RV College of Engineering®

Go, change the world

Exception Handling

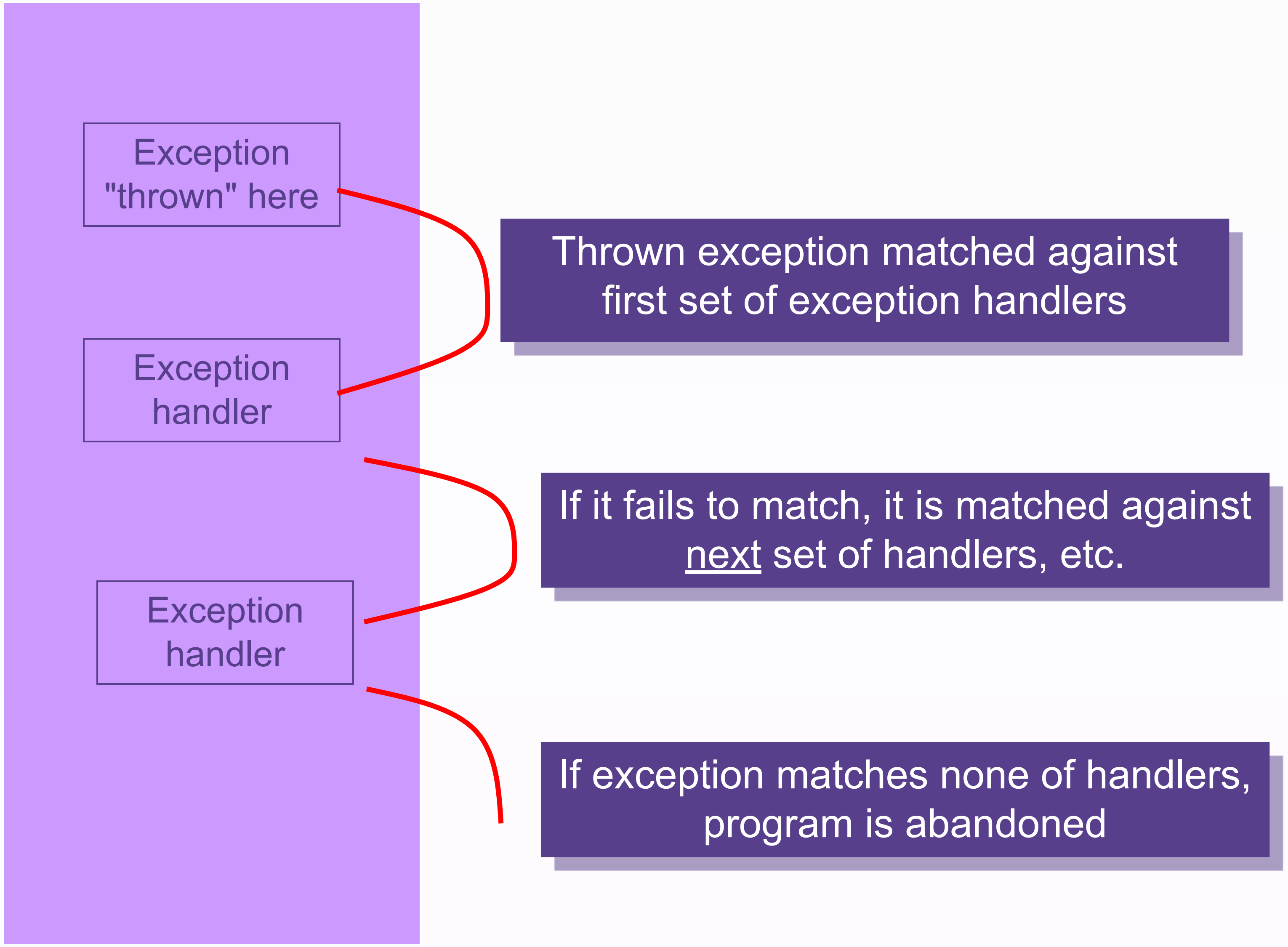
Overview

- **Exception**
  - Indication of problem during execution
  - Exception is an error that occurs at run time.
  - Using Java’s exception handling subsystem, you can, in a structured and controlled manner, handle run time errors.
- **Uses of exception handling**
  - Handle exceptions in a uniform manner in large projects
  - Remove error-handling code from “main line” of execution
- **A method detects an error and throws an exception**
  - Uncaught exceptions yield adverse effects
    - Might terminate program execution

Overview

- Code that could generate errors put in **try** blocks
  - Code for error handling enclosed in a **catch** clause
  - The **finally** clause always executes
- Termination model of exception handling
  - The block in which the exception occurs expires
- **throws** clause specifies exceptions method throws

Exception Handler


Exception "thrown" here
Exception handler
Exception handler
Thrown exception matched against first set of exception handlers
If it fails to match, it is matched against next set of handlers, etc.
If exception matches none of handlers, program is abandoned

Exception Handling

1) [redacted]
    1) [redacted]
    2) [redacted]
    3) [redacted]
    4) [redacted]
    5) [redacted]

Terminology

- **Thrown** exception – an exception that has occurred
- **Stack trace**
  - Name of the exception in a descriptive message that indicates the problem
  - Complete method-call stack
- **ArithmeticException** – can arise from a number of different problems in arithmetic
- **Throw** point – initial point at which the exception occurs, top row of call chain
- **InputMismatchException** – occurs when Scanner method nextInt receives a string that does not represent a valid integer

Termination Model of Exception Handling

- When an exception occurs:
    - try block terminates immediately
    - Program control transfers to first matching catch block
- **try** statement – consists of try block and corresponding catch and/or finally blocks
- **Termination model**
    - program control does not return to the throw point
    - try block has expired;
    - Flow of control proceeds to the first statement after the last catch block
- Resumption model
    - program control resumes just after throw point

Enclosing Code in a try Block

- **try** block – encloses code that might throw an exception and the code that should not execute if an exception occurs
- Consists of keyword **try** followed by a block of code enclosed in curly braces

Using the ***throws*** *Clause*

- Appears after method’s parameter list and before the method’s body
- Contains a comma-separated list of exceptions
- Exceptions can be thrown by statements in method’s body of by methods called in method’s body
- Exceptions can be of types listed in throws clause or subclasses

Using throws clause

If you don’t want the exception to be handled in the same function you can use the throws class to handle the exception in the calling function.

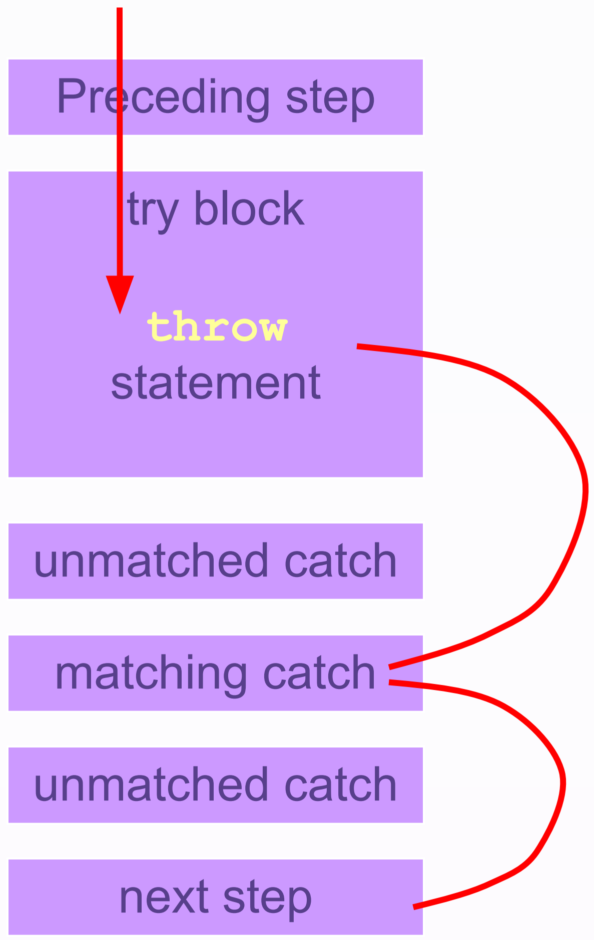
```
public class myexception{
    public static void main(String args[]){
        Try{
        String s = null;
            checkEx(s);
        } catch(FileNotFoundException ex){
        }
    }
    public void checkEx(String s) throws NullPointerException{
    System.out.println(“String length =”+s.length);
        //continue processing here.
    }
}
```

- In this example, the main method calls the checkex() method and the checkex method tries to open a file, If the file in not available, then an exception is raised and passed to the main method, where it is handled.

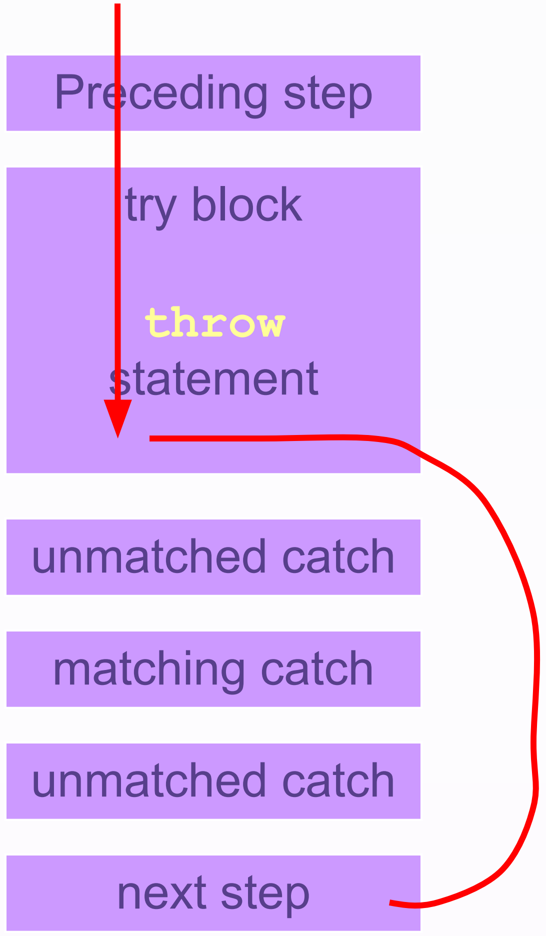
Example: Handling ArithmeticExceptions and InputMismatchExceptions

- With exception handling
  - program catches and handles the exception
- Example, Figure 13.2
  - Allows user to try again if invalid input is entered (zero for denominator, or non-integer input)

Sequence of Events for `throw`



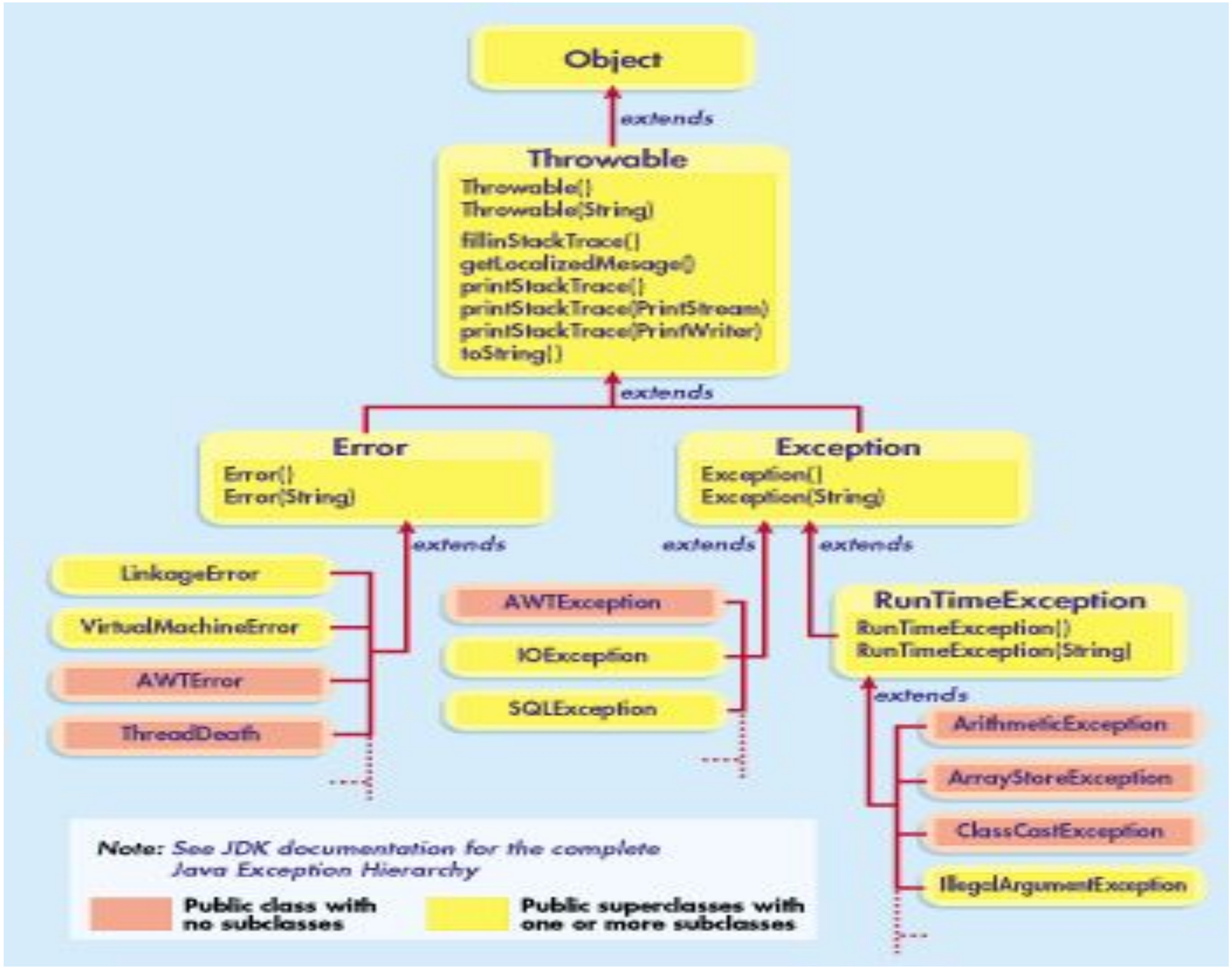
Sequence of Events for No `throw`



When to Use Exception Handling

- Exception handling designed to process synchronous errors
  - Synchronous errors – occur when a statement executes
  - Asynchronous errors – occur in parallel with and independent of the program’s flow of control
- Avoid using exception handling as an alternate form of flow of control.

Hierarchy


Object
extends
Throwable
Throwable()
Throwable(String)
fillinStackTrace()
getLocalizedMesage()
printStackTrace()
printStackTrace(PrintStream)
printStackTrace(PrintWriter)
toString()
extends
Error
Error()
Error(String)
Exception
Exception()
Exception(String)
extends
extends
extends
LinkageError
VirtualMachineError
AWTError
ThreadDeath
AWTException
IOException
SQLException
RunTimeException
RunTimeException()
RunTimeException(String)
extends
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
Note: See JDK documentation for the complete Java Exception Hierarchy
Public class with no subclasses
Public superclasses with one or more subclasses

Hierarchy

- Throwable class is a super class of all exception and errors.
- Exceptions has a special subclass, the RuntimeException
- A user defined exception should be a subclass of the exception class

Exception Handling

[redacted]

1) [redacted]

   [redacted]

2) [redacted]

   [redacted]

3) [redacted]

   [redacted]

Checked Exceptions

- Inherit from class Exception but not from **RuntimeException**
- Compiler enforces catch-or-declare requirement
- Compiler checks each method call and method declaration
  - determines whether method throws checked exceptions.
  - If so, the compiler ensures checked exception caught or declared in throws clause.
  - If not caught or declared, compiler error occurs.

Unchecked Exceptions

- Inherit from class RuntimeException or class Error
- Compiler does not check code to see if exception caught or declared
- If an unchecked exception occurs and not caught
  - Program terminates or runs with unexpected results
- Can typically be prevented by proper coding

Exception Handling

1) [redacted]
2) [redacted]
3) [redacted]

- [redacted]
  - [redacted]
  - [redacted]

Example

```
public class myexception{
        public static void main(String args[]){
                try{
                String s = null;
        System.out.println("length of string ="+s.length);
                }catch(NullPointerException ex){

                        String s ="hello"
                        System.out.println("length of string ="+s.length);

                 }finally{ // the finally block
                        System.out.println("Irrespective of Exception caught, Finally block will
                                   execute");
                       //continue processing here.
                  }
        }
}
```

- In this example we are trying to open a file and if the file does not exists we can do further processing in the catch block.
- The try and catch blocks are used to identify possible exception conditions. We try to execute any statement that might throw an exception and the catch block is used for any exceptions caused.
- If the try block does not throw any exceptions, then the catch block is not executed.
- The finally block is always executed irrespective of whether the exception is thrown or not.

Java Exception Hierarchy

- catch block catches all exceptions of its type and subclasses of its type
- If there are multiple catch blocks that match a particular exception type, only the first matching catch block executes
- Makes sense to use a catch block of a superclass when all catch blocks for that class's subclasses will perform same functionality

Catching Multiple exceptions

```
public class myexception{
     public static void main(String args[]){
          try{
          File f = new File(“myfile”);
          FileInputStream fis = new FileInputStream(f);
          }catch(FileNotFoundException ex){
               File f  = new File(“Available File”);
               FileInputStream fis = new FileInputStream(f);
          }catch(IOException ex){
               //do something here
          }finally{
                              // the finally block
            }
             //continue processing here.
        }
}
```

Catching Multiple exceptions

- We can have multiple catch blocks for a single try statement. The exception handler looks for a compatible match and then for an exact match. In other words, in the example, if the exception raised was myIOCustomException, a subclass of FileNotFoundException, then the catch block of FileNotFoundExeception is matched and executed.
- If a compatible match is found before an exact match, then the compatible match is preferred.
- We need to pay special attention on ordering of exceptions in the catch blocks, as it can lead to mismatching of exception and unreachable code.
- We need to arrange the exceptions from specific to general.

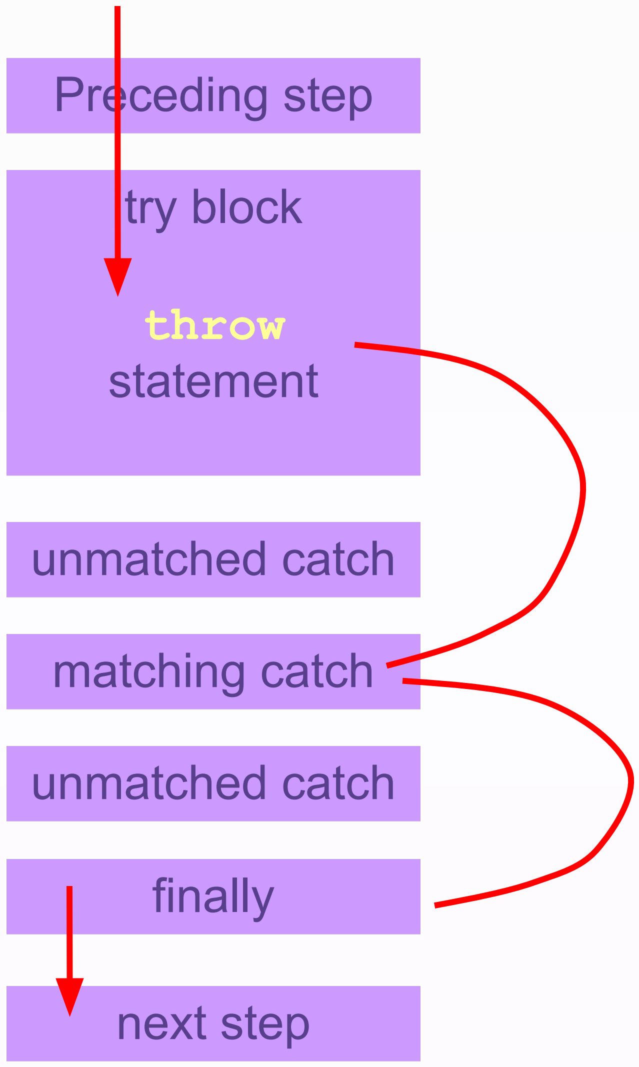
finally Block

- Consists of **finally** keyword followed by a block of code enclosed in curly braces
- Optional in a try statement
- If present, is placed after the last catch block
- Executes whether or not an exception is thrown in the corresponding try block or any of its corresponding catch blocks
- Will not execute if the application exits early from a try block via method **System.exit**
- Typically contains resource-release code

Using `finally`

- Note
  - Re-throw of exception
  - Code for **throw exception**
  - Blocks using **finally**
- Suggestion
  - Do not use a try block for every individual statement which may cause a problem
  - Enclose groups of statements
  - Follow by multiple catch blocks

Sequence of Events for `finally` clause



Java.lang.errors

The java.lang.Errors provides for different errors thrown under java lang package.

- **AbstractMethodError**
- **AssertionError**
- **ClassCircularityError**
- **ClassFormatError**
- **Error**
- **IllegalAccessError**
- **NoClassDefFoundError**
- **NoSuchFieldError**
- **NoSuchMethodError**
- **StackOverflowError**