PROGRAMMING AND DATA STRUCTURES

OOP APPLICATIONS

HOURIA OUDGHIRI

FALL 2021

OUTLINE

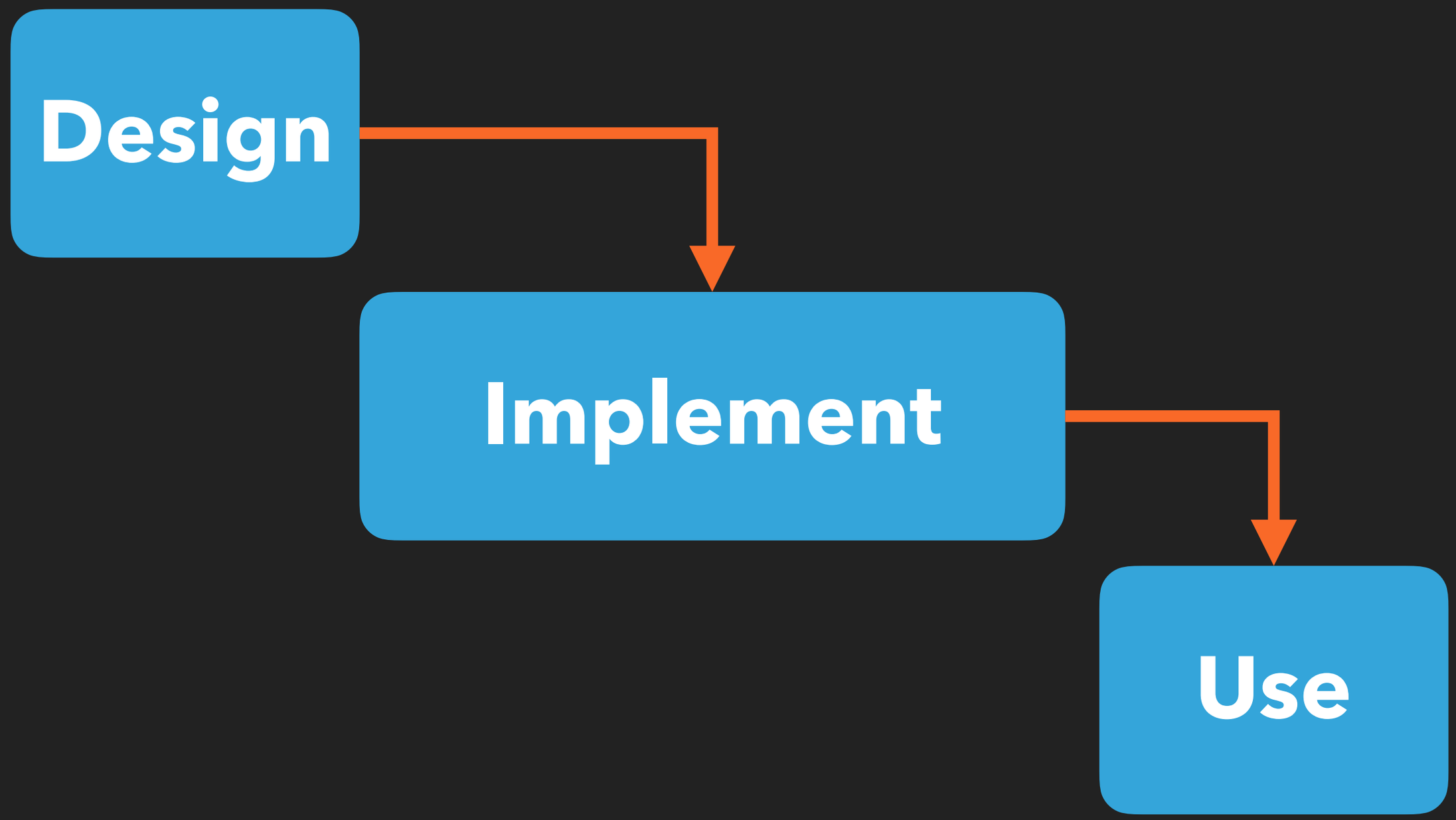
- Encapsulation and Abstraction
- Java Wrapper Classes
- String Class
- Exception Handling
- Input/Output from/to Files

STUDENT LEARNING OUTCOMES

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

- Explain the difference between the design and the implementation of a class (OOD)
- Use Java Wrapper Classes
- Use Java **`String`** Class methods
- Use Exception Handling for input validation
- Read/Write data from/to text files

Life Cycle of Java Classes



ENCAPSULATION AND ABSTRACTION

- Design Classes (UML diagram and documentation) - interface only
- Implement Classes (code, .java file)
- Use Classes
  - Applications or programs
  - Association
  - Inheritance

Class Design

- UML Diagram/ documentation
- Encapsulation (block of data and methods)
- Abstraction - class can be used without knowing how it is implemented (JAVA API)

```
Student
-name: String
-id: int
-gpa: double
+Student()
+Student(String, int, double)
+getName(): String
+getID(): int
+getGPA(): double
+setName(String): void
+setID(int): void
+setGPA(double): void
+toString(): String
```

Class Implementation

- Java code of the class
- Definition of the methods inside the class
- Modifying the body of the methods does not affect the class interface or usage

Using the Class

- In a main program
  (for testing or application)
- Association
  In another class as a data member
- Inheritance
  Extended to create a derived class

Using the Class

- In a program

```
class UsingStudent{
 // main method
 public static void main(String[] args){
  Student[] studentList = new Student[20];
  studentList[0] = new Student("Lily", 123,
                                    3.5);
  …
  for(int i=0; i<studentList.length; i++)
     System.out.println(
       studentList[i].toString());
 }
}
```

Using the Class

- In another class as a data member (**association**)

```
class ClassRoster{

  private Student[] studentList;

  }
}
```

Using the Class

- Extended to create a derived class (**inheritance**)

```
class GradStudent extends Student{

 private String supervisor;

  ...
}
```

SUMMARY

- Encapsulation and Abstraction - Interface to use the class
- Class Implementation - details of the class
- Using classes - Association/Inheritance

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`

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
```

Using Class Integer

```
public static void main(String[] args) {
  System.out.println("The maximum integer is "+Integer.MAX_VALUE);
  Integer number1 = 12; // equivalent to new Integer(12)
  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(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
```

Practice

- What is the output of the following code?

```
public static void main(String[] args) {

  System.out.println(Integer.parseInt("10"));
  System.out.println(Integer.parseInt("10", 2));
  System.out.println(Integer.parseInt("10", 16));
  System.out.println(Integer.parseInt("11"));
  System.out.println(Integer.parseInt("11", 8));
  System.out.println(Integer.parseInt(“FF”, 16));

}
```

Boxing - Unboxing

- Boxing: converting from primitive type to wrapper class type (`int` → `Integer`)
- Unboxing: converting from the wrapper class type to the primitive type (`Double` → `double`)
- Auto-boxing and Auto-unboxing: automatic boxing and unboxing in Java

Boxing - Unboxing

```
public static void main(String[] args) {

  // Automatic Boxing
  Integer[] intArray = {1, 2, 3};
  // Automatic Unboxing
  int x = intArray[0] * intArray[1];
}
```

SUMMARY

- Wrapper Classes - primitive types encapsulated in class types
- Utility methods to manipulate primitive types
- Auto-boxing/Auto-unboxing

CLASS STRING

- Class to manipulate text - String
- String Objects are immutable (cannot be changed once created)
- Wide set of methods to manipulate String objects (13 constructors and 40 methods)

CLASS STRING

Sample text manipulation methods

String

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

Sample methods

- Replacing and Splitting

```
public static void main(String[] args) {

  “Hello World”.replace(‘o’, ‘A’);
   // returns “HellA WArld”
   “Hello World”.replace(“He”, “Ha”);
   // returns “Hallo World”
  “Hello World”.replaceFirst(“o”, “A”);
   // returns “HellA World”
  “Hello World”.replaceAll(“o”, “A");
   // returns “HellA WArld”
}
```

Sample methods

- Replacing and Splitting

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

regex: regular expression - general pattern in the string

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 for text analysis

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

```
“2+3-5”.replaceFirst(“[+-*/]”,“%”);
// returns “2%3-5

“2+3-5”.replaceAll(“[+-*/], “%”);
// returns “2%3%5
```

Regular Expressions

```
String[] items =
“02/25/2021”.split(“/”);
// returns items = {“02”,“25”,“2021”}

String[] tokens =
   “Java,C?C#,C++”.split(“[.,:;?]”);
// returns tokens={“Java”, “C”, “C#”, “C++”}
```

Regular Expressions

- Matching

```
"2+3-5".matches("\\d[+-]\\d[+-]\\d");
// returns true

"2+3-5".equals("\\d[+-]\\d[+-]\\");
// returns false

"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}");
// returns true
```

CLASS STRING

- 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("[,;]", "#"));
String[] tokens = "A,B;C".split("[,;]");
for (int i=0; i<tokens.length; i++)
  System.out.print(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 - Text manipulation
- Utility methods to manipulate text
- Regular Expressions (regex)- `replaceFirst`, `replaceAll`, `split`, and `matches`

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 program or exit with friendly message

EXCEPTION HANDLING

```
import java.util.Scanner;
public class Test {
   public static void main(String[] args) {
      Scanner keyboard = new Scanner(System.in);
      int x, y;
      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));
```

```
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 Test.main(Test.java:9)
```

EXCEPTION HANDLING

```
public class Test {
  public static void main(String[] args) {
    int x, y;
    int[] a = {10, 20, 30, 40};
    x = 10;
    y = x / 2;
    a[y] = a[y] * 2;
  }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
  at Test.main(Test.java:7)
```

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

```
public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     System.out.println("Enter the number of students: “);

       int studentCount = input.nextInt();
       Student[] studentList = new Student[studentCount];
       for(int i=0; i<studentCount; i++) {
          String name; int id; double gpa;
          System.out.println("Enter student information(name id gpa): ");
          name = input.next() + input.next();
         try{
           id = input.nextInt();
           gpa = input.nextDouble();
           studentList[i] = new Student(name, id, gpa);
          }
         catch(Exception e){
          System.out.println(“Input Mismatch Runtime Error.”);
          input.next();
          }
       }
}
```

InputMismatchException Runtime Error

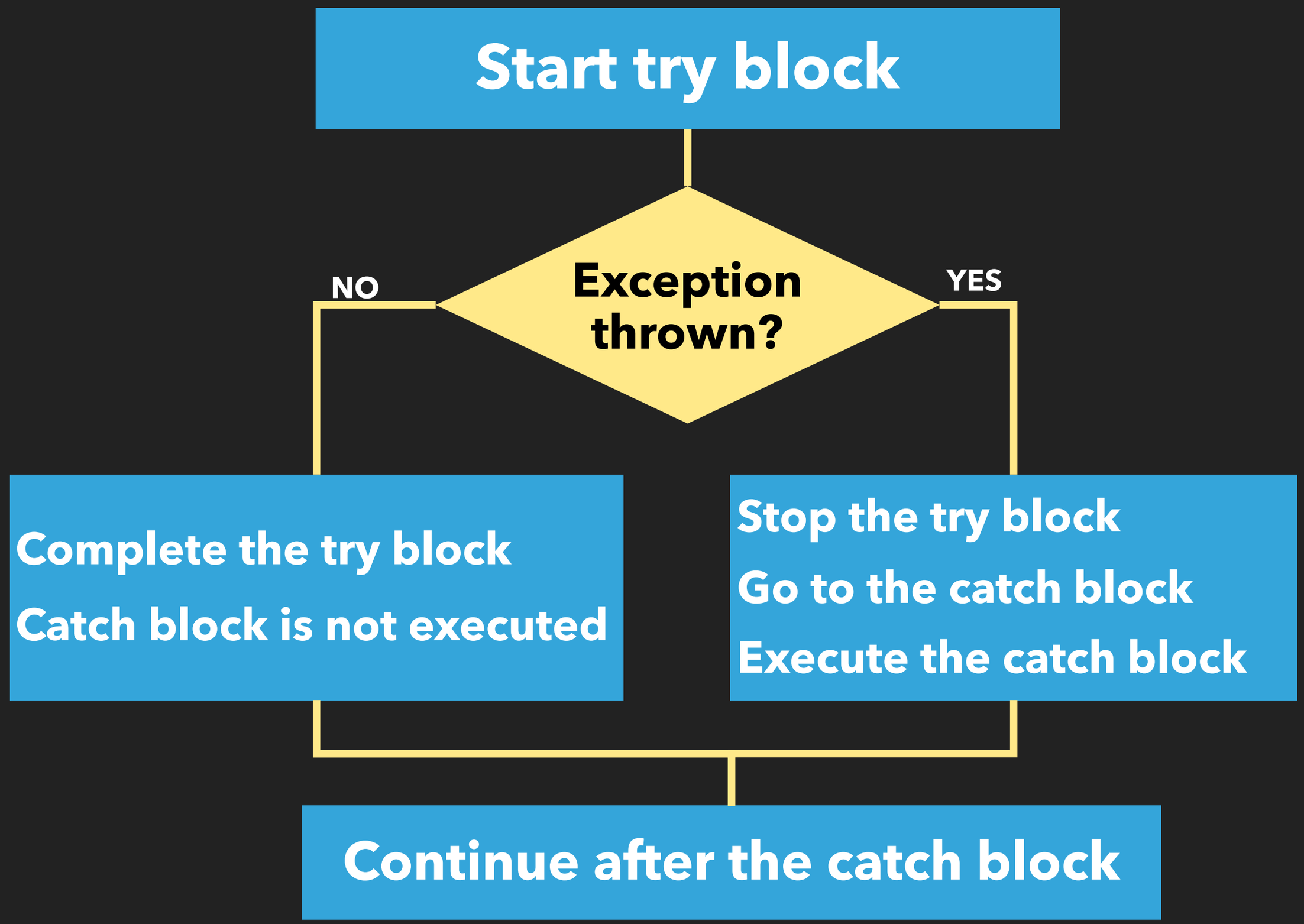
EXCEPTION HANDLING

```
public static void main(String[] args) {
     Scanner input = new Scanner(System.in);
     System.out.println("Enter the number of students: “);

       int studentCount = input.nextInt();
       Student[] studentList = new Student[studentCount];
       for(int i=0; i<studentCount; i++) {
          String name; int id; double gpa;
          System.out.println("Enter student information(name id gpa): ");
          name = input.next() + input.next();
          try{                                   Where the exception may happen
           id = input.nextInt();
           gpa = input.nextDouble();
           studentList[i] = new Student(name, id, gpa);
          }
          catch(Exception e){
           System.out.println(“Input Mismatch Runtime Error.”);
           input.next();
          }                                      Where the exception is handled when it happens
       }
}
```

InputMismatchException Runtime Error

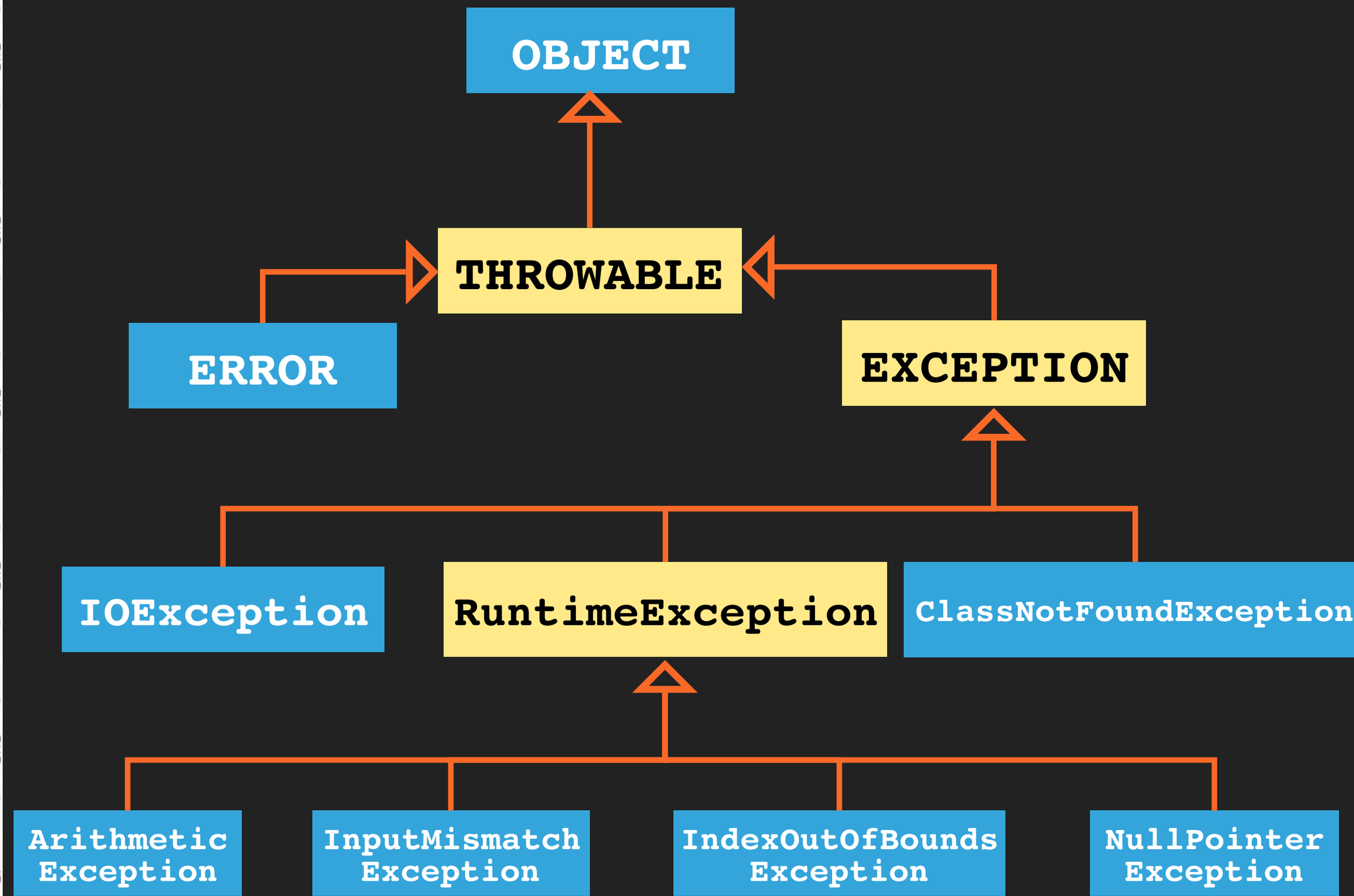
EXCEPTION HANDLING



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`** - Hierarchy of classes in Java API

EXCEPTION HANDLING



Throwing Exceptions

```
Java.lang.Throwable
-message: String
+getMessage(): String
+toString(): String
+printStackTrace(): void
+getStackTrace():StackTraceElement[]
```

```java
import java.util.InputMismatchException;
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter the number of students: ");
   int studentCount = input.nextInt();
   Student[] studentList = new Student[studentCount];
   for(int i=0; i<studentCount; i++) {
      String name; int id; double gpa;
      System.out.println(
          "Enter student information(name id gpa): ");
      name = input.next() + input.next();
      try{
         id = input.nextInt();
         gpa = input.nextDouble();
         studentList[i] = new Student(name, id, gpa);
         System.out.println("Student " + (i+1) + ":" +
                                 name + ", " + id + ", " + gpa);
      }
      catch(InputMismatchException e){
         System.out.println("Input Mismatch Runtime Error.");
         input.next();
       }
   }
}
```

Throwing Exceptions

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

Throwing Exceptions

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

- Anonymous **`Exception`** object passed to the catch block
- **`e.getMessage()`** returns the string passed to **`Exception`** class constructor

Throwing Exceptions

```
public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter the number of students: “);
      int studentCount = input.nextInt();
      Student[] studentList = new Student[studentCount];
      for(int i=0; i<studentCount; i++) {
         String name; int id; double gpa;
         System.out.println("Enter student information(name id gpa): ");
         name = input.next() + input.next();
         try{
            id = input.nextInt();
            gpa = input.nextDouble();
            if(gpa < 0.0 || gpa > 4.0)
               throw new Exception(“Invalid GPA: ” + gpa);
            studentList[i] = new Student(name, id, gpa);
         }
         catch(Exception e){
          System.out.println(“Runtime error: “ + e.getMessage());
          input.next();
         }
      }
}
```

Throwing Exceptions

```
try {
      Statement(s)
      throw statement(s) or call to
       Methods that throw exceptions;
      Statement(s)
}
catch (ExcpetionClass e) {
      Statement(s) to execute when an
       ExceptionClass object e is thrown
}
```

EXCEPTION HANDLING

What is the output of the following code?

```
int waitTime = 40;
try{
    System.out.println("Try block entered.");
    if (waitTime > 30)
      throw new  Exception("Over 30.”);// anonymous
    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 your own exception classes
- Programmer-created exceptions must be derived from Java exception classes
- Derived classes must have two constructors at least (no-arg and one parameter of type `String`)

Creating New Exception Classes

```
public class InvalidGPAException extends Exception {

 public InvalidGPAException() {
   super(“Invalid GPA Exception");
 }
 public InvalidGPAException(String message) {
   super(message);
 }

}
```

Creating New Exception Classes

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter the number of students: “);
   int studentCount = input.nextInt();
   Student[] studentList = new Student[studentCount];
   for(int i=0; i<studentCount; i++) {
      String name; int id; double gpa;
      System.out.println("Enter student information(name id gpa): ");
      name = input.next() + input.next();
      try{
         id = input.nextInt();
         gpa = input.nextDouble();
         if(gpa < 0.0 || gpa > 4.0)
               throw new InvalidGPAException(“Invalid GPA: ” + gpa);
         studentList[i] = new Student(name, id, gpa);
      }
      catch(InvalidGPAException e){
          System.out.println(“Runtime error: “ + e.getMessage());
          input.next();
      }
   }
}
```

Practice

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

```
TestException e = new TestException();
System.out.println(e.getMessage());
e.testMethod();
```

Multiple Catch blocks

- Each Catch block associated with a specific type of exception (type of the parameter **e**)
- A try block may throw exceptions of different types
- Multiple catch blocks - one for each type of exception thrown

EXCEPTION HANDLING

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter the number of students: “);
   int studentCount = input.nextInt();
   Student[] studentList = new Student[studentCount];
   for(int i=0; i<studentCount; i++) {
      String name; int id; double gpa;
      System.out.println("Enter student information(name id gpa): ");
      name = input.next();
      try{
        id = input.nextInt();
        gpa = input.nextDouble();
        if(gpa < 0.0 || gpa > 4.0)
           throw new InvalidGPAException(“Invalid GPA: ” + gpa);
        studentList[i] = new Student(name, id, gpa);
      }
      catch(InputMismatchException e){
          System.out.println(“Input Mismatch Runtime Error.”);
          input.next();
      }
      catch(InvalidGPAException e){
         System.out.println(“Runtime error: “ + e.getMessage());
         input.next();
      }
}
```

EXCEPTION HANDLING

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter the number of students: “);
   int studentCount = input.nextInt();
   Student[] studentList = new Student[studentCount];
   for(int i=0; i<studentCount; i++) {
      String name; int id; double gpa;
      System.out.println("Enter student information(name id gpa): ");
      name = input.next();
      try{
        id = input.nextInt();
        gpa = input.nextDouble();
        if(gpa < 0.0 || gpa > 4.0)
           throw new InvalidGPAException(“Invalid GPA: ” + gpa);
        studentList[i] = new Student(name, id, gpa);
      }
      catch(InputMismatchException e){
          System.out.println(“Input Mismatch Runtime Error.”);
          input.next();
      }
      catch(InvalidGPAException e){
         System.out.println(“Runtime error: “ + e.getMessage());
         input.next();
      }
}
```

Multiple Catch blocks

- Order of catch blocks matters
- From specific to general
- Follow the hierarchy of inheritance (from sub classes to super classes)

Multiple Catch blocks

```
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

- `nextInt()` throws an exception and does not handle it - (`declare rule`)
- The caller of `nextInt()` decides to handle the exception or not
- You can also create methods that throw exceptions and handle them (`catch rule`)

Catch-Declare Rule

- Declare-rule : use the clause '**throws**'

```
public void safeDivide(int a, int b)
throws DivisionByZeroException
```

- A method may declare more than one exception to be thrown (list separated by **,**)

Catch-Declare Rule

- Declare rule

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

Catch-Declare Rule

- Catch rule

```
double safeDivide(int a, int b)
{
   try{
    if (b == 0)
     throw new DivisionByZeroException();
    else
     return (a/b);
   }
   catch(DivisionByZeroException e){
     return 0;
   }
}
```

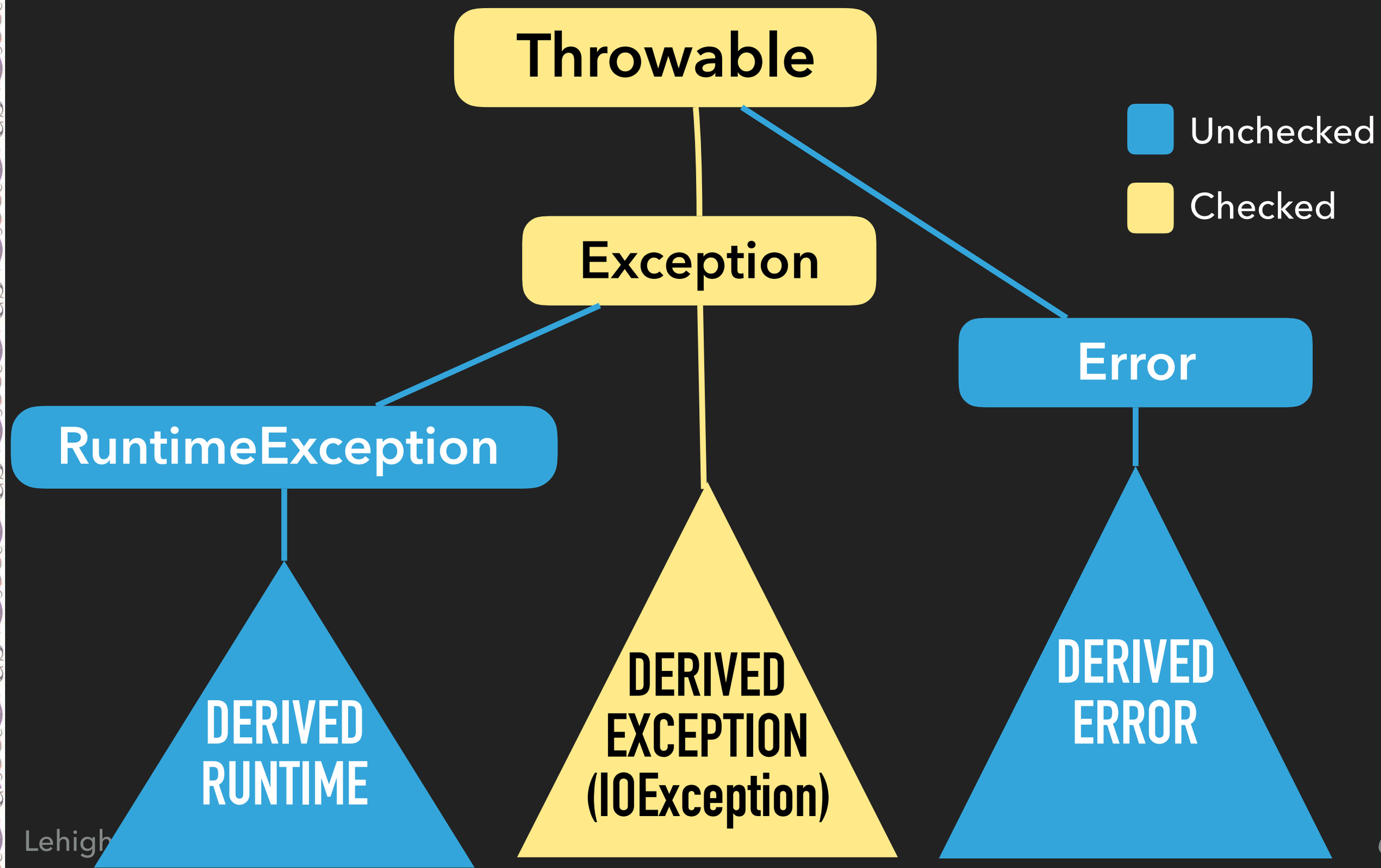
Catch-Declare Rule

- Catch rule - A method has try/catch for an exception - throws clause not required
- Declare rule - method does not catch the exception - throws clause required
- Mix of **catch** and **declare** exceptions

Catch-Declare Rule

- Rule enforced for checked exceptions
  - Checked Exception - Exception for which Java enforces the rule catch or declare
  - Unchecked Exception - Exception not checked by Java for catch or declare rule

Catch-Declare Rule



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

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

EXCEPTION HANDLING

```
public class FinallyDemo {
 public static void main(String[] args){
   try { exerciseMethod(0);}
   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 NegativeNumberException();
    else
      System.out.println("No Exception.");
    System.out.println("Still in exerciseMethod.");
  }
  catch(NegativeNumberException 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`**
- Declare exception - **`throws`**
- Deriving new exception classes
- Catch or declare rule - **checked**/**unchecked** exceptions

FILE IO

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

Class File

- Wrapper Class for files
  - Allow access to file properties
  - Example: file exists? Can read file? Can write to file? etc.

Class File

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

Class File

```
import java.io.File;
import java.util.Scanner;
public class Test {
   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 text file.");
         System.out.println("Size: " + file.length() + " bytes");
      }
      if(file.isDirectory()) {
         System.out.println(filename + " is a directory.");
         System.out.println("Path: " + file.getPath());
      }
   }
}
```

Reading/Writing Files

- **`Open`** the file for reading or writing
- **`Read`** from / **`Write`** to the file
- **`Close`** the file

Open Files for reading

- Create a **`Scanner`** object linked to a class File object - Class File object is linked to the file (to read from)

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

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

Open Files for Writing

- Create a **`PrintWriter`** object linked to a class File object - Class File object is linked to the file (write to)
- **`PrintWriter(File)`** constructor throws a checked "**`FileNotFoundException`**"

```
File file = new File("output.txt");
PrintWriter fileWrite =new PrintWriter(file);
```

Reading from File

- Use **`Scanner`** methods: **`nextInt()`**, **`nextDouble()`**, **`next()`**, **`nextLine()`**

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

int value = fileScanner.nextInt();//reading

String name = fileScanner.nextLine();
```

Writing to File

- Use **`PrintWriter`** methods:

`print(), println(), printf()`

```
File file = new File("output.txt");
PrintWriter fileWrite =new PrintWriter(file);

fileWrite.println("value  = " + value);
```

Close Files

- `close()` method from class `Scanner`

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

- `close()` method from class `PrintWriter`

```
fileWrite.close();
```

FILE IO

```
import java.util.InputMismatchException;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        File file = new File("students.txt");
        int studentCount;
        Student[] studentList;
        System.out.println("Enter the number of students: ");
        try {
            studentCount = input.nextInt();
            studentList = new Student[studentCount];
            Scanner readFile = new Scanner(file);
            System.out.println("File opened successfully.");
            for(int i=0; i<studentCount; i++) {
                String fname, lname; int id; double gpa;
                fname = readFile.next();
                lname = readFile.next();
                id = readFile.nextInt();
                gpa = readFile.nextDouble();
                studentList[i] = new Student(fname + " " + lname, id, gpa);
                System.out.println("Student " + (i+1) + ": " + studentList[i].toString());
            }
            readFile.close();
        }
        catch(InputMismatchException e) {
            System.out.println("Input Format Error."); System.exit(0);
        }
        catch(FileNotFoundException e) {
            System.out.println("Cannot open file \"students.txt\"");
        }
    }
}
```

```java
import java.util.InputMismatchException;
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
public class Test {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        File file = new File("students.txt");
        int studentCount;
        Student[] studentList;
        System.out.println("Enter the number of students: ");
        try {
            studentCount = input.nextInt();
            studentList = new Student[studentCount];
            Scanner readFile = new Scanner(file);
            System.out.println("File opened successfully.");
            for(int i=0; i<studentCount; i++) {
                String fname, lname; int id; double gpa;
                fname = readFile.next();
                lname = readFile.next();
                id = readFile.nextInt();
                gpa = readFile.nextDouble();
                studentList[i] = new Student(fname + " " + lname, id, gpa);
                System.out.println("Student " + (i+1) + ": " + studentList[i].toString());
            }
            readFile.close();
            PrintWriter writeFile = new PrintWriter(file);
            for(int i=0; i<studentCount; i++) {
                writeFile.println(studentList[i].getName() + " " +
                                  studentList[i].getID() + " " +
                                 studentList[i].getGPA());
            }
            writeFile.close();// must close file after writing
        }
        catch(FileNotFoundException e) {
            System.out.println("Cannot open file."); System.exit(0);
```

FILE IO

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

FILE IO

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.util.InputMismatchException;
import java.io.IOException;
public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        File file = new File("students.txt");
        int studentCount; Student[] studentList;
        System.out.println("Enter the number of students: ");
        try {
            studentCount = input.nextInt();
            studentList = new Student[studentCount];
            Scanner readFile = new Scanner(file);
            System.out.println("File opened successfully.");
            int i=0;
            while (readFile.hasNext()) {
                String fname, lname; int id; double gpa;
                fname = readFile.next(); lname = readFile.next();
                id = readFile.nextInt(); gpa = readFile.nextDouble();
                studentList[i] = new Student(fname + " " + lname, id, gpa);
                System.out.println("Student " + (i+1) + ": " + studentList[i].toString());
                i++;
            }
            readFile.close();
            PrintWriter writeFile = new PrintWriter(file);
            for(i=0; i<studentCount; i++) {
                writeFile.println(studentList[i].getName() + " " + studentList[i].getID() + " " +
                                  studentList[i].getGPA());
            }
            writeFile.close();
        }
        catch(FileNotFoundException e) {
            System.out.println("Cannot open file."); System.exit(0);
```

Class Scanner methods

Method	Purpose	Exception thrown
Scanner()	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
- `Scanner` object - Reading from file - catch `FileNotFoundException` - read using methods `next()`, `nextInt()`, ...
- `PrintWriter` Object - Writing to file - catch `FileNotFoundException` - write using methods `print()`, `printf()`, `println()`