# Java: Exception Handling

## 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

*We have been hit by these errors almost in all the sessions!*

*It is time to learn how to tackle them.*

## Defining Exception

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

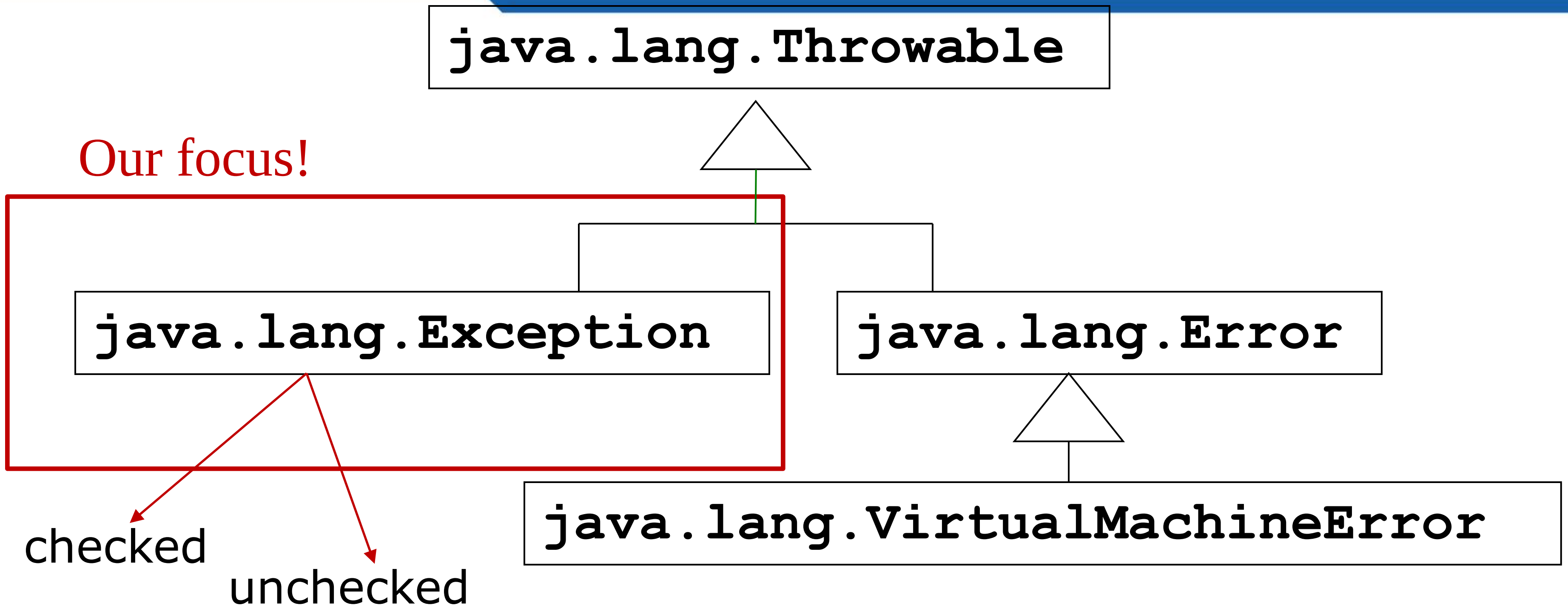
Examples:

- Attempt to divide an integer by zero causes an exception to be thrown at run time.
- 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 Hierarchy



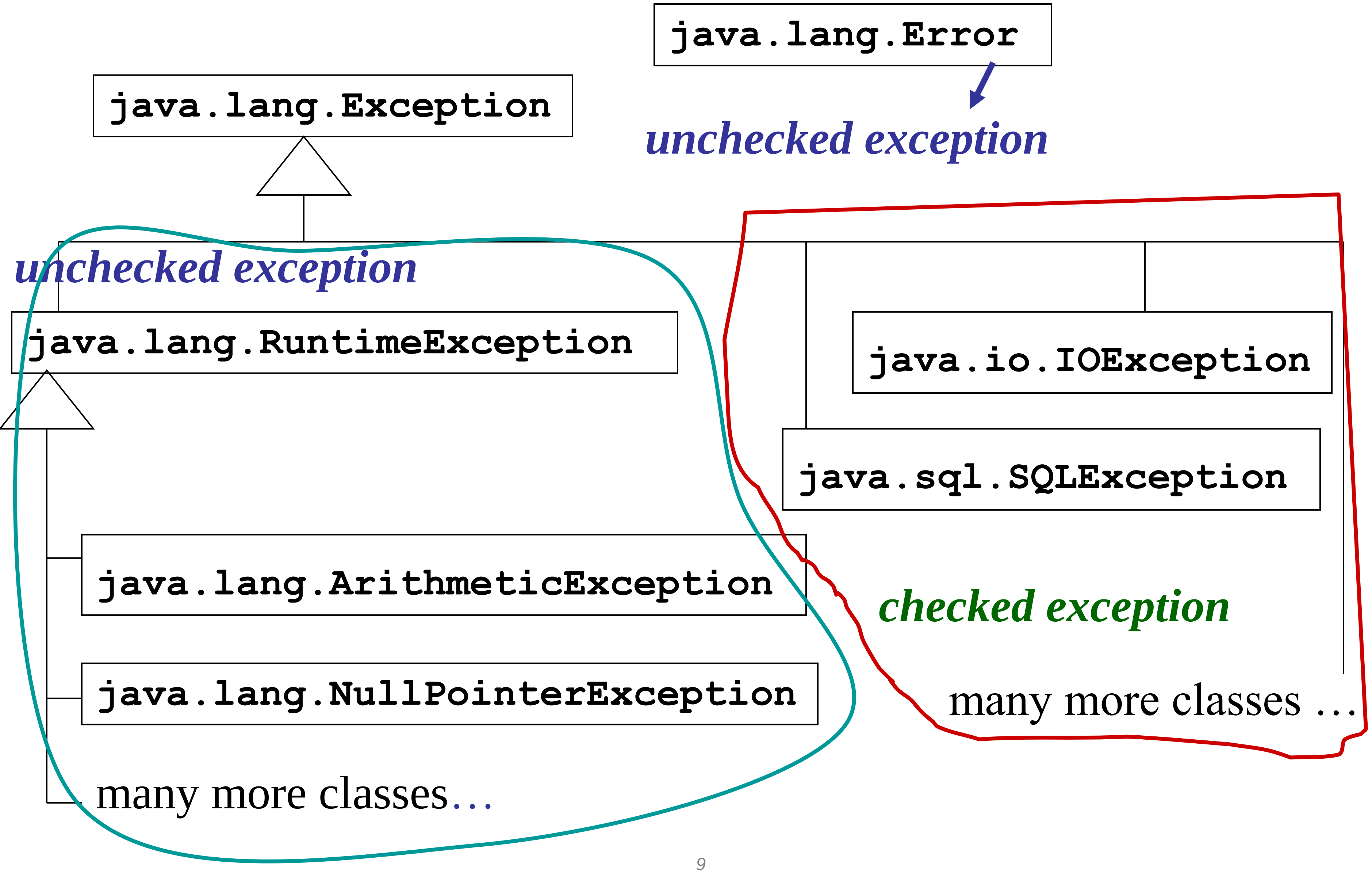
- **`Throwable`** class is super class for all exceptions in Java. When an exception occurs in a method, an object of **`Throwable`** type is thrown.
- The 2 subclasses –**`Error`** and **`Exception`**

## Types of exception

- 3 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.
    - Exceptions that we encountered so far are all unchecked exceptions.
    - Subclass of **`Exception`** called **`RuntimeException`** and all its subclasses are unchecked exception.

- Checked exceptions or compiler-enforced exception
  - 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`**

- Errors
  - These exceptions are external to the application and application cannot anticipate errors like virtual memory errors, JVM errors etc.
  - Java Documentation states that an **`Error`** is something "a reasonable application should not try to catch"
  - **`Error`** and its subclasses come under this.
  - **`Error`** and all its subclasses are **unchecked exception**.


java.lang.Error
unchecked exception
java.lang.Exception
unchecked exception
java.lang.RuntimeException
java.io.IOException
java.sql.SQLException
java.lang.ArithmeticException
checked exception
java.lang.NullPointerException
many more classes …
many more classes…

# Syntax

Exception handling block:

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

Code that may throw exception

## Collect your thoughts

```
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?

It is important that we know what exceptions (particularly runtime exceptions) may be thrown by a piece of code so that handlers can be written.

# Example: Handling exceptions in the code

```
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");
}}}
```

This code handles **NullPointerException** by providing useful information to the end user.
Note that this helps the user to know what went wrong exactly.
The error which was returned by JVM was not end-user friendly

## 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.
- Example in the next slide catches an **`ArithmeticException.`**
- An **`ArithmeticException`** is a unchecked exception that is thrown when an attempt to divide by 0 is made.

```
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;  i!=0
  }                     i=0
  catch(ArithmeticException e){
  System.out.println("Incorrect value for i entered.");
  }
  System.out.println("j="+j+" i="+i +" k="+k+" l="+l );
  }}
```

# Multiple catches

- Block of code may throw multiple exceptions. That is why the syntax of exception allows us to handle multiple handlers.
- When an exception is thrown the first matching handler is called. By matching it means any handler that has exact type or automatically convertible type of the exception object .
- The next slide has such example which throws **`ArithmeticException`** and **`ArrayIndexOutOfBoundsException`**
- **`ArrayIndexOutOfBoundsException`** is an unchecked exception that is thrown when array element accessed is beyond its size.
- Execute the code without exception handlers and see what JVM prints for the various combinations of input as specified in the slide next to the code.

## Example: Code for multiple catches

```
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");
}
}
}
```

## 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
```

## Recall

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

## Catch all exception

- What if you are not sure what exception will be thrown by a block of code but still want to provide a handler?
- In such case we could use Exception class in  the catch block which will catch all the left over exceptions.
- In the example below we could have at least 3 exceptions. First one handle will handle only divide by 0. Others will be handled by the 2nd catch block.

```
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”);}
```

# Tell me why?

```
class DivideByZero {
   public static void main(String[] args) {
   try{int j=10/args.length; }
       catch(Exception e){
       System.out.println(“general error");}
       catch(ArithmeticException e1){
       System.out.println(“div by zero ");}       }}
```

What do you think will happen when you execute the code below?

If the code executes, it must print **`general error`** (the first matching handler)

So when will the 2nd catch handler be called?

Never!

So it is unreachable code? Remember something about this?

## Sequencing catches

- Compiler flags an error if the code is unreachable.
- The code in the previous slide leads to a compilation error.
- Hence it is important that the exception handlers are sequenced properly.
- The more derived classes of exception must be among the first catch handlers.

- In Java SE 7, a single catch block can handle more than one type of exception. This will reduce code duplication.

```
catch (ArithmeticException|NullPointerExceptionn e)
```

## Test your understanding- Nesting `try` blocks

```
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?

# 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> //comment throw already caught exception., throw ee;`**

## Tell me why?

- Why do we need to throw an exception explicitly?
- Throwing business application specific exception
    - Business application specific exceptions are business application related errors. For instance, a user entering a null value for account number in a banking application is a serious error and your may want to throw a checked exception (instead of NullPointerExeption) just to make sure that such types of errors are explicitly handled.
- Grouping related exception under a single exception:
    - All types of mathematical errors like divide by 0, square root of a –ve number, should be group under one common exception. In such case a user defined exception can be created and object can be thrown from the catch block in cases of all mathematical exceptions.
- You implement an interface or extend a class that throws only certain type of exception. You may want to throw a different kind of exception or wrap the same exception object and provide additional information.

## Throwing an unchecked exception

```
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

What if we change the code to throw 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 exception a method throws and does not want to handle.
- The exception classes in the exception list can be either exact match or 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 ..

## … let JVM handle it… not a good idea, however...!

```
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

## 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
  - totally new exception object
  - 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

If string other than “Hello” is sent as command line argument, an exception is thrown by method1. This is partially handled by method1  and it is thrown once again to the main method. The main method provides another handler.

```
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); }
```

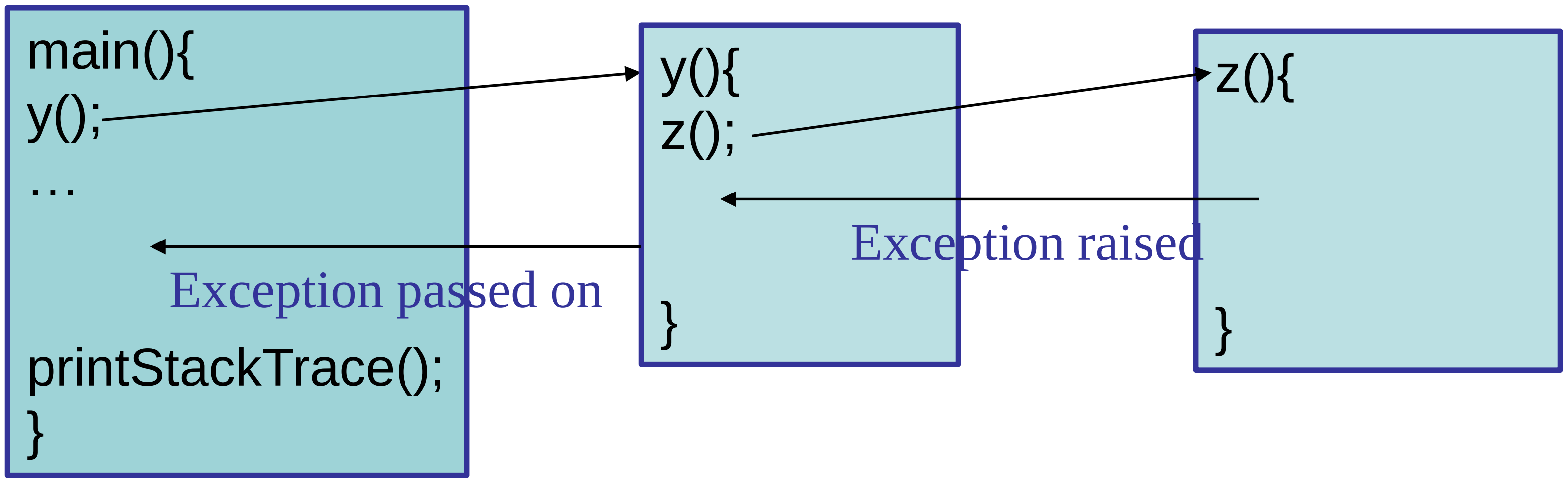
- Built-in exceptions has constructors that can take a "cause" parameter for this purpose.
  - **`Exception(String message, Throwable cause)`**
  - **`Exception(Throwable cause)`**
- The **`getCause()`** method will return the wrapped exception object.

## Chained exception information

- What we have seen is - Checked exception can be completely delegated or partially delegated.
- In case of partial delegation, exception object re-thrown may change. In such case the original cause of error is lost. (That is why sometimes programmers used Wrapped Exceptions).
- For debugging purpose, it is often helpful to have complete information about all the exceptions that have occurred in the exception chain.
- Stack Trace methods help the programmer get information about the chained exceptions.

## Example1 : printing  Stack Trace

```
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;}
}
```



Result of execution:

Stack Trace

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

## StackTraceElement

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

## Example 2: more information about exception

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

public static void main(String s[]){
try{method1(s[0]);
}catch(Throwable e){

StackTraceElement elements[] = e.getStackTrace();
for (int i = 0, n = elements.length; i < n; i++) {
System.err.println(elements[i].getFileName() + "
line number:“+ elements[i].getLineNumber()
+ " of  “ + elements[i].getMethodName() + "()");}
}
}
}
```

```
Result:
A.java line number:9 of  method1()
A.java line number:12 of  main()
```

# `finally` – Why another keyword needed?

- Let us say we are performing some file operations. Invariably all IO methods in Java throws checked exception called **`IOException`**.
- Now if an exception occurs in between, we must make sure that files that we opened are closed before continuing further.
- Where should we write this code?

```
boolean read(){
try{
//open  files
//read from files
//some more operation
// Code to close the Files 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
}
}
```

On exception ok but
Repetition 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 { ... }
```

- **`finally`** executes even if there is a return statement in the **`try`** block.

```
boolean read(){
try{
//open a file
//read from file
return true;
}
catch(IOException e){}
catch(Exception e1){}
finally{
//close files etc
}
return false;
```

Will always execute

# Example: `finally`

Code displays "Thank you" for all conditions.

```
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!");
   }
}}
```

## Beware!

```
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));
}
}
```

Prints 11!

## Few points on syntax

- A **`try`** block must have either a **`catch`** block or a **`finally`** block.
- Situation where this would be needed is when a method does not want to handle the exception (leaving it to caller to handle the exception) but wants to ensure clean up is done when an exception occurs.

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");}

 }
```

## User-defined exceptions

- Real world applications will have many errors that may occur and these errors will have to be classified such that handlers the are written based on the type of error.
- So there is a need to create a new Exception class which will map to the application exceptions.
- Only objects of **`Throwable`** class can be thrown.
- So new (user-defined classes) can either inherit from **`Throwable`** class or more correctly inherit from **`Exception`** class (because subclasses of **`Exception`** classes are supposed to define application exception).

## User-defined exceptions

- The example below demonstrates creation of a new exception class.
- The object of this class will be thrown when invalid name is provided in Person class.

```
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;}
}
```

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

# 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;}
}}
```

# Without assertions

```
private static int binarySearch(int numbers[], int key ){
   int low=0,high;
       int div=numbers.length;
       high=numbers.length-1;
       while(true){
       div=(low+high)/2;
       if(low>high)System.out.println(“div”+div);
       if(key>numbers[div])low=div+1;
       else if  (key<numbers[div])
       high=div-1;
       else return div;
       if(low> high) return -1;   } }
```

Oh no! I forgot to remove  the code I added for debugging. Code is already in UAT!

- While unit testing the code manually, print statements are often inserted in between the code for debugging purpose.
- These then are removed before code is deployed.
- Assertions are statements allows us to add debugging statements in the code such that we don’t have to worry about removing them.

# Assertions

- Assertions in Java are statements that are added such that the code can be executed to either include them or ignore them.
- These are created with manual unit testing and debugging  in mind.
- This facility is very similar to the C/C++ assertion.
- Syntax : 2 forms

```
1.  assert (condition);

2.  assert(condition): statement returning a string;
```

- If the condition is **`false`**, a runtime exception called **`AssertionError`** is thrown (code executes in assertion enabled mode)
- It is one of the fastest ways to test without using a tool and also perhaps helps in understanding the working of code better.

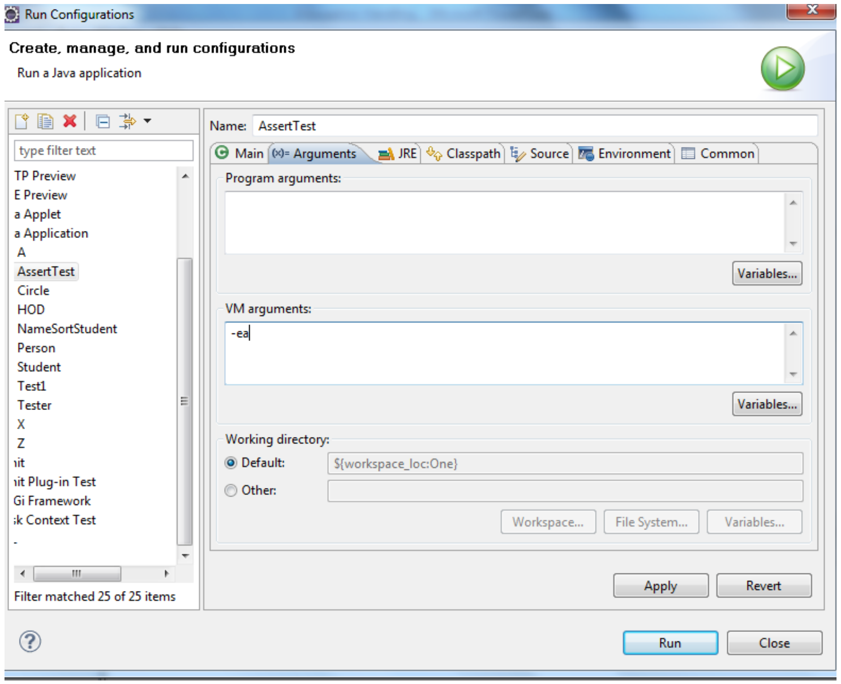
# Assertion enabling/disabling

- To enable assertion at runtime :

  **`java -ea classname`** or

  **`java -enableassertions classname`**

- In Eclipse, this can be done in "**`x=Arguments`**" tab of "**`Run Configuration`**". In **`VM Arguments`** section enter **`–ea`**.



Create, manage, and run configurations

Run a Java application

type filter text

TP Preview
E Preview
a Applet
a Application
A
AssertTest
Circle
HOD
NameSortStudent
Person
Student
Test1
Tester
X
Z
ıit
ıit Plug-in Test
Gi Framework
ık Context Test

Filter matched 25 of 25 items

Name: AssertTest

Program arguments:

VM arguments:
-ea

Working directory:
[✓ Default:] ${workspace_loc:One}
Other:

Disable assertion :

**`java -da classname`**
**`Or`**
**`java -disableassertions classname`**

## Example

```
public class AssertTest {
 private static int binarySearch(int numbers[], int key ){
 int low,high;
 try{
      int div=numbers.length;
      low=0;
      high=numbers.length-1;
      while(true){
      div=(low+high)/2;
       assert(low<=high):"div:"+div;
       if(key>numbers[div])low=div+1;
      else if (key<numbers[div])
      high=div-1;
      else return div;
      if(low> high) return -1;
      }
 }catch(ArrayIndexOutOfBoundsException a){
      return -1; }
}
public static void main(String[] args) {
   int k=binarySearch(new int[]{}, 8);
   System.out.print(k);
}
}
```

This code when executed with assertion enabled prints:

```
Exception in thread "main" java.lang.AssertionError:
div:0
at AssertTest.binarySearch(AssertTest.java:12)
at AssertTest.main(AssertTest.java:30)
```

And  when executed with assertion not enabled prints:

```
-1
```

## `AssertionError`

- This is a class that inherits from **`Error`** in **`java.lang`** package.
- This error is thrown at runtime when assert condition fails.
- This error must be seen only if code is executed with assertion enabled.
- So while you are free to throw an **`AssertionError`** from the code, this is not the desired usage.

## Convention on assert conditions

- If you want to just print  values of the variables, you can use
  - `assert(true): “x=“+x;`
- However, it is recommended that assert conditions must be such that it evaluates to false in case of wrong values that lead to incorrect results.
- Assertions are a good way of ensuring preconditions, postconditions, and invariants.
- Preconditions and post conditions are “contractual guarantees” for the conditions that must be true at the start  and end of a method. Invariants  are conditions that should always be true.
- Assert checks must be generally done on arguments to private methods.  *Why not public methods?*

## Test your understanding

```
public class AssertTest {
 public static int sum(int numbers[] ){
    int sum=0;
    assert(numbers.length>0):"array size must be > 0";
    for(int k:numbers)
    sum+=k;
    return sum;
}
}
```

Do you see any problem with the code above? What happens when you execute the code with –ea option and without it?

## Where assertions must not be used

- *The code works fine – no compiler and runtime error. But our responsibility as programmers does not stop there!*
- Assertions are not for validity checks on parameters passed to non-private methods.
- The logic checks that the business require must not be part of assertion code since assert code is switched off while in deployment.
- *Hence the previous example has a serious error – the test for array length must not be part of assert code!*

# Correct way!

```
public class AssertTest {
 public static int sum(int numbers[] ){
 int sum=0;
 ~~assert(numbers.length>0):"array size must be > 0";~~
   if (numbers.length<0) {
   throw
   new IllegalArgumentException(“array size must be > 0
   ");
   for(int k:numbers)
   sum+=k;
   return sum;
}}
```

## `IllegalArgumentException`

- Class in **`java.lang`**.
- Subclass of **`RuntimeException`**
- Used to indicate that a method has been passed an illegal or inappropriate argument.

## Recall

- Can you put down the list of un-checked exception that we have seen so far?
- These are the new exceptions that we are going to see further
  - `IllegalThreadStateException`
  - `IllegalMonitorStateException`
  - `NumberFormatException`
  - `ParseException`
  - `UnknownFormatConversionException`
  - `UnsupportedOperationException`
  - `NoSuchElementException`
  - `NotSerializableException`
  - `MissingResourceException`

# Recall

- Can you put down the list of checked exception that we have seen so far?

- These are the new exceptions that we are going to see further
  - **`InterruptedException`**
  - **`FileNotFoundException`**