```java
public class Stacktrace {
public static void main(String[] s) {
try{

       y();

}catch(Exception e){
e.printStackTrace();
}
}
static  void y(){z();}
static void z(){int p=45/0;}
}
```

main(){
y();
...

printStackTrace();
}

y(){
z();

}

z(){

}

Exception raised

Exception passed on
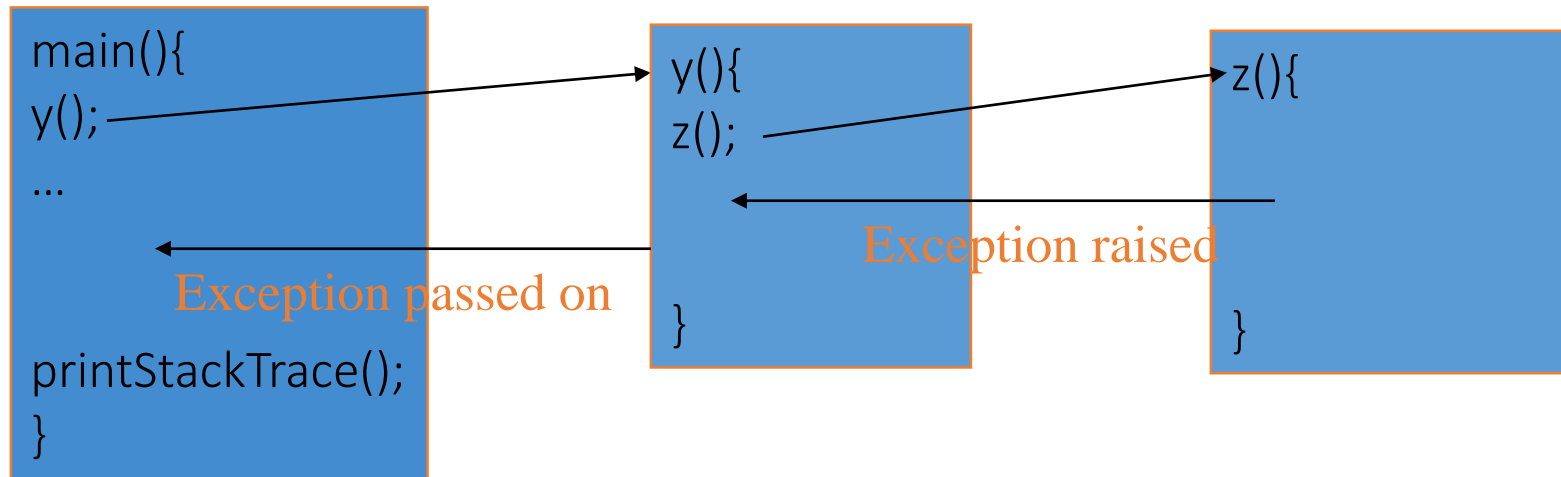
Result of execution:

```
java.lang.ArithmeticException: / by zero
        at Stacktrace.z(Stacktrace.java:17)
        at Stacktrace.y(Stacktrace.java:13)
        at Stacktrace.main(Stacktrace.java:4)
```

Stack
Trace

StackTraceElement

- Methods:
  - **public int getLineNumber()**
  - **public String getFileName()**
  - **public String getClassName()**
  - **public String getMethodName()**
  - **public boolean isNativeMethod()**

Overriding and Exception

- An overridden method CANNOT throw
  - new checked exceptions
  - parent class exception

- An overridden method
  - can throw child class exception
  - completely omit the exception

```
class Student{
public Object clone() throws Exception
{
    try{
    return super.clone();
    }catch(CloneNotSupportedException e)
    { return null;}
}}
```

Exercise

Create a class called LinkedList. Override and provide public access to clone method for this class.

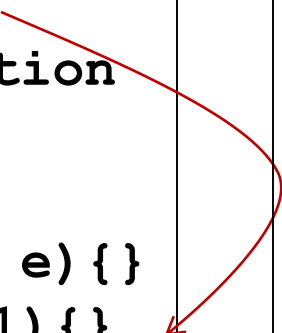*Hint: LinkedList class encapsulates Node object that has attributes :num (int) and next (Node).*

Since LinkedList has reference, make sure that clone is not Object's clone() that does bitwise copy only.
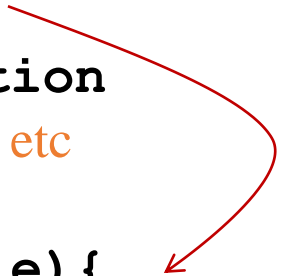
(40 mins)

`finally` – Why another keyword needed?

```
boolean read(){
try{
//open  files
//read from files
//some more operation
// file close etc
}
catch(IOException e){}
catch(Exception e1){}
}
```
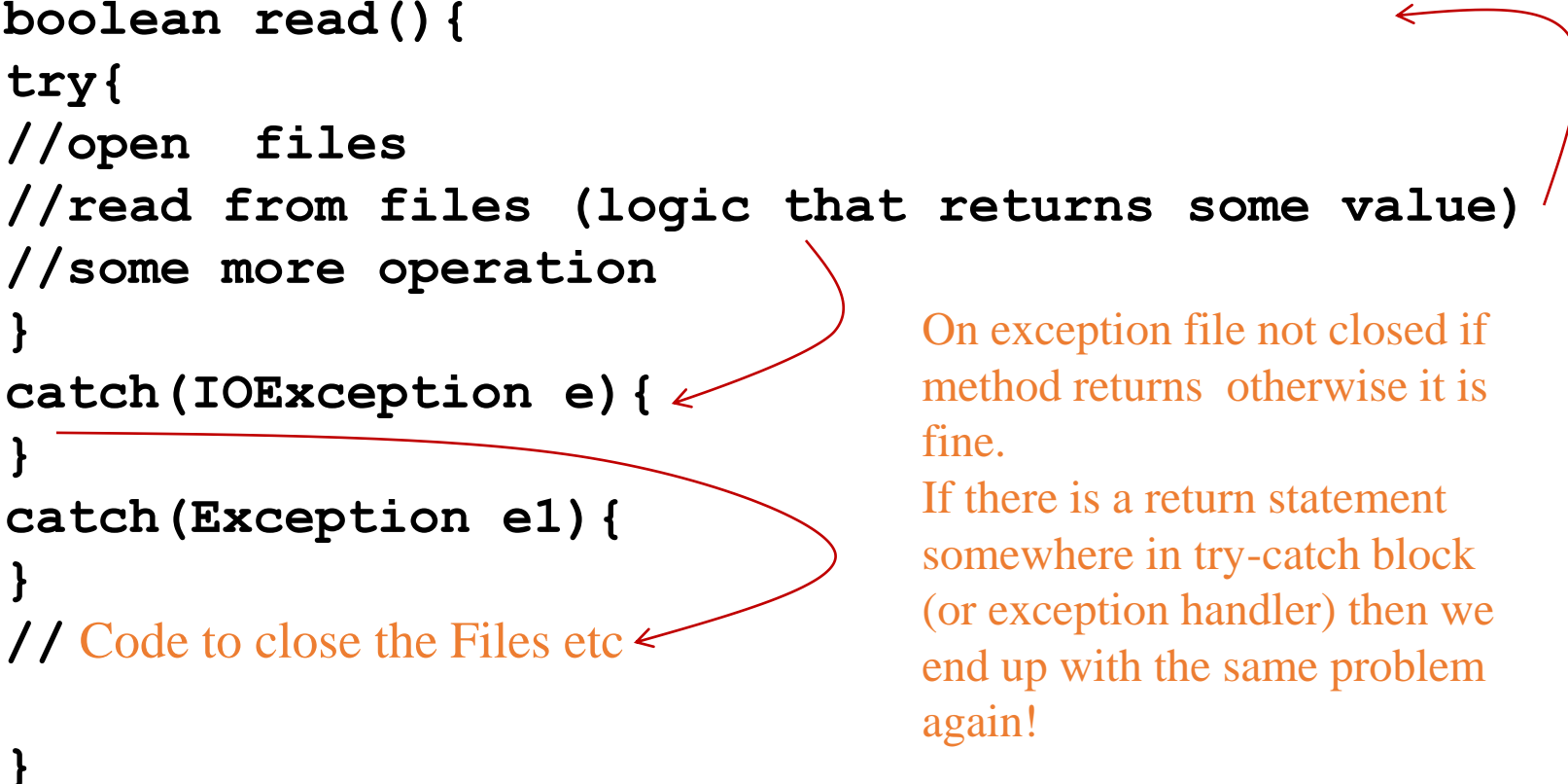
On exception, file not closed!

```
boolean read(){
try{
//open  files
//read from files
//some more operation
// Code to close the Files etc
}
catch(IOException e){
// Code to close the Files etc
}
catch(Exception e1){
// Code to close the Files etc
}
}
```

Repetition of code

```
boolean read(){
try{
//open  files
//read from files (logic that returns some value)
//some more operation
}
catch(IOException e){
}
catch(Exception e1){
}
// Code to close the Files etc

}
```

On exception file not closed if method returns otherwise it is fine.
If there is a return statement somewhere in try-catch block (or exception handler) then we end up with the same problem again!

So all these solutions don't work.

Java's solution -`finally`

- Code enclosed within a **`finally`** block will always be executed (whether or not an exception occurs).

- This facility eliminates the risk of accidently skipping cleanup code because of a **`return, continue,`** or **`break`** statement

- **`finally`** is a part of **`try-catch`** syntax

```
try{…}
catch(ExceptionType1 e){ …}
[catch(ExceptionType2 e){ …}]
finally { ... }
```

Example: `finally`

Code displays "Thank you" for all conditions.

```java
public class FullName {
public static void main(String s[]){
    try{
        int length=s[0].length()+ s[1].length();
        if(length<20)return;
        System.out.println("Name length should be less
        than 20 in total");
    }
    catch(ArrayIndexOutOfBoundsException e){
        System.out.println("2 command line arguments
        required");
    }
    finally{
        System.out.println("Thank you!");
    }
}}
```

Test your understanding

What will the code print?

```
public class Tester{
  static int m(int i){
    try{
        i++;
        if(i==1) throw new Exception();
    }catch(Exception e){ i+=10; return i;}
    finally{
        i+=5;
    }
    i++;
    return i;
  }
public static void main(String[] args) {
        System.out.println(m(0));
} }
```

Exercise

- *Create a class such that it resets the value of the objects it used to null after its usage in all cases.*

(30 mins)

Example 1: Unchecked Exception

```
class Test{

public static void main(String s[]){

    try{

    int y=10/0;

    }

        finally{   System.out.println("Thanks"); }

} }
```

Example 2: Checked Exception

```
class Test{

public static void main(String s[])throws
Exception{

try{

        throw new Exception();

    }

 finally

        { System.out.println("Thanks");}

    }
```

Try Catch Finally syntax

- You cannot have a catch or finally without a try

- You cannot put code between try and catch

- A try must be followed by either a catch or finally

- A try without a finally must still declare the exception

User-defined exceptions

```java
package general;

public  class InvalidNameException  extends Exception
  {

  String exStr;

  public InvalidNameException(){

  exStr= "invalid name";

  }

  public InvalidNameException(String s){exStr=s;}

  public String toString(){

      return "InvalidNameException" + exStr;}

  }
```

```java
public abstract class Person{

        public void setName(String name) throws
                                    InvalidNameException {

    if(name==null)

    throw new InvalidNameException();

    else     this.name=name;

    }}
…
}
```

Exercise

- *Create a class called Employee that asks the user to input the name and the age of an employee. Raise a custom defined exception when the user enters an employee name that has already been entered and raise another exception if the age is negative or less than 18 or greater than 60. If there is any occurrence of InputMismatchException and NumberFormatException, throw those also as user defined exceptions.*

- *Hint: use exception wrapping*

(30 mins)

Activity

Write the exception that will be thrown when the event happens?

- Thrown when an application tries to load in a class but no definition for the class with the specified name could be found. -

- Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.

- Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance

- Thrown but a reasonable application should not try to catch

- Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects.

- Thrown if an application tries to create an array with negative size.

- Thrown when an application attempts to use null in a case where an object is required.

Activity

- *For each exception listed in the previous exercise, write the code or create situation that will cause the exception to be thrown.*

Summary

- An exception is an abnormal condition that arises while running a program.

- Throwable class is super class for all exceptions in Java.

- Two types of exception are Unchecked exceptions and Checked exceptions.

- When exception occurs, the statements after exception are skipped and the control goes to the matching catch block.

- A method can throw an exception explicitly by using throw keyword.

- A method that does not want to handle exceptions can delegate this to the calling method by using throws keyword.

- An overridden method cannot throw new checked exceptions or parent class exception. An overridden method can throw child class exception or completely omit the exception.

- Code enclosed within a finally block will always be executed whether or not an exception occurs.