PROGRAMMING AND DATA STRUCTURES

USEFUL JAVA API CLASSES

HOURIA OUDGHIRI & JIALIANG TAN

FALL 2023

OUTLINE

Useful classes to enhance our Java programs

- Exception Handling Mechanism
- Classes for Input/Output from/to Files
- Java Wrapper Classes
- Class String (regular expressions)

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Use Exception Handling for input validation
- Read/Write data from/to text files
- Use the Java Wrapper Classes
- Use the Java **`String`** Class methods with regular expressions

EXCEPTION HANDLING

✦ Exception?

- Runtime error thrown by the program
- Causes the program to stop immediately

✦ Handling an exception?

- Avoid immediate termination
- Inform the user
- Continue the program or exit with a friendly message

EXCEPTION HANDLING

```
import java.util.Scanner;
public class Exceptions{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter a number: ");
    x = keyboard.nextInt();
    System.out.println("Enter a number: ");
    y = keyboard.nextInt();
    System.out.println(x + " + " + y + " = " + (x+y));
    a[y] = a[y] * 2;
    System.out.println("a[" + y + "] = " + a[y]);
  }
}
```

EXCEPTION HANDLING

```java
import java.util.Scanner;
public class Exceptions{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter a number: ");
    x = keyboard.nextInt();
    System.out.println("Enter a number: ");
    y = keyboard.nextInt();
    System.out.println(x + " + " + y + " = " + (x+y));
    a[y] = a[y] * 2;
    System.out.println("a[" + y + "] = " + a[y]);
  }
}
```

```
Enter a number:
12
Enter a number:
2w
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at Exceptions.main(Exceptions.java:10)
```

EXCEPTION HANDLING

```java
import java.util.Scanner;
public class Exceptions{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter a number: ");
    x = keyboard.nextInt();
    System.out.println("Enter a number: ");
    y = keyboard.nextInt();
    System.out.println(x + " + " + y + " = " + (x+y));
    a[y] = a[y] * 2;
    System.out.println("a[" + y + "] = " + a[y]);
  }
}
```

```
Enter a number:
12
Enter a number:
5
12 + 5 = 17
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for
length 4
        at Exceptions.main(Exceptions.java:12)
```

EXCEPTION HANDLING

- Mechanisms for handling exceptions
  - **Try Block** - code block where the exception might be thrown
  - **Catch Block** - code block executed only when the exception is thrown

EXCEPTION HANDLING

```
import java.util.Scanner;
public class TryCatch{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    try{
      System.out.println("Enter a number: ");
      x = keyboard.nextInt();
      System.out.println("Enter a number: ");
      y = keyboard.nextInt();
      System.out.println(x + " + " + y + " = " + (x+y));
      a[y] = a[y] * 2;
      System.out.println("a[" + y + "] = " + a[y]);
    }
    catch(Exception e){
      System.out.println("An exception happened: " + e.getMessage());
    }
  }
}
```

Where the exception may happen

Where the exception is handled

```
Enter a number:
12
Enter a number:
2w
An exception happened: null
```

EXCEPTION HANDLING

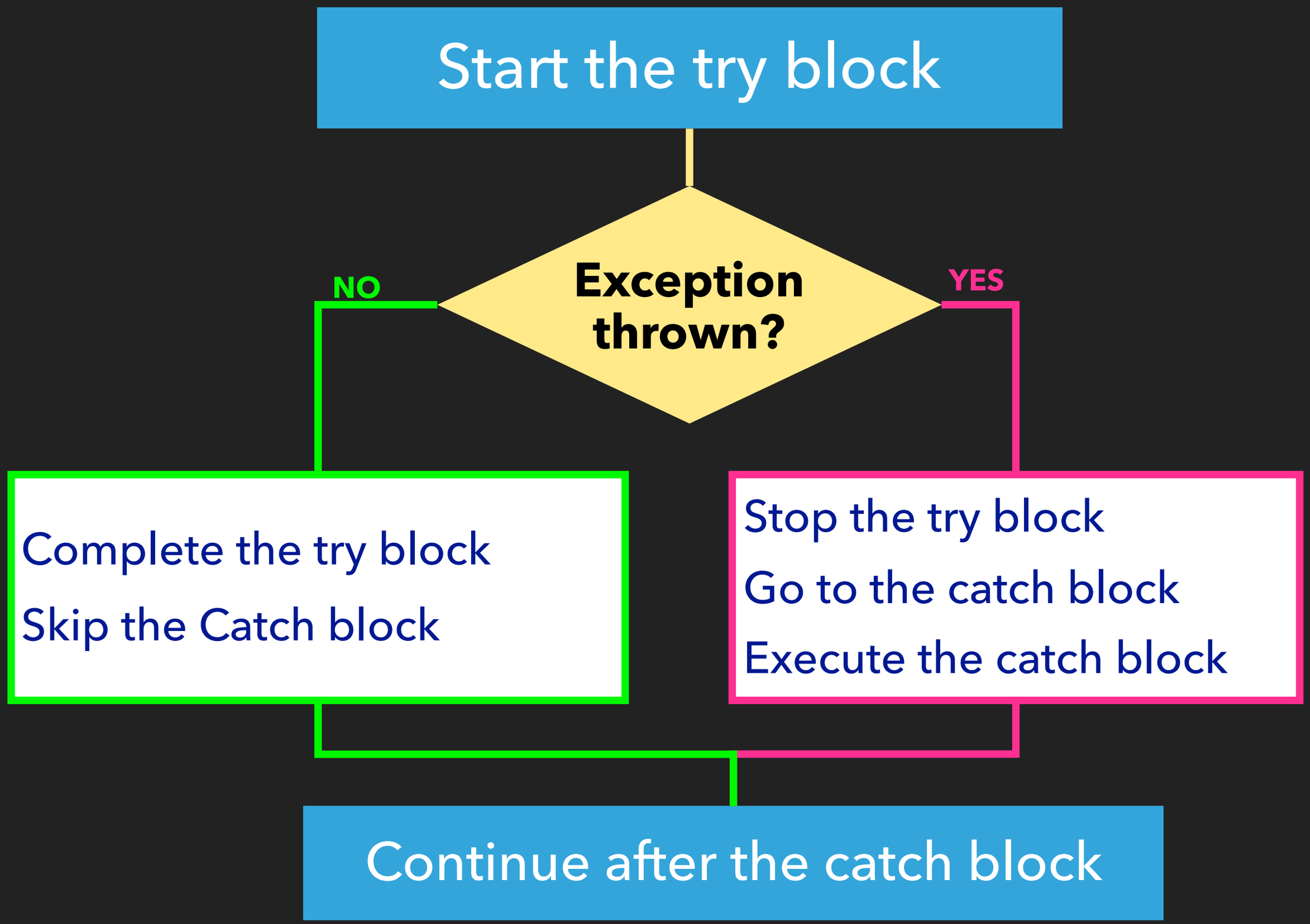
```
import java.util.Scanner;
public class TryCatch{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    try{
      System.out.println("Enter a number: ");
      x = keyboard.nextInt();
      System.out.println("Enter a number: ");
      y = keyboard.nextInt();
      System.out.println(x + " + " + y + " = " + (x+y));
      a[y] = a[y] * 2;
      System.out.println("a[" + y + "] = " + a[y]);
    }
    catch(Exception e){
      System.out.println("An exception happened: " + e.getMessage());
    }
  }
}
```

Where the exception may happen

Where the exception is handled

```
Enter a number:
12
Enter a number:
5
An exception happened: Index 5 out of bounds for length 4
```

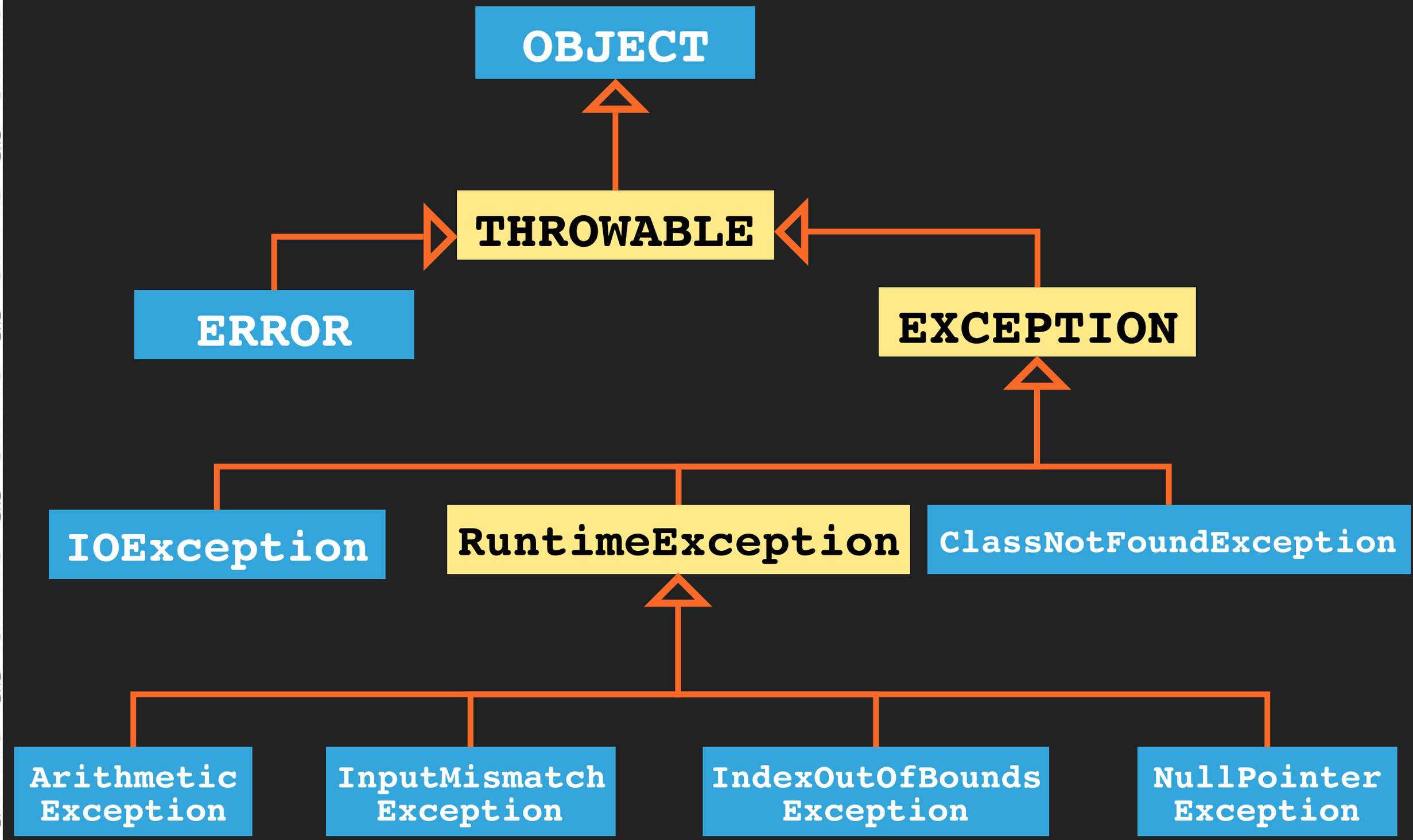
EXCEPTION HANDLING



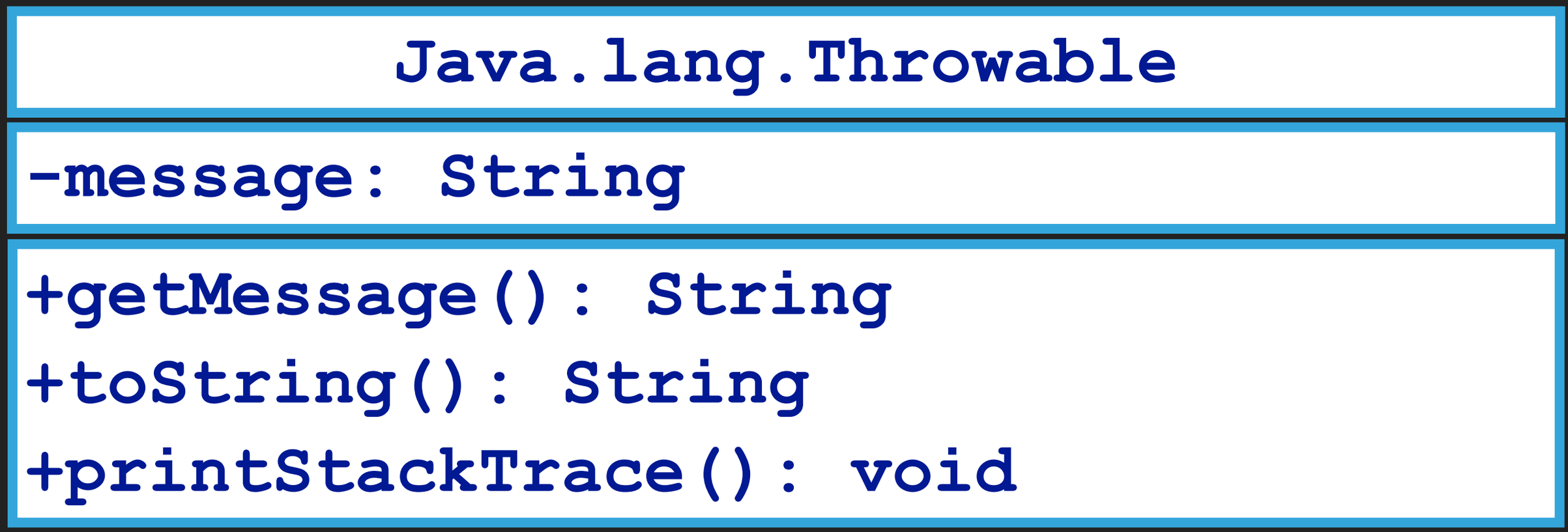
Catch block

- Like a method - called when an exception is thrown in the `try` block
- Never returns to the `try` block
- Has one parameter of type `Throwable` - Super class of a hierarchy of Exception classes in the Java API

Exception Class Hierarchy



Exception Class Hierarchy



Example - Handling Specific Exception

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class SpecificExceptions{
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    int x, y;
    Scanner keyboard = new Scanner(System.in);
    try{
      System.out.println("Enter a number: ");
      x = keyboard.nextInt();
      System.out.println("Enter a number: ");
      y = keyboard.nextInt();
      System.out.println(x + " + " + y + " = " + (x+y));
      a[y] = a[y] * 2;
      System.out.println("a[" + y + "] = " + a[y]);
    }
    catch(InputMismatchException e){
      System.out.println("Input Mismatch Exception: Input must be an integer");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Array Index Exception: " + e.getMessage());
    }
  }
}
```

Throwing Exceptions

- Exceptions are thrown by specific methods or operations (`nextInt()`, `/`)
- You can explicitly throw exceptions in your code using "`throw`" with an instance of one of the exception classes

```
throw new Exception("Something went wrong.");
```

- `Exception` object will be passed to the `catch` block as an argument (`e`)
- `e.getMessage()` returns the string passed to the `Exception` class constructor

Throwing Exceptions

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class Throwing{
  public static void main(String[] args) {
    String name; int id; double gpa;
    Scanner input = new Scanner(System.in);
    System.out.println("Enter student information\nname (first and last name): ");
    name = input.next() + " " + input.next();
    while(true){
      try{
        System.out.print("id: "); id = input.nextInt();
        System.out.print("gpa: "); gpa = input.nextDouble();
        if(gpa < 0.0 || gpa > 4.0){
          throw new Exception("Invalid GPA " + gpa);
        }
        break;
      }
      catch(InputMismatchException e){
        System.out.println("Input Mismatch Exception. Try again.");
        input.next(); //discard bad input
      }
      catch(Exception e){
        System.out.println(e.getMessage() + ". GPA must be between 0.0 and 4.0.");
      }
    }
    System.out.println("Student: " + name + " " + id + " " + gpa);
  }
}
```

What is the output of the following code?

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class Practice1{
  public static void main(String[] args) {
    int waitTime = 30;
    try{
      System.out.println("Try block entered.");
      if (waitTime > 30)
        throw new  Exception("Over 30.");
      else if (waitTime < 30){
        Exception e;
        e = new Exception("Under 30.");
        throw e;
      }
      else
        System.out.println("No exception.");
      System.out.println("Leaving try block.");
    }
    catch(Exception ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("After catch block");
  }
}
```

Creating New Exception Classes

- You can create new exception classes; derived from Java exception classes; must have two constructors at least (no-arg and one parameter of type `String`)

```
public class InvalidGPAException extends Exception {
 public InvalidGPAException() {
     super("Invalid GPA Exception");
 }
 public InvalidGPAException(String message) {
     super(message);
 }
}
```

Creating New Exception Classes

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class CreatingExceptions{
  public static void main(String[] args) {
    String name; int id; double gpa;
    Scanner input = new Scanner(System.in);
    System.out.println("Enter student information\nname (first and last name): ");
    name = input.next() + " " + input.next();
    while(true){
      try{
        System.out.print("id: ");
        id = input.nextInt();
        System.out.print("gpa: ");
        gpa = input.nextDouble();
        if(gpa < 0.0 || gpa > 4.0){
         throw new InvalidGPAException("Invalid GPA " + gpa);
        }
        break;
       }
       catch(InputMismatchException e){
        System.out.println("Input Mismatch Exception. Try again.");
        input.next(); //discard bad input
       }
       catch(InvalidGPAException e){
        System.out.println(e.getMessage() + " (GPA must be between 0.0 and 4.0.)");
       }
      }
      System.out.println("Student: " + name + " " + id + " " + gpa);
  }
}
```

Practice

What is the output of the following code?

```
public class PracticeException extends Exception{
  public PracticeException() {
    this("Test Exception thrown");
    System.out.println("Practice exception thrown #1");
  }
  public PracticeException(String message) {
    super(message);
    System.out.println("Practice exception thrown #2");
  }
  public void testMethod() {
    System.out.println("Message: " + getMessage());
  }
}
```

```
public class Practice2{
    public static void main(String[] args){
        PracticeException pe = new PracticeException();
        System.out.println(pe.getMessage());
        pe.testMethod();
    }
}
```

Multiple Catch blocks

- The order of the `catch` blocks matters
  - From specific to general
  - Follow the hierarchy (from sub classes to super classes)

```
import java.util.InputMismatchException;
public class MultipleCatch{
    public static void main(String[] args){
        int n = -42;
        try{
            if (n > 0)  throw new Exception();
            else if (n < 0)  throw new InputMismatchException();
            else System.out.println("Bingo!");
        }
        catch (Exception e) {
            System.out.println("First catch.");
        }
        catch(InputMismatchException e) {
            System.out.println("Second catch.");
        }
    }
}
```

Catch-Declare Rule

- Methods can catch or declare throwing exceptions
- **Catch Rule**: method throws an exception and handles it (No declaration)
- **Declare Rule**: method throws an exception, but does not handle it (declaration needed)

Catch-Declare Rule

```
public class Catch{
    public static void main(String[] args){
        int x = 20;
        int k = 15/20;
        System.out.println(safeDivide(x, k));
    }
    public static double safeDivide(double a, double b){
        try{
            if (b == 0)
                throw new RuntimeException();
            else
                return (a/b);
        }
        catch(RuntimeException e){
            return 0;
        }
    }
}
```

Catch-Declare Rule

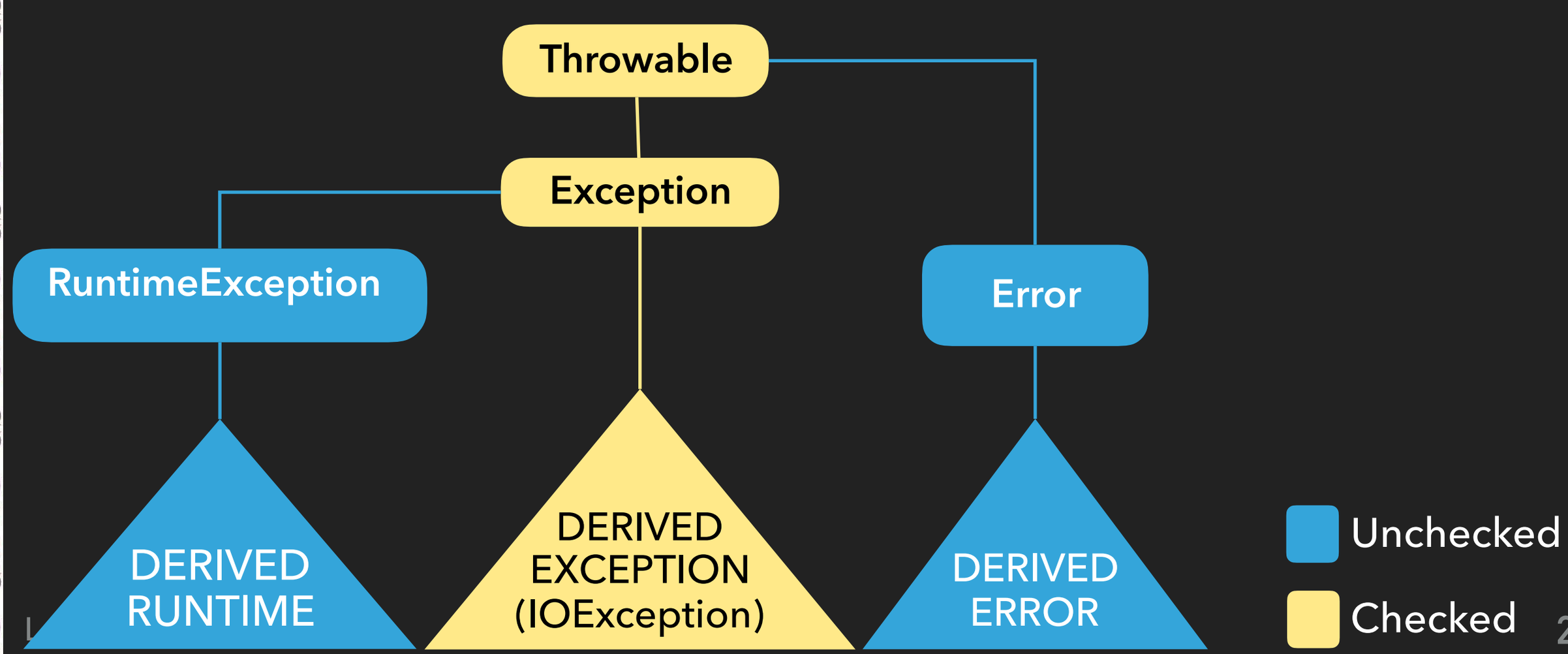
```
public class Declare{
    public static void main(String[] args){
        int x = 20;
        int k = 15/20;
        try{
            System.out.println(safeDivide(x, k));
        }
        catch(Exception e){
            System.out.println(0.0);
        }

    }
    public static double safeDivide(double a, double b) throws Exception
    {
        if (b == 0)
            throw new Exception();
        else
        return (a/b);
    }
}
```

Catch-Declare Rule

- Declare rule indicated in the signature using the `throws` clause (more than one exception can be listed in the throws clause)
- A method can use a mix of declare rule and catch rule

Catch-Declare Rule

- Catch-Declare rule is enforced for checked exceptions only
  - Checked Exception - Exception for which Java enforces the rule “catch or declare”



Finally Block

- A block after the **`try`** block and all its **`catch`** blocks
- The **`finally`** block is always executed whether an exception is thrown or not
- **`finally`** is not executed only if the code exits from the try block or the catch block

```
try
{
    block of statements
}
catch(specificException se)
{
    block of statements
}
catch(Exception e)
{
    block of statements
}
finally
{
    block of statements
}
```

What is the output of the following code?

```
import java.util.InputMismatchException;
public class Finally {
 public static void main(String[] args){
   try {
    exerciseMethod(-10);
   }
   catch(Exception e){
     System.out.println("Caught in main.");
   }
 }
 public static void exerciseMethod(int n) throws Exception {
  try{
    if (n > 0) throw new Exception();
    else if (n < 0) throw new InputMismatchException();
    else
      System.out.println("No Exception.");
    System.out.println("Still in exerciseMethod.");
  }
  catch(InputMismatchException e) {
     System.out.println("Caught in exerciseMethod.");
  }
  finally{
     System.out.println("In finally block.");
  }
  System.out.println("After finally block.");
 }
}
```

SUMMARY

- Exception Handling - **`try`** - **`catch`** - **`throw`** - **`finally`**
- Deriving new exception classes
- Declare throwing exceptions - **`throws`**
- Catch or declare rule - **checked**/**unchecked** exceptions

Accessing Files

- Accessing files on your hard disk or remotely
- Two types of access:
  - Access the file properties (size, location, folder/file, ...)
  - Access the data inside the file (reading and writing)

Class `File`

- Wrapper Class for files: allow access to file properties

File	
`+File(String filename)`	Constructor
`+exists(): boolean`	return true if file exists
`+canRead(): boolean`	return true if can read from file
`+canWrite(): boolean`	return true if can write to file
`+isFile(): boolean`	return true if it is a file
`+isDirectory(): boolean`	return true if it is a directory
`+getName(): String`	returns the name
`+getPath(): String`	returns the path
`+length(): long`	returns the size in bytes
`+delete(): boolean`	deletes the file
`+createNewFile(): boolean`	create a new file
`+renameTo(String name): boolean`	rename the file
`+mkdir(): boolean`	create directory
`+listFiles(): File[]`	returns the list of files/folders contained in a folder

Class File

```
import java.io.File;
import java.util.Scanner;
public class ClassFile {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter file name: ");
    String filename = keyboard.next();
    File file = new File(filename);
    if (!file.exists()) {
       System.out.println("File not found");
       System.exit(0);
    }
    if(file.isFile()) {
       System.out.println(filename + " is a file.");
       System.out.println("Size of the file: " + file.length() + " bytes");
    }
    if(file.isDirectory()) {
       System.out.println(filename + " is a folder.");
       File[] list = file.listFiles();
       System.out.println("There are " + list.length + " items in the folder");
    }
  }
}
```

Reading/Writing Files

- Steps to read/write from/to a file

I. **`Open`** the file for reading or writing

II. **`Read`** from or **`Write`** to file

III. **`Close`** the file

Reading data from text files

I. Open the file: create a **`Scanner`** object linked to a class **`File`** object; the **`File`** object is linked to the text file

```
File file = new File("data.txt");
Scanner fileScanner = new Scanner(file);
```

`Scanner(File)` throws a checked **`FileNotFoundException`**

II. Read from the file: Use the methods from the class Scanner

```
int ivalue = fileScanner.nextInt();
double dvalue = fileScanner.nextDouble();
String str = fileScanner.next();
```

III. Close the file

```
fileScanner.close();
```

Reading data from text files

- Reading multiple lines from a file without knowing the number of lines
- How to detect the end of the file
- **`hasNext()`** returns **`true`** - **`Scanner`** object has more data to read
- **`hasNextInt()`**, **`hasNextLine()`**

Reading from text files

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class Read {
  public static void main(String[] args) {
    File file = new File("students.txt");
    int count = 0;
    try {
      Scanner readFile = new Scanner(file);
      System.out.println("File opened successfully.");
      while(readFile.hasNext()) {
        String fname = readFile.next();
        String lname = readFile.next();
        int id = readFile.nextInt();
        double gpa = readFile.nextDouble();
        System.out.println("Student " + (count+1) +
                           ": (" + fname + " " + lname +
                           ", " + id + ", " + gpa + " )");
        count++;
      }
      readFile.close();
      System.out.println(count + " students read from the file.");
    }
    catch(FileNotFoundException e) {
      System.out.println("Cannot open file \"students.txt\"");
    }
  }
}
```

Writing data to text files

I. Open the file: create a **`PrintWriter`** object linked to a class **`File`** object; the **`File`** object is linked to the text file

```
File file = new File("data.txt");
PrintWriter fileWriter = new PrintWriter(file);
```

`PrintWriter(File)` throws a checked **`FileNotFoundException`**

II. Write to the file: Use the methods from class **`PrintWriter`**

```
fileWriter.println("this is a line");
fileWriter.print("Value = " + value);
fileWriter.printf("%d %f\n", v1, v2);
```

III. Close the file

```
fileWriter.close();
```

Writing to text files

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class Write {
    public static void main(String[] args) {
        File file = new File("numbers.txt");
        try {
            PrintWriter writeFile = new PrintWriter(file);
            for(int i=0; i<1000; i++) {
                writeFile.println(Math.random());
            }
            writeFile.close();// must close file after writing
        }
        catch(FileNotFoundException e) {
            System.out.println("Cannot write to file.");
        }
    }
}
```

Class **Scanner** methods

Method	**Purpose**	**Exception thrown**
Scanner(File)	Constructor	FileNotFoundException
int nextInt() long nextLong double nextDouble() short nextShort() byte nextByte() float nextFloat()	returns next token	NoSuchElementException InputMismatchException IllegalStateException
String next() String nextLine()	returns next token return remaining of current line	NoSuchElementException IllegalStateException
boolean hasNextInt() boolean hasNextLong boolean hasNextDouble() boolean hasNextShort() boolean hasNextByte() boolean hasNextFloat()	returns true if there is a next token of the type specified	IllegalStateException

SUMMARY

- File IO - Accessing text files for reading and writing
- Class **`File`** - Wrapper class for files (access file properties)
- **`Scanner`** object - Reading from file - must catch `FileNotFoundException` - read using the methods **`next()`**, **`nextInt()`**, ...
- **`PrintWriter`** Object - Writing to file - must catch `FileNotFoundException` - write using the methods **`print()`**, **`printf()`**, **`println()`**

Wrapper Classes

- Abstraction of primitive types in Java
- Useful methods to manipulate primitive types

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Wrapper Class Integer

static data members

static methods

Integer

```
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
```

```
+Integer(int)
+Integer(String)
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+toString(): String
+compareTo(Integer): int
+valueOf(String): Integer
+valueOf(String, int): Integer
+parseInt(String): int
+parseInt(String, int): int
```

Wrapper Class Integer

```
public class Wrapper{
   public static void main(String[] args) {
      System.out.println("The maximum integer is "+Integer.MAX_VALUE);
      Integer number1 = 12; // equivalent to new Integer(12)- autoboxing
      Integer number2 = 25;
      System.out.println("Number 1 = " + number1.toString());
      int equal = number1.compareTo(number2);
      if (equal == 0)
          System.out.println("Equal numbers.");
      else if (equal > 0)
          System.out.println(number1 + " > " + number2);
      else
          System.out.println(number1 + " < " + number2);
      String s = "15";
      Integer number3 = Integer.valueOf(s);
      System.out.println("number3: " + number3);
      System.out.println(Integer.parseInt("111", 2));  //binary
      System.out.println(Integer.parseInt("12", 8));  //octal
      System.out.println(Integer.parseInt("15", 10)); // decimal
      System.out.println(Integer.parseInt("1A", 16)); // hexadecimal
   }
}
```

WRAPPER CLASSES

- Wrapper Classes - primitive types encapsulated in class types
- Provide utility methods to manipulate primitive types

Class `String`

- **`String`** objects are immutable (cannot be changed once created)
- Wide set of methods to manipulate **`String`** objects (13 constructors and 40 methods)
- Four special methods that accept regular expressions as arguments

`String`

`+replaceFirst(String, String): String`
`+replaceAll(String, String): String`
`+split(String): String[]`
`+matches(String): boolean`

Regular Expressions

- Used to describe a general pattern in a text
- Analyze text for specific patterns - validate user input

  for example

  Phone number `(ddd)ddd-dddd`

  Social Security Number `ddd-dd-dddd`
- Very powerful tool used for text analysis/parsing

```
+replaceFirst(String regex, String):String
+replaceAll(String regex, String):String
+split(String regex):String[]
+matches(String regex):boolean
```

Regular Expressions

`"Java.*"` `*` stands for any zero or more characters

`"\\d{3}-\\d{2}-\\d{4}"`

`\d` single digit

`{2}` number of digits

`"[$+#%]"` `[ ]` any one of the characters

Regular Expressions

Regex	Description	Regex	Description
`x`	Specific character x	`\s`	Whitespace character
`.`	Any single character	`\S`	Non whitespace character
`(ab|cd)`	ab or cd	`p*`	Zero or more occurrences of p
`[abc]`	a or b, or c	`p+`	One or more occurrences of p
`[^abc]`	Any character except a, b, or c	`p?`	Zero or one occurrence of p
`[a-z]`	a through z	`p{n}`	Exactly n occurrences of p
`[^a-z]`	Any character except a through z	`p{n,}`	At least n occurrences of p
`\d`	Single digit	`p{n,m}`	Between n and m occurrences of p (inclusive)
`\D`	Non digit		

Regular Expressions

```
public class Regex{
  public static void main(String[] args) {
    System.out.println("2+3-5".replaceFirst("[+-]","%"));
    System.out.println("2+3-5".replaceAll("[+-]", "%"));
    String[] items = "02/25/2021".split("/");
    for(String item: items){
      System.out.println(item + " ");
    }
    String[] tokens = "Java,C?C#,C++".split("[.,:;?]");
    for(String token: tokens){
      System.out.println(token + " ");
    }
    System.out.println("2+3-5".matches("\\d[+-]\\d[+-]\\d"));
    System.out.println("2+3-5".equals("\\d[+-]\\d[+-]\\d"));
    System.out.println("440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}"));
  }
}
```

CLASS STRING

✦ Practice: Show the output of the following code

```
System.out.println("Hi,ABC, good".matches(".*ABC .*"));
        System.out.println("Hi,ABC,good".matches(".*ABC.*"));
        System.out.println("A,B;C".replaceAll(",;", "#"));
        System.out.println("A,B;C".replaceAll("[,;]", "#"));
        tokens = "A,B;C".split("[,;]");
        for (int i=0; i<tokens.length; i++)
            System.out.println(tokens[i] + " ");
```

StringBuilder Class

StringBuilder
+StringBuilder() +StringBuilder(int) +StringBuilder(String) +append(char[]): StringBuilder +delete(int, int): StringBuilder +deleteChatAt(int): StringBuilder +insert(int, char[], int, int): StringBuilder +insert(int, char[]): StringBuilder +insert(int, String): StringBuilder +replace(int, int, String): StringBuilder +reverse(): StringBuilder +setCharAt(int, char): void

SUMMARY

- `String` Class - for manipulating text
- Utility methods to manipulate text
- Regular Expressions (regex)- `replaceFirst`, `replaceAll`, `split`, and `matches`
- `StringBuilder` to create mutable String objects