Exception and Error Handling, Input / Output, and File IO

Core Java: Day 4

Agenda

Day 04

01 Exception and Error Handling

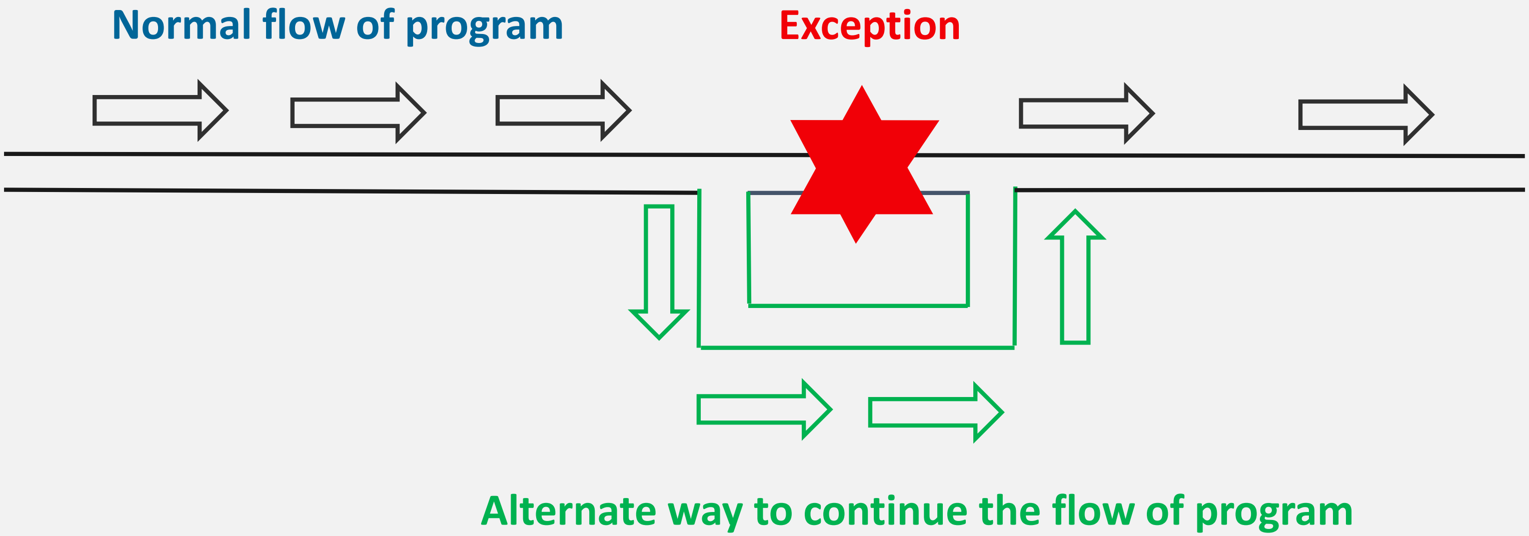
02 Input and Output Streams

03 File I / O (Input / Output)

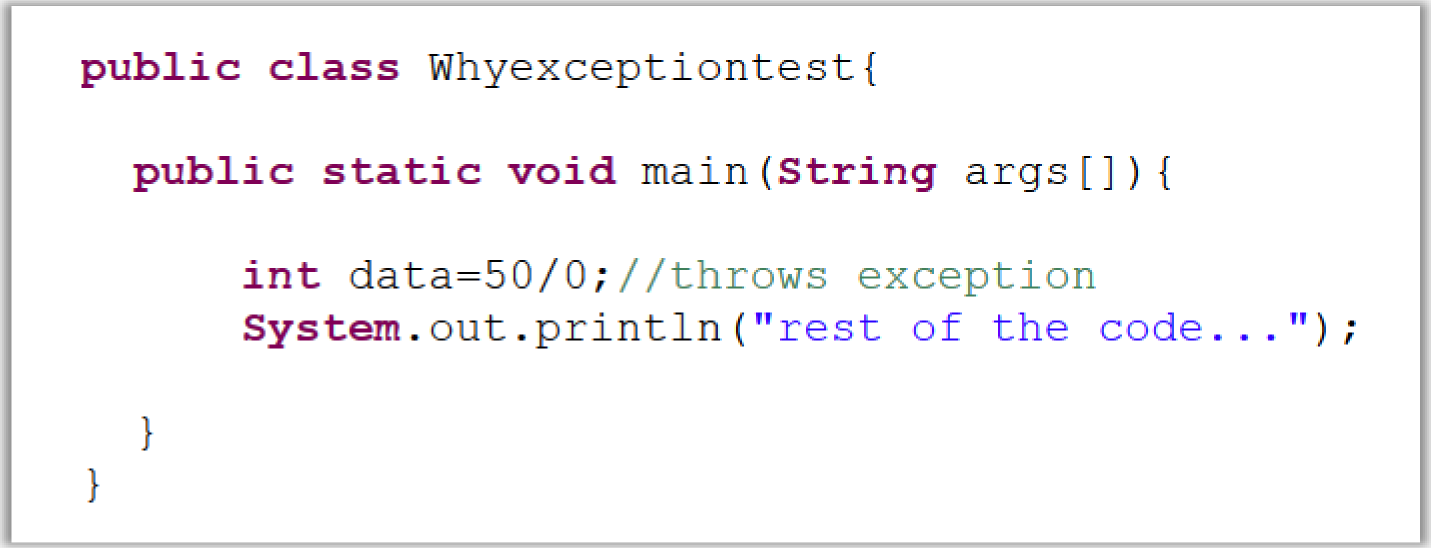
01 Exception and Error Handling

What is an **exception**?

An **exception** is an **event**, which **occurs** during the **execution of a program**, that **disrupts the normal flow** of the program's instructions.

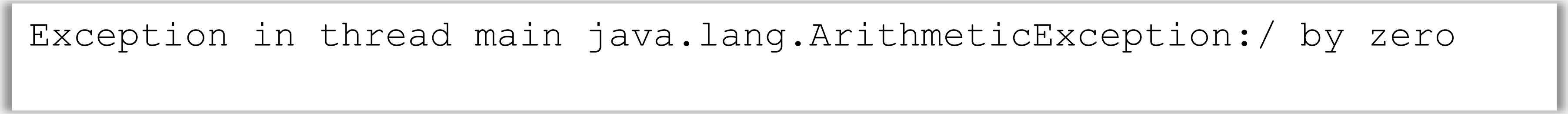


Why use exception?



```
public class Whyexceptiontest{

  public static void main(String args[]){

      int data=50/0;//throws exception
      System.out.println("rest of the code...");

  }
}
```

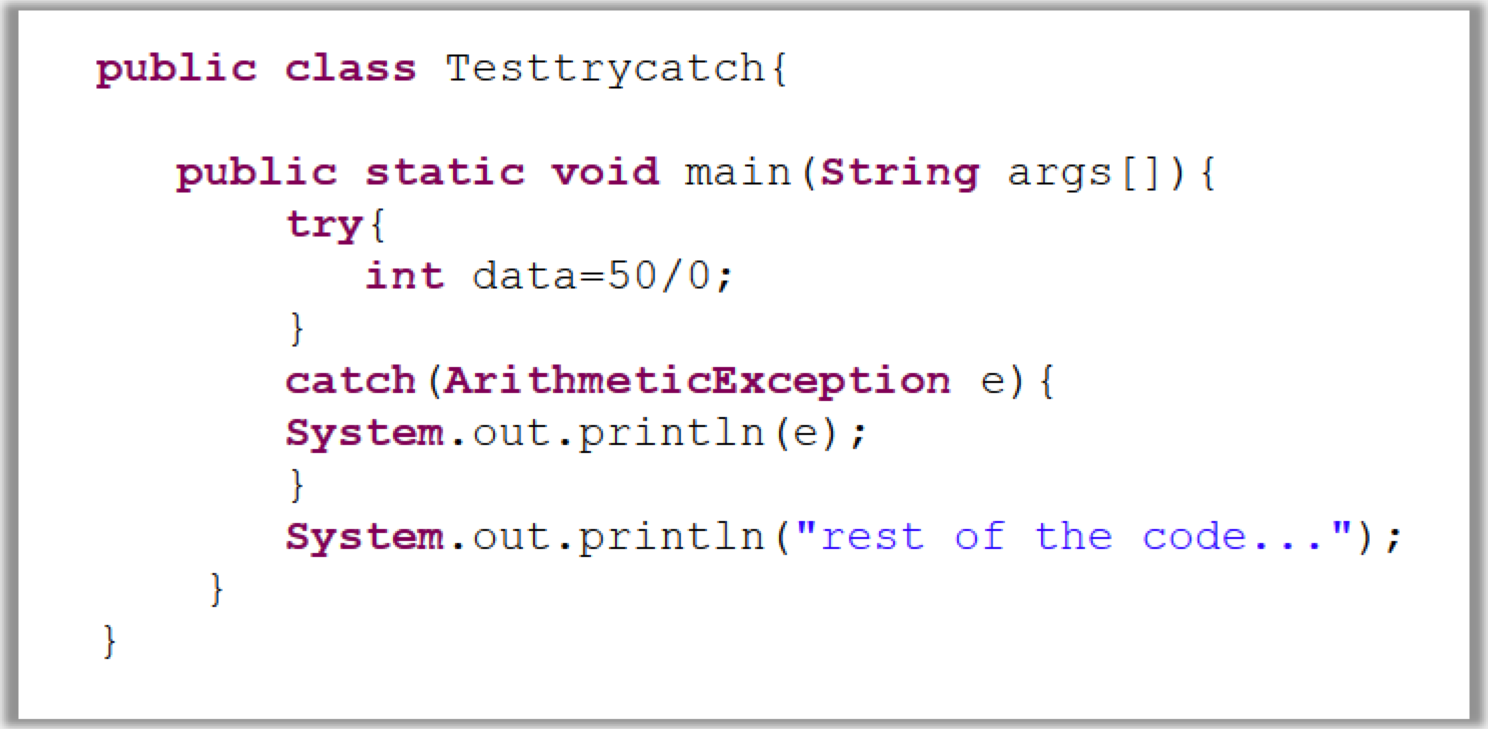
Output:



```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

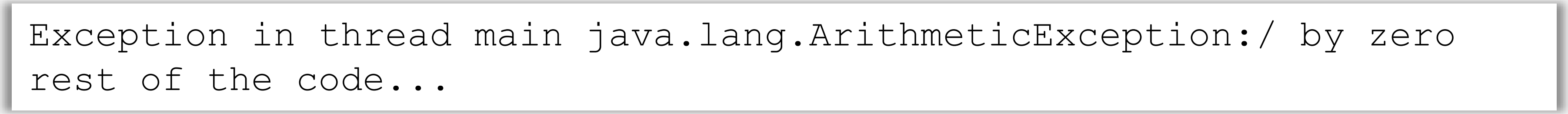
The rest of the code will not be executed. Imagine if there are be 100s of lines of code after exception. So all the code after exception will not be executed.

How to use exception?



```
public class Testtrycatch{

   public static void main(String args[]){
       try{
          int data=50/0;
       }
       catch(ArithmeticException e){
       System.out.println(e);
       }
       System.out.println("rest of the code...");
    }
}
```

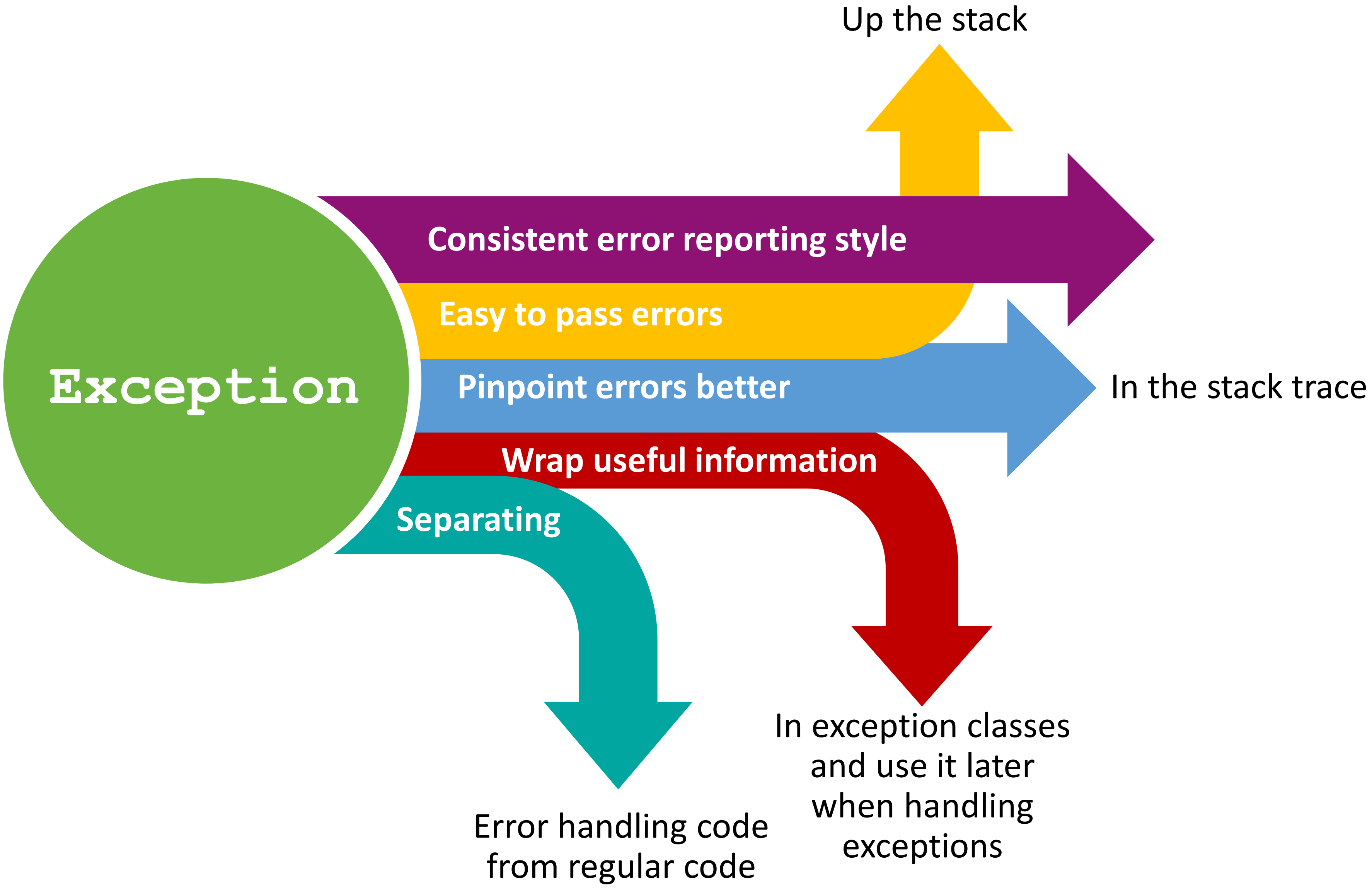
Output:



```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above output, rest of the code is executed even though there was an exception.

Advantages of exception



Up the stack

Exception

Consistent error reporting style

Easy to pass errors

Pinpoint errors better

In the stack trace

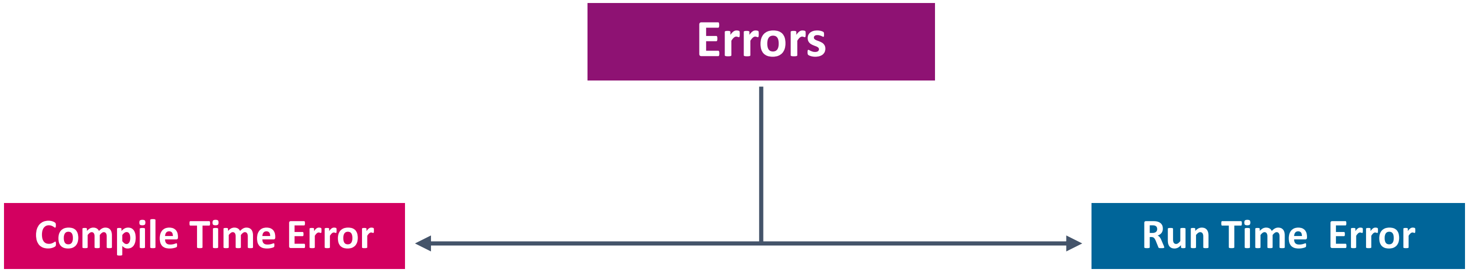
Wrap useful information

Separating

In exception classes and use it later when handling exceptions

Error handling code from regular code

Types of **errors**



Compile time error occurs while program is being **compiled.**

Examples:

- Syntax Errors
- Type checking Errors
- Semantic Errors

Run time error occurs during the **execution** of a program

Examples:

- Running out of memory
- Trying to open a file that isn’t there
- Division by Zero

Reasons and levels of exception occurrence

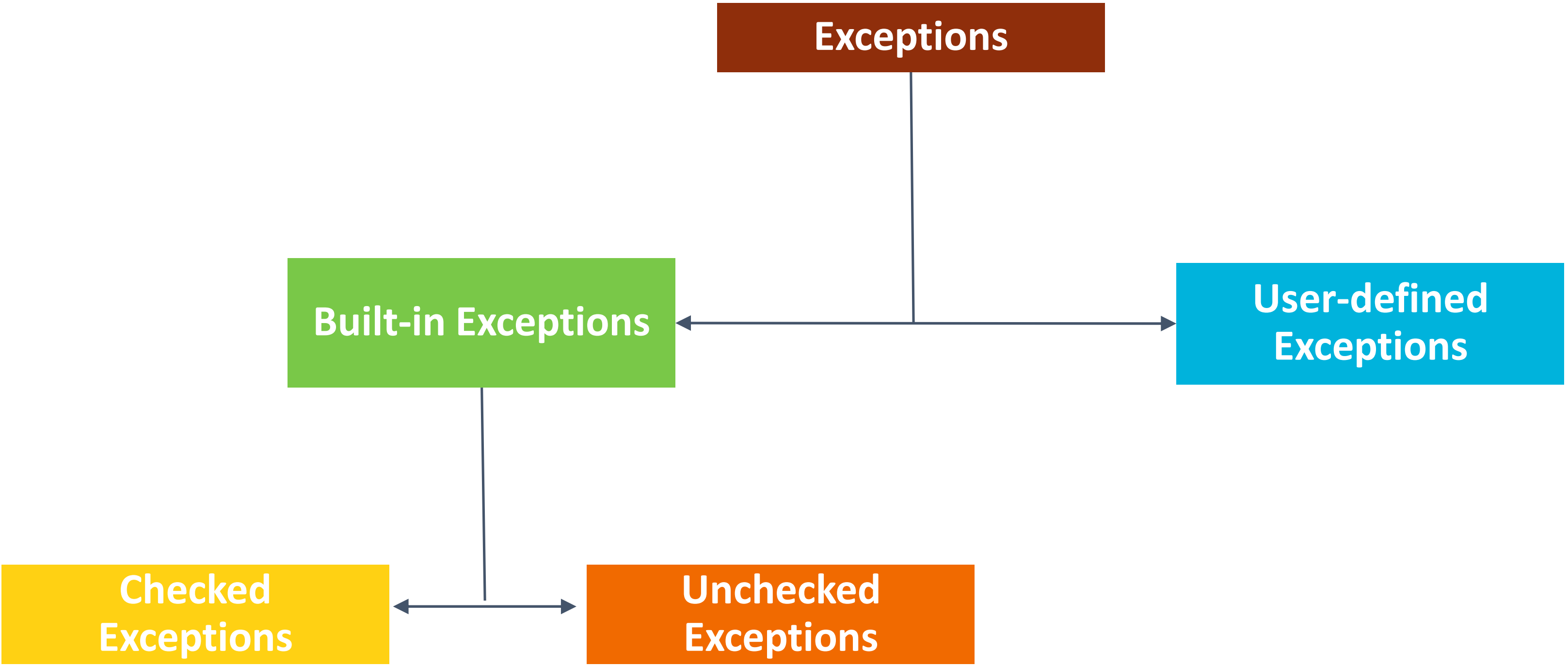
Reasons of Exception Occurrence:

- Running **out of memory**
- Resource allocation **errors**
- **Inability** to find files
- Problem in **network connectivity**

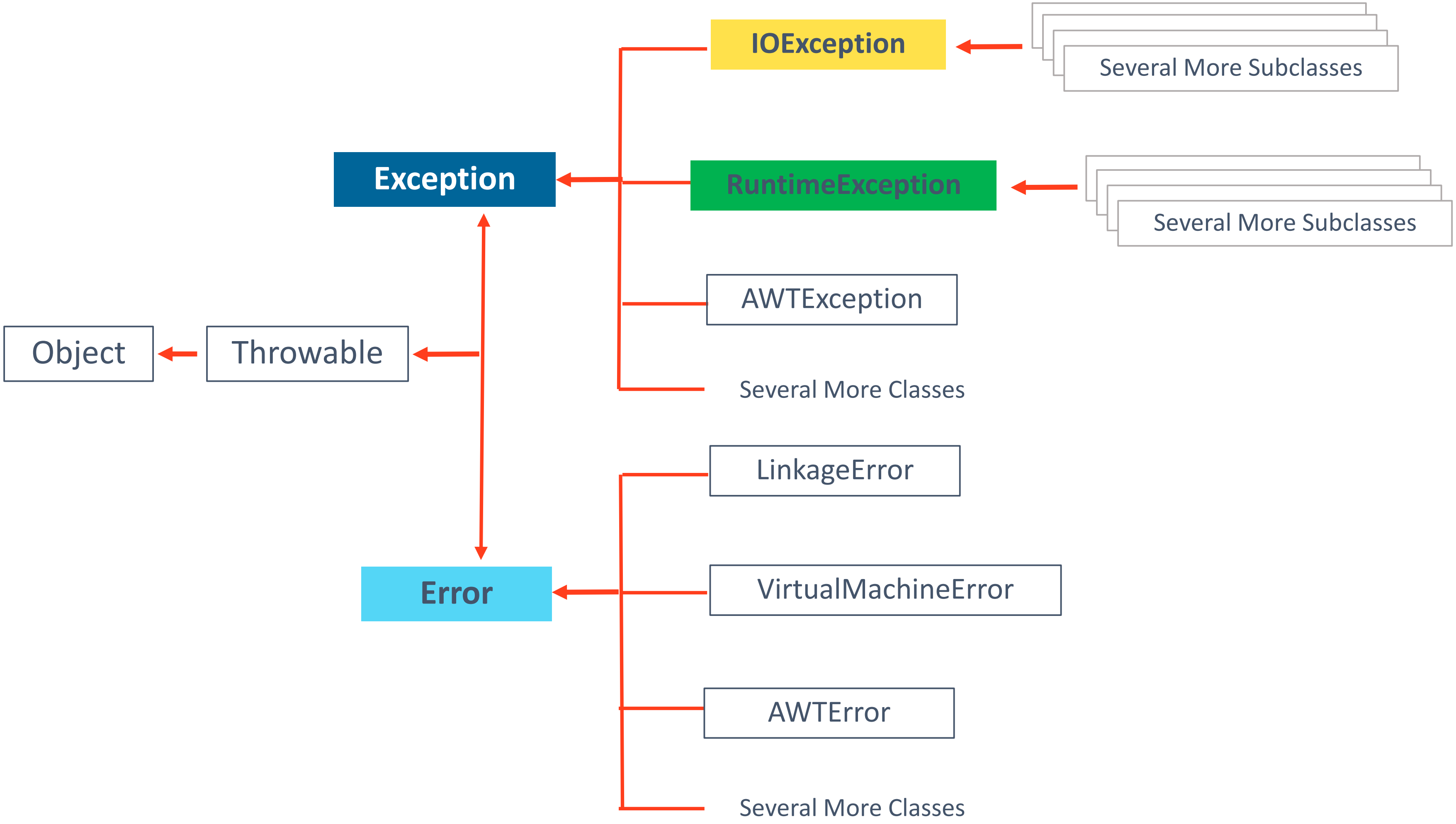
Levels of Exception Occurrence:

- Hardware/ Operating System Level:
  - Arithmetic exceptions; Divide by 0; under/overflow
  - Memory access violations; segment fault; stack over/underflow
- Language Level:
  - Type conversion; illegal values, improper casts
  - Bounds violations; illegal array indices
  - Bad references; null pointers
- Program Level:
  - User defined exceptions

Categories of **exceptions**



Hierarchy of Java exception



What are **checked exceptions**?

Checked Exceptions

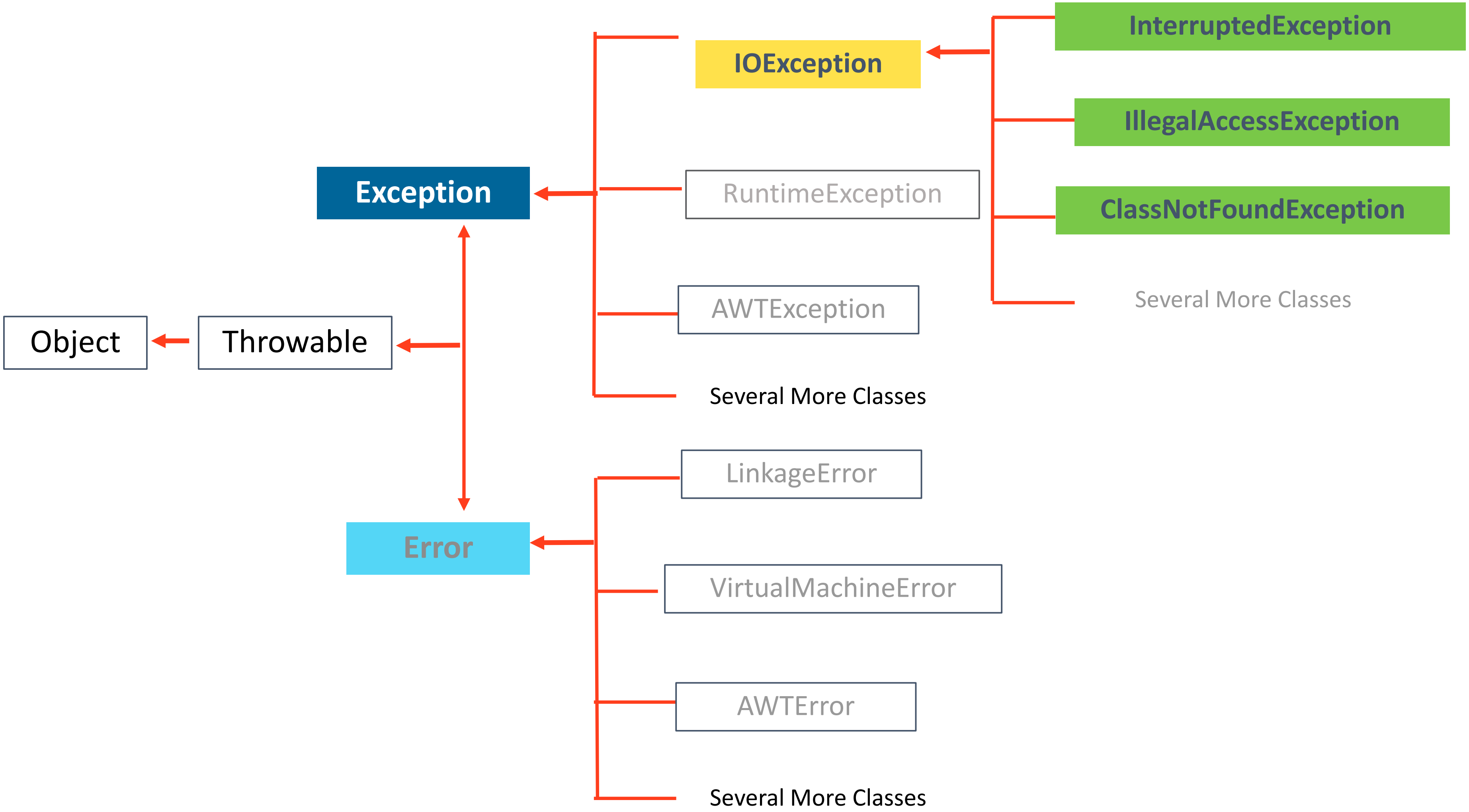
Exceptions that are **checked during the compile time**.

If some code within a method throws a checked exception, then the method **must handle the exception**, or must **specify the exception** using ***throws*** keyword.

Examples:

- `IOException`
- `FileNotFoundException`
- `ParseException`
- `ClassNotFoundException`
- `CloneNotSupportedException`
- `InstantiationException`
- `InterruptedException`
- `NoSuchMethodException`
- `NoSuchFieldException`

Occurrence of **checked exceptions**



What are **unchecked exceptions**?

Unchecked Exceptions

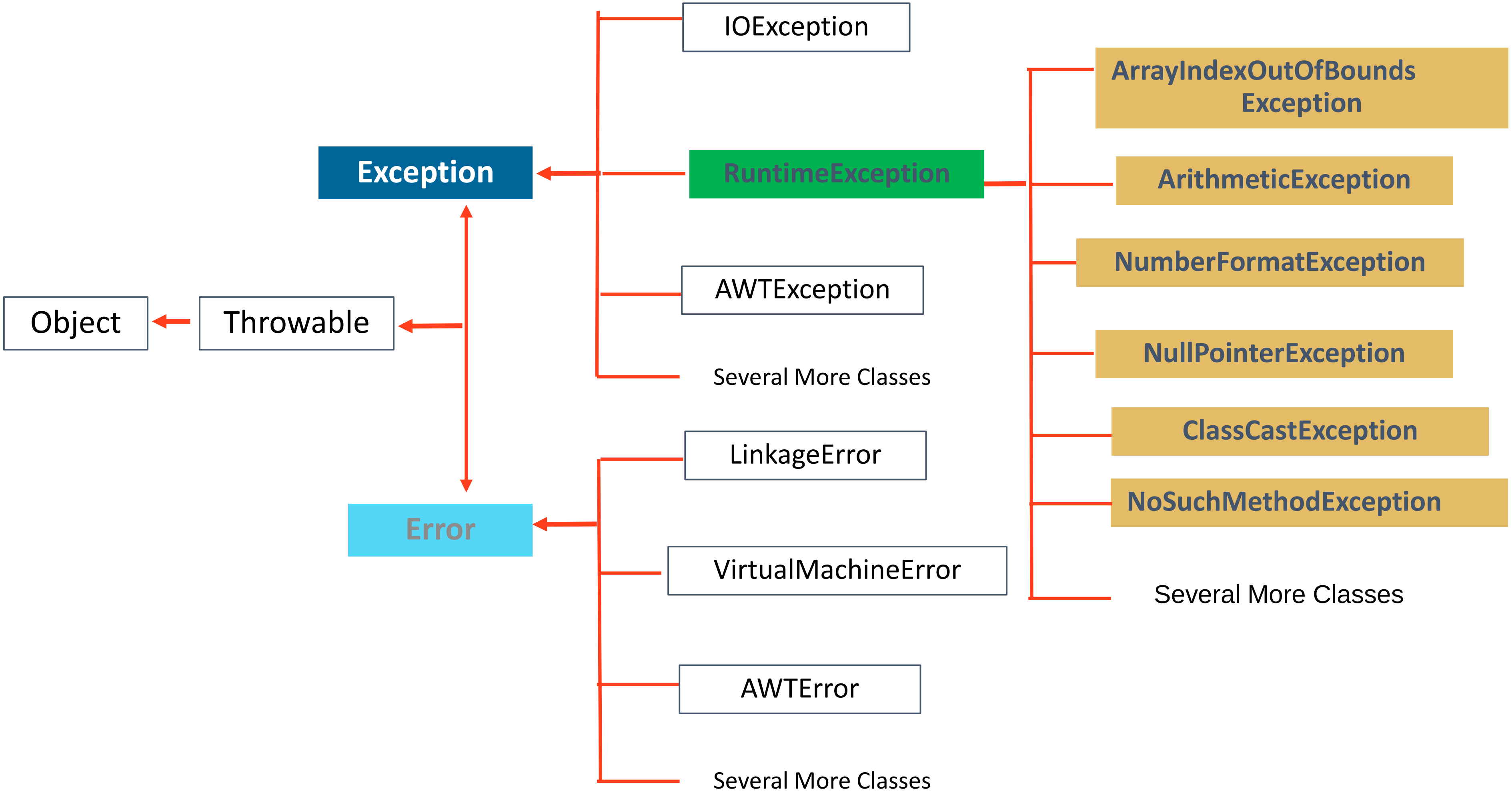
Exceptions that are **not checked during the compile time**.

In Java exceptions under ***Error*** and ***RuntimeException*** classes are **unchecked** exceptions, **everything** else under **throwable** is **checked**.

Examples:

- `ArrayIndexOutOfBoundException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalStateException`
- `NullPointerException`
- `NumberFormatException`
- `AssertionError`
- `ExceptionInitializeError`
- `StackOverflowError`
- `NoClassDefFoundError`

Occurrence of **unchecked exceptions**



Object

Throwable

Exception

Error

IOException

RuntimeException

AWTException

Several More Classes

LinkageError

VirtualMachineError

AWTError

Several More Classes

ArrayIndexOutOfBounds Exception

ArithmeticException

NumberFormatException

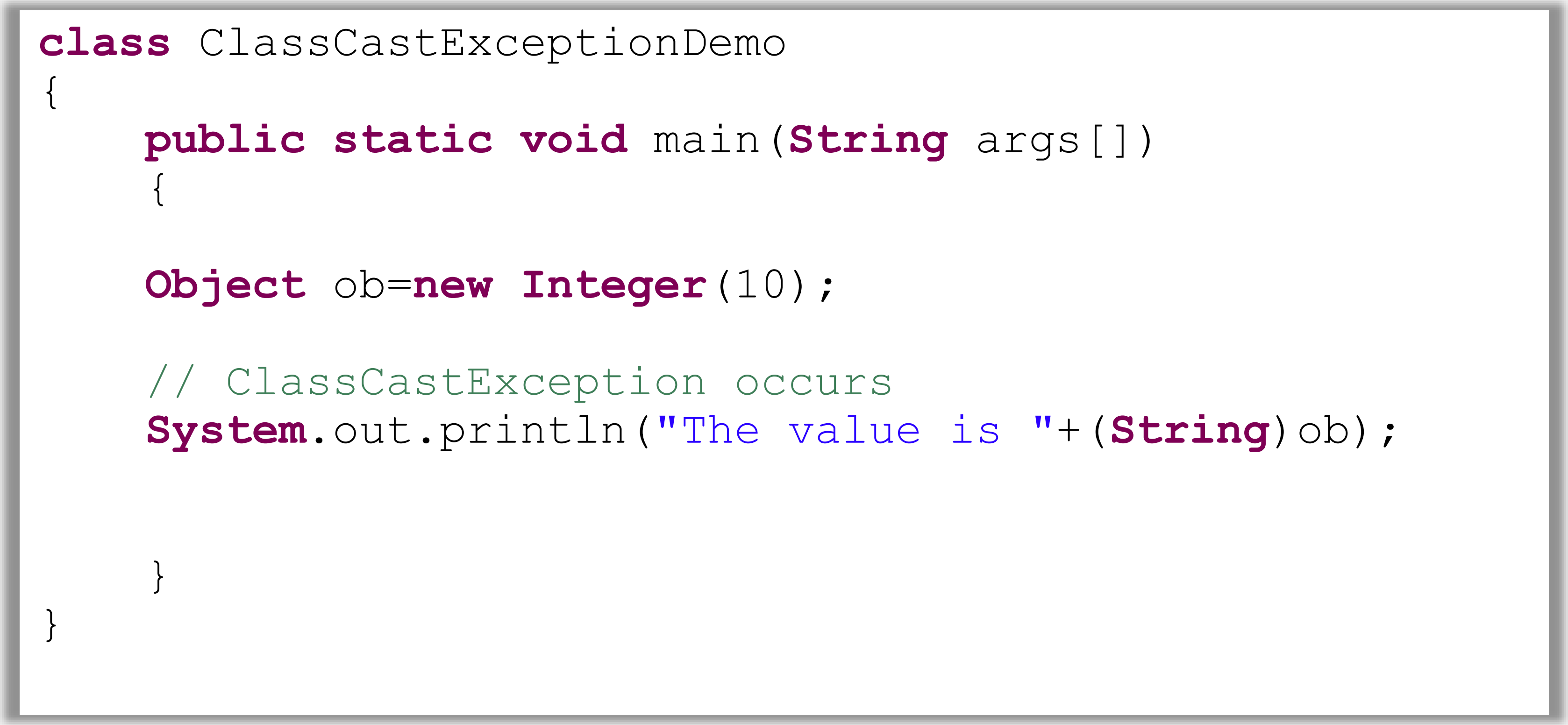
NullPointerException

ClassCastException

NoSuchMethodException

Several More Classes

ClassCastException: example



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

    Object ob=new Integer(10);

    // ClassCastException occurs
    System.out.println("The value is "+(String)ob);


    }
}
```

ArrayIndexOutOfBoundsException: example

```
public class Main {
   public static void main (String args[]) {
      int array[]={20,20,40};
      int num1=15,num2=10;
      int result=10;
      try {
         result = num1/num2;
         System.out.println("The result is" +result);
         for(int i =5;i >=0; i--) {
            System.out.println
            ("The value of array is" +array[i]);
         }
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array is out of Bounds"+e);
      }
      catch (ArithmeticException e) {
         System.out.println ("Can't divide by Zero"+e);
      }
   }
}
```

Keywords used to implement **exception handling**

`try`	The **try** block contains a block of program statements within which an exception might occur.
`catch`	A **catch** block **must be associated** with a **try** block. The corresponding catch block executes if an exception of a particular type occurs within the try block.
`throws`	If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword **appears at the end of a method's signature.**
`throw`	You can throw an exception, either a **newly instantiated** one or an **exception that you just caught**, by using the **throw** keyword.
`finally`	The **finally** keyword is used to **create a block of code that follows a try block**. A finally block of code **always executes**, **whether or not** an **exception** has **occurred**.

Implementing **exception handling**: using **try** and **catch**

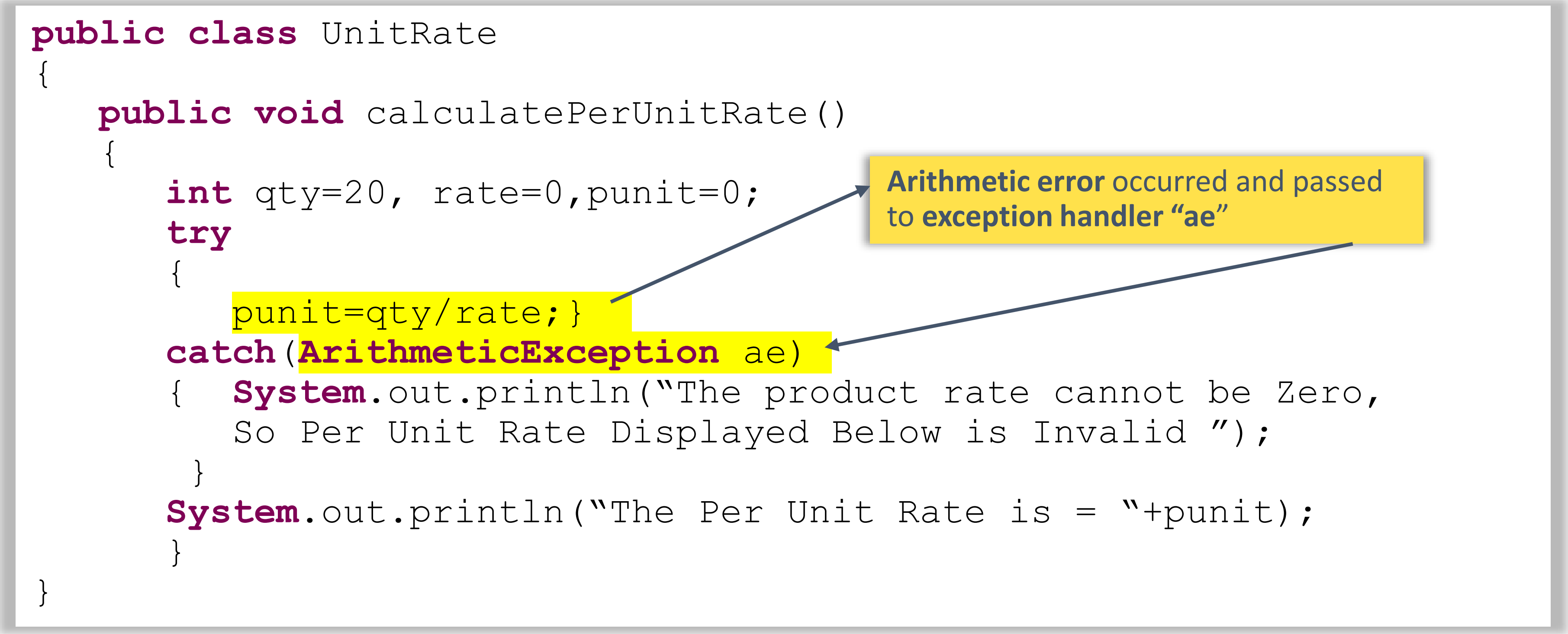
A method catches an exception using a combination of the `try` and `catch` keywords.

A `try/catch` block is placed around the code that might generate an exception.

```
try
{
   // Statements that cause an exception.
   }
catch(ExceptionName obj)
{
   // Error handling code.
}
```

The exception handler

An **arithmetic error** occurs on **dividing quantity by rate**.

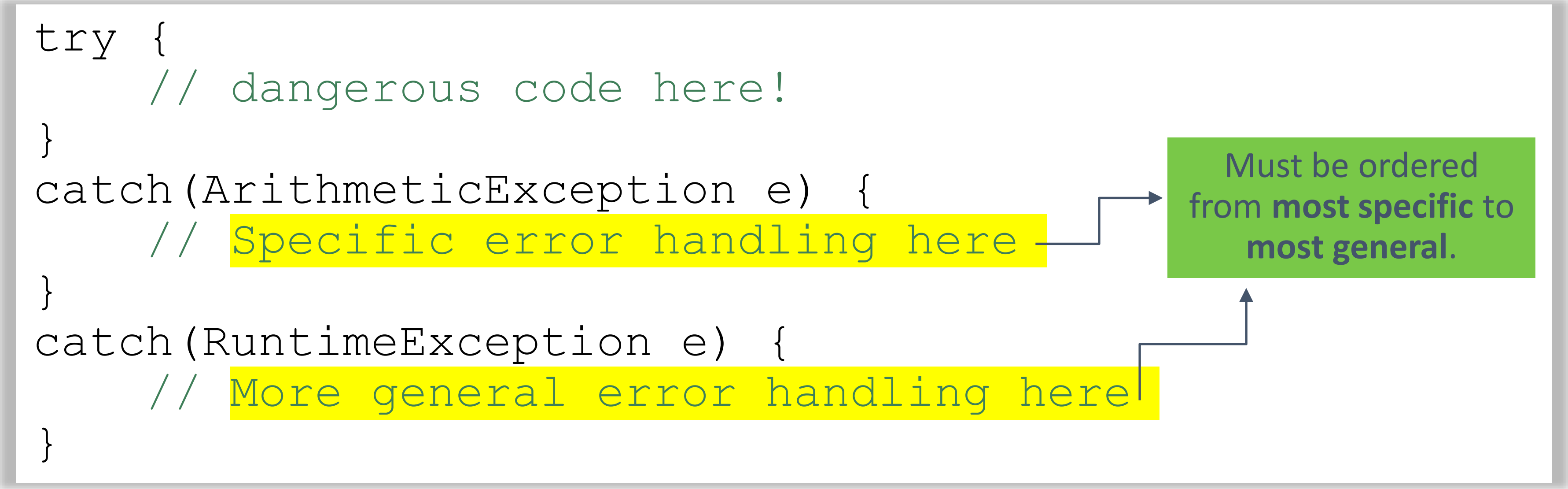


```
public class UnitRate
{
   public void calculatePerUnitRate()
    {
      int qty=20, rate=0,punit=0;
      try
      {
         punit=qty/rate;}
      catch(ArithmeticException ae)
      {  System.out.println("The product rate cannot be Zero,
         So Per Unit Rate Displayed Below is Invalid ");
       }
      System.out.println("The Per Unit Rate is = "+punit);
      }
}
```

Arithmetic error occurred and passed to **exception handler "ae"**

Using **multiple catch** statement

If you have to perform different tasks at the occurrence of different exceptions, use Java multiple catch statement.



```
try {
    // dangerous code here!
}
catch(ArithmeticException e) {
    // Specific error handling here
}
catch(RuntimeException e) {
    // More general error handling here
}
```

Must be ordered from **most specific** to **most general**.

Using **finally** clause

The `finally` block is used to process certain statements, no matter whether an exception is raised or not.

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
{
   //The finally block always executes.
}
```

A **`finally`** block appears at the end of the **`catch`** blocks.

Using **try...catch...finally** statement

The `try...catch...finally` statement provides a way to **handle** some or all of the **errors** that may **occur in a given block of code**, while still running code.

```
try {
     tryStatements
}
catch (ExceptionType1  identifier) {
      catchStatements
}
catch (ExceptionType2 identifier) {
       catchStatements
}
finally {
        finallyStatements
}
```

Using the throw statement

The Java `throw` keyword is used to **explicitly throw an exception**.

The following syntax shows how to declare the throw statement: `throw ThrowableObj`

```java
public double divide(int dividend, int divisor)
                                throws ArithmeticException {
   if(divisor == 0) {
      throw new ArithmeticException(“Divide by 0 error”);
   }
   return dividend / divisor;
}
```

Using the throws statement

The throws statement is **used** by a method **to specify** the **types of exceptions** the method throws.

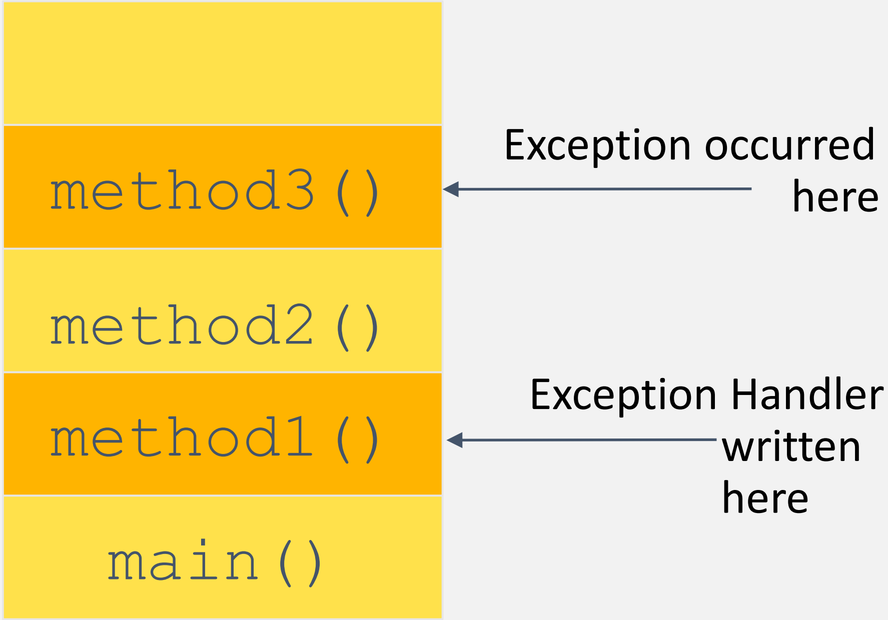
```
public void sample() throws IOException,SQLException{
     //Statements
     //if (somethingWrong)
     IOException e = new IOException();
     throw e;
     SQLException e = new SQLException();
     throw e;
     //More Statements
}
```

NOTE:

In case a method **throws more than one exception**, all of them should be listed in **throws** clause.

What is exception propagation?

When exception occurs at the top of the stack and no exception handler is provided then exception is propagated.



Exception propagation: example

```
1.   public  class  UnitRate {
2.               void  calculatePerUnitRate()  {
3.                        int  qty = 25,   rate = 0,punit=0;
4.                        punit = qty / rate;
5.               }
6.               void  callPerUnitRate() {
7.                        calculatePerUnitRate();
8.               }
9.               public static void main(String[] args) {
10.                        UnitRate ur = new UnitRate();
11.                        ur.callPerUnitRate();
12.               }
13.     }
```

Arithmetic Exception occurred here

Output by Default Exception Handler

```
java.lang.ArithmeticException:        / by zero
at UnitRate.calculatePerUnitRate(UnitRate.java:4)
at UnitRate.callPerUnitRate(UnitRate.java:7)
at UnitRate.main(UnitRate.java:11)
```

Activity

Write the **`printStackTrace`** for the `ExerciseException` class.

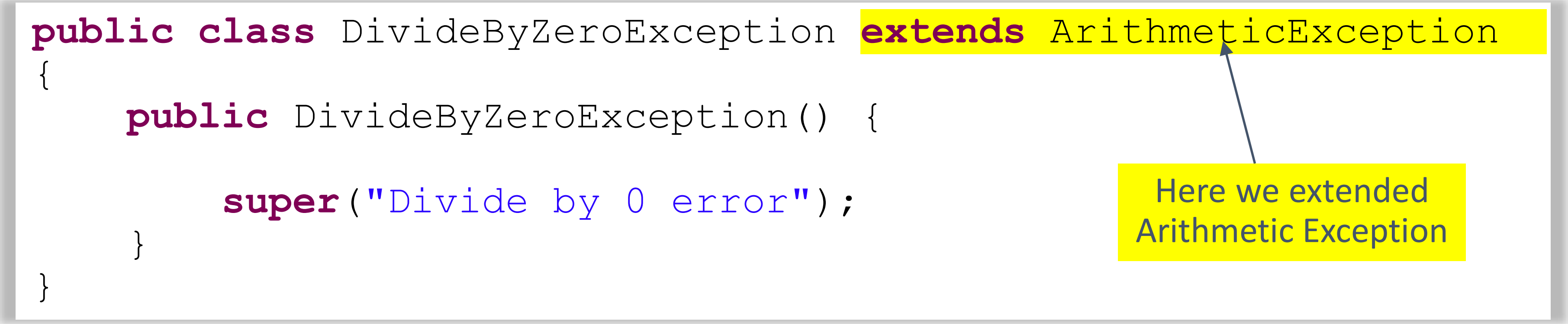
```
public class ExerciseException {
 public static void main(String[] args) {
    try {
      happy1();
     }
    catch(Exception e) {
      e.printStackTrace();
      }
     }
  public static void happy1() throws Exception {
    happy2();
  }
 public static void happy2() throws Exception {
    happy3();
  }
  public static void happy3() throws Exception {
    throw new Exception(“Be Happy Exception Prevents U!");
  }
}
```

User defined exceptions: creating your own exceptions

Defining your own exceptions **allows** you **to handle** specific **exceptions** that are **customized** for your application.

Procedure:

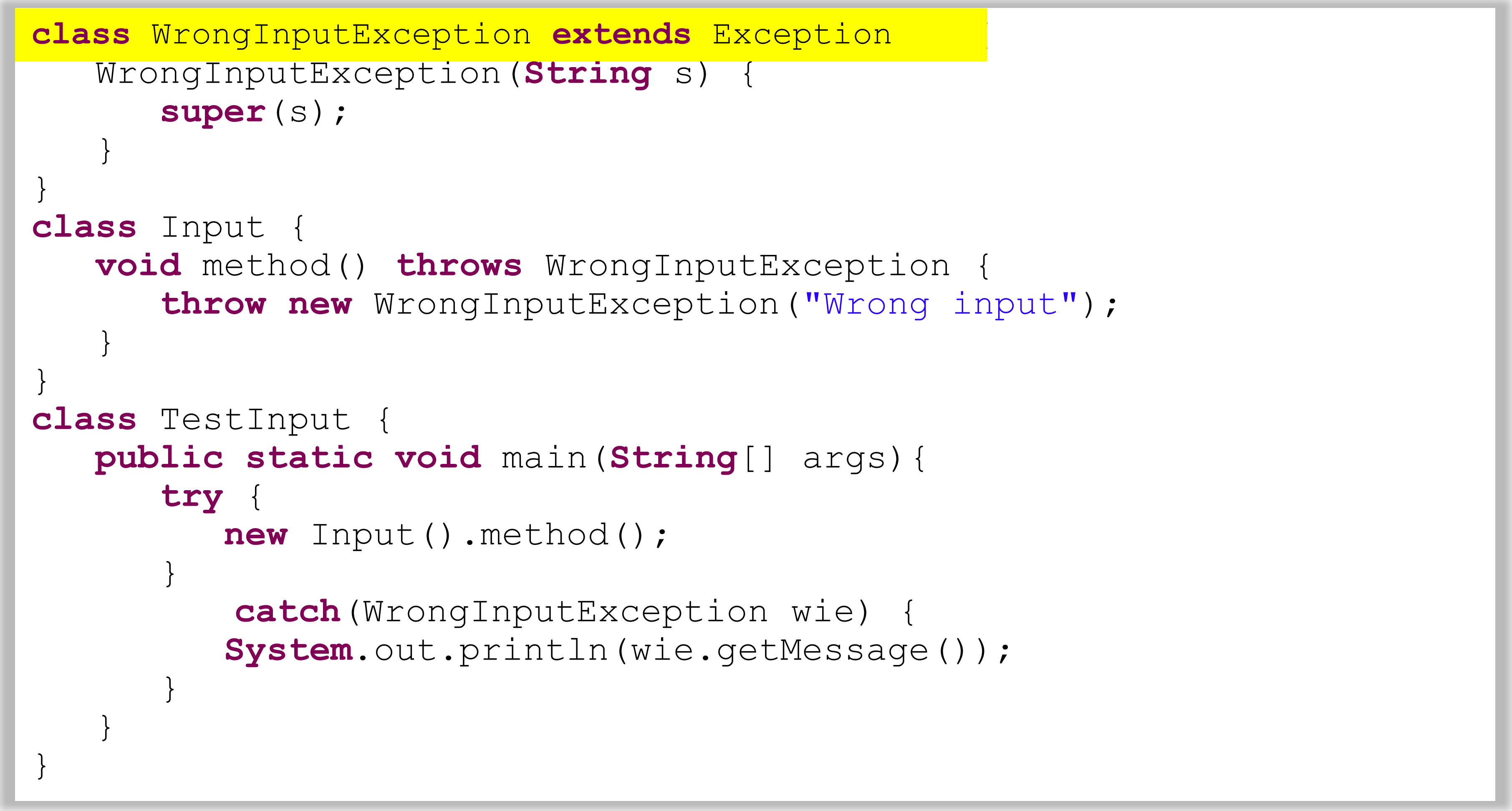
- Extend the exception class to create your own exception class.
- No methods are required.
- A constructor may be used if you want.
- You can override the toString() function, to display customized message.



```
public class DivideByZeroException extends ArithmeticException
{
    public DivideByZeroException() {

        super("Divide by 0 error");
    }
}
```

Here we extended Arithmetic Exception

User defined exceptions: **extending exception class**



```
class WrongInputException extends Exception
    WrongInputException(String s) {
        super(s);
    }
}
class Input {
    void method() throws WrongInputException {
        throw new WrongInputException("Wrong input");
    }
}
class TestInput {
    public static void main(String[] args){
        try {
            new Input().method();
        }
         catch(WrongInputException wie) {
            System.out.println(wie.getMessage());
        }
    }
}
```

Java SE7: catching multiple exception types

Java SE7 introduces an **enhancement** in the Exception Handling method. It **allows catching multiple exception types** in a single `catch` clause.

Consider this:

```
try {
    doSomeRiskyThings();
} catch (IOException ex) {
    //catch block 1
}
  catch (SQLException ex) {
   //catch block 2
}
  catch (CustomException ex) {
   //catch block 3
}
```

Cluttered, Duplicated and Redundant

In Java 7, we can catch all these three exception types in a single `catch` block:

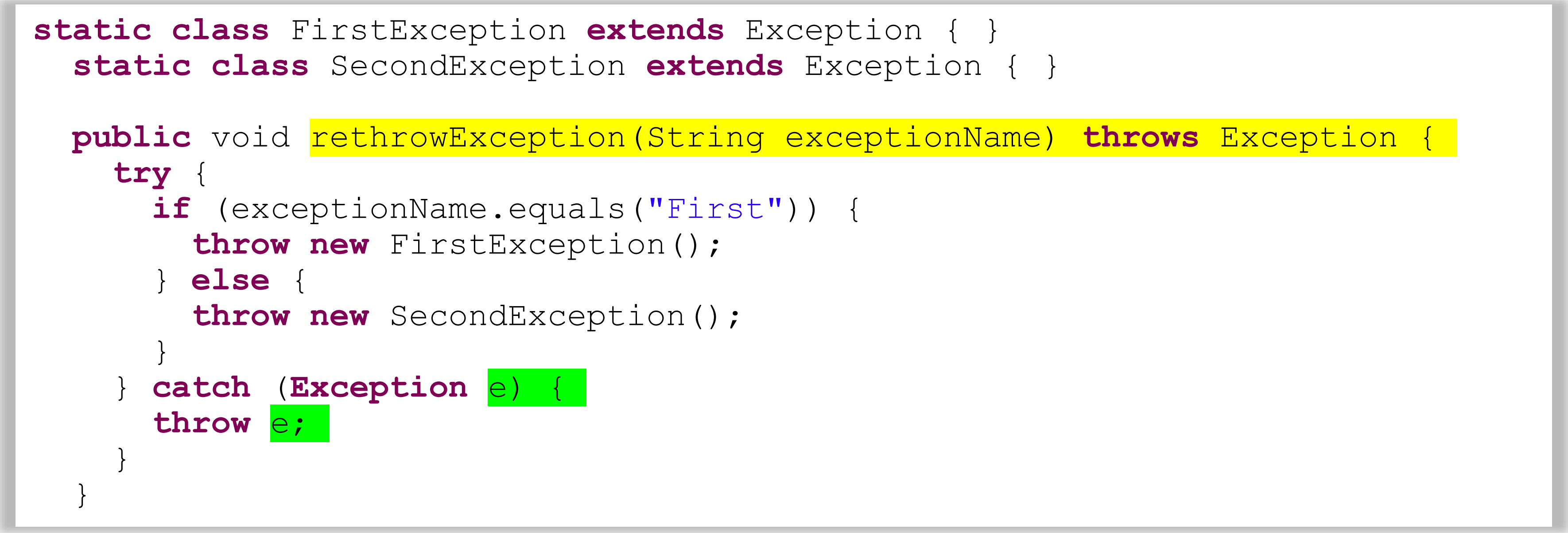
```
try {
     doSomeRiskyThings();
} catch (IOException | SQLException | CustomException ex) {
     //just one catch block needed!
}
```

All the 3 exception types caught in a single `catch` block.

Java SE7: **rethrowing exceptions** with improved type checking

The Java SE 7 compiler performs more **precise analysis** of rethrown exceptions. It enables **specifying more specific exception** types in the `throws` clause of a method declaration. In releases prior to Java 7, this was not possible.

Consider this:

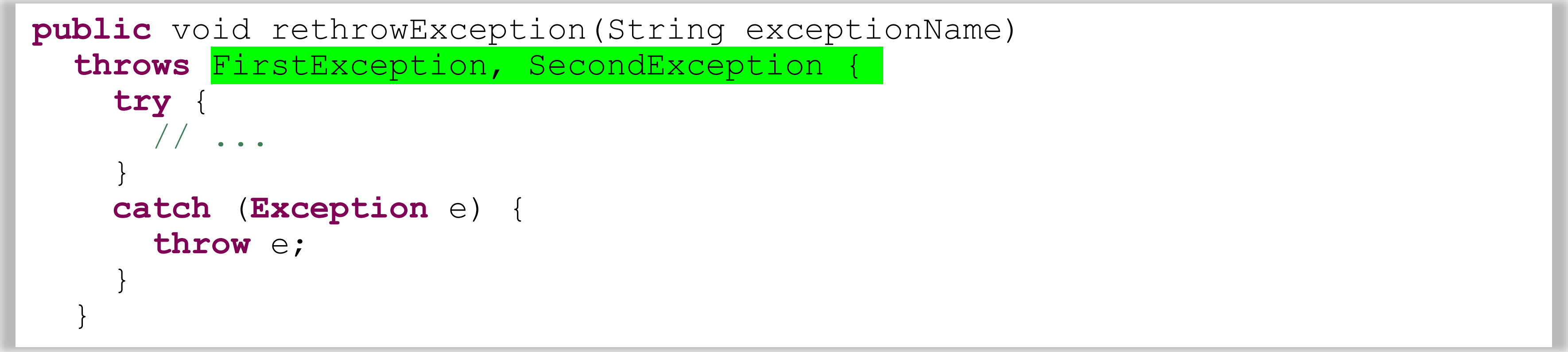


```
static class FirstException extends Exception { }
  static class SecondException extends Exception { }

  public void rethrowException(String exceptionName) throws Exception {
    try {
      if (exceptionName.equals("First")) {
        throw new FirstException();
      } else {
        throw new SecondException();
      }
    } catch (Exception e) {
      throw e;
    }
  }
```

Java SE7: **benefits of the compiler**

Java SE7 helps to:

- Specify one or more exception types in the `throws` clause in the `rethrowException()` declaration
- Determine if the exception thrown by `throw e` has come from `try` block
- Determine that the only exceptions thrown by `try` block can be `FirstException` and `SecondException`
- Determine that the exception parameter of `catch` clause `e` is an instance of either `FirstException` or `SecondException`



```
public void rethrowException(String exceptionName)
  throws FirstException, SecondException {
    try {
      // ...
    }
    catch (Exception e) {
      throw e;
    }
  }
```

Spot quiz

Brighter Blue

01

Which exception occurs when you try to create an object of an abstract class or interface?

A ClassNotFoundException

B IllegalAccessException

C InstantiationException

D NoSuchMethodException

Spot quiz

02

Which of the following keywords appear at the end of a method's signature?

A throw

B finally

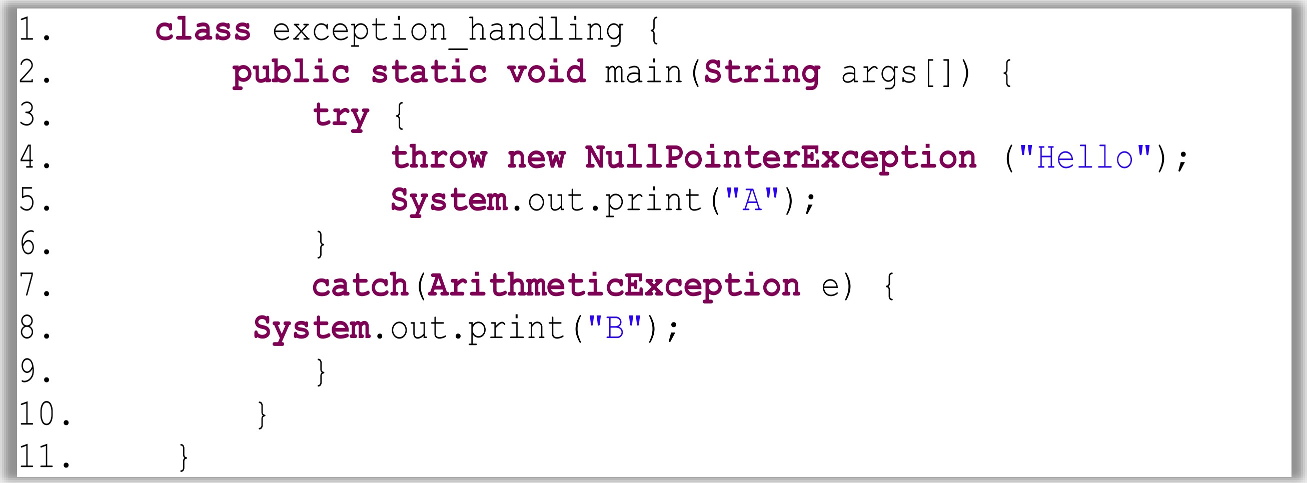
C throws

D catch

Spot quiz

03

What is the **output** of this program?



```
class exception_handling {
    public static void main(String args[]) {
        try {
            throw new NullPointerException ("Hello");
            System.out.print("A");
        }
        catch(ArithmeticException e) {
     System.out.print("B");
        }
     }
 }
```

A B

B A

C Compilation Error

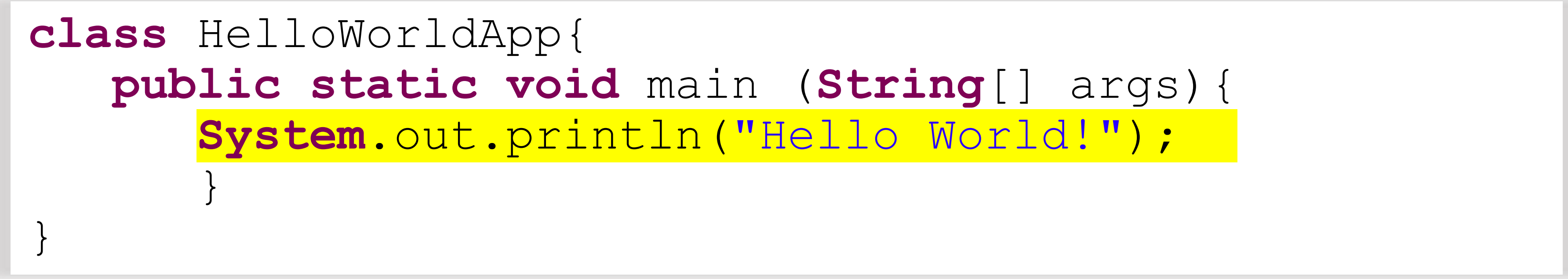
D Runtime Error

Questions?

02 Input and Output Streams

First encounter with I / O in Java

System.out refers to an output stream managed by the System class.



```
class HelloWorldApp{
   public static void main (String[] args){
      System.out.println("Hello World!");
      }
}
```

out is an instance of PrintStream class defined in java.io

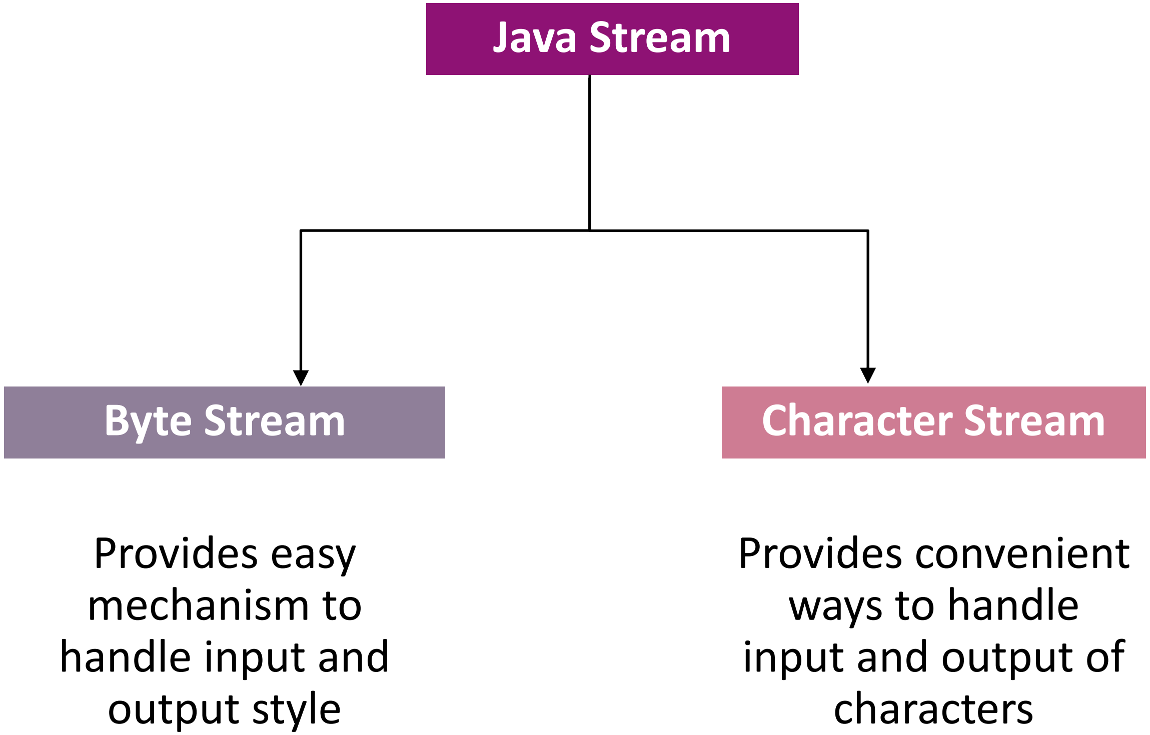
Overview of **I / O (Input – Output)**

The features of Java I / O are:

- Java programs use **classes** in java.io package to **read** and **write data**.
- The program can get **input** from a data source by **reading a sequence characters** from a **InputStream** attached to the **source**.
- The program can produce **output** by **writing a sequence of characters** to an **OutputStream** attached to a **destination**.

What is a **stream**?

- A stream is a sequence or a flowing set of data.
- Streams support different kinds of data – simple bytes and primitive data types.
- Streams may either pass on data, or manipulate and transform the data usefully.
- Java encapsulates streams under java.io package.


Java Stream

Byte Stream

Character Stream

Provides easy mechanism to handle input and output style

Provides convenient ways to handle input and output of characters

Streams: Examples

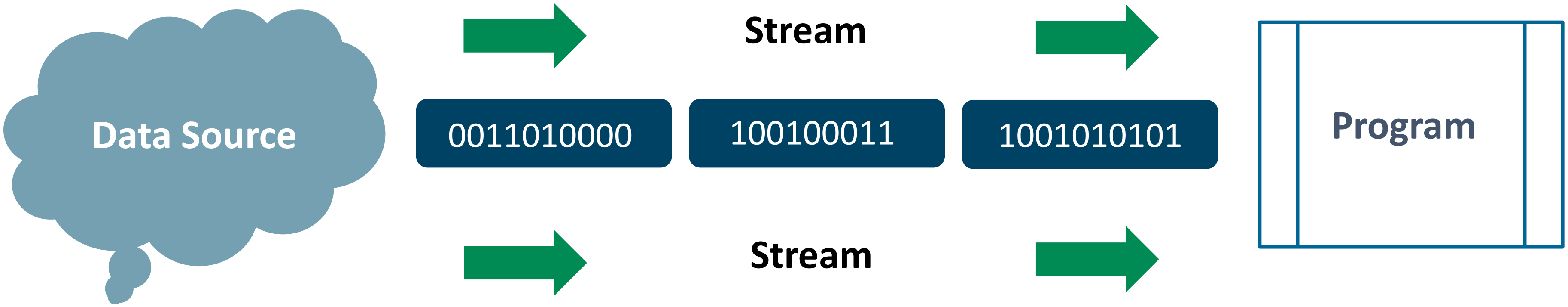
Streams are **created automatically** and are attached with **console**.

```
System.out.println("simple message");
System.err.println("error message");
```

```
int i=System.in.read();
//returns ASCII code of 1st character
System.out.println((char)i);
//will print the character
```

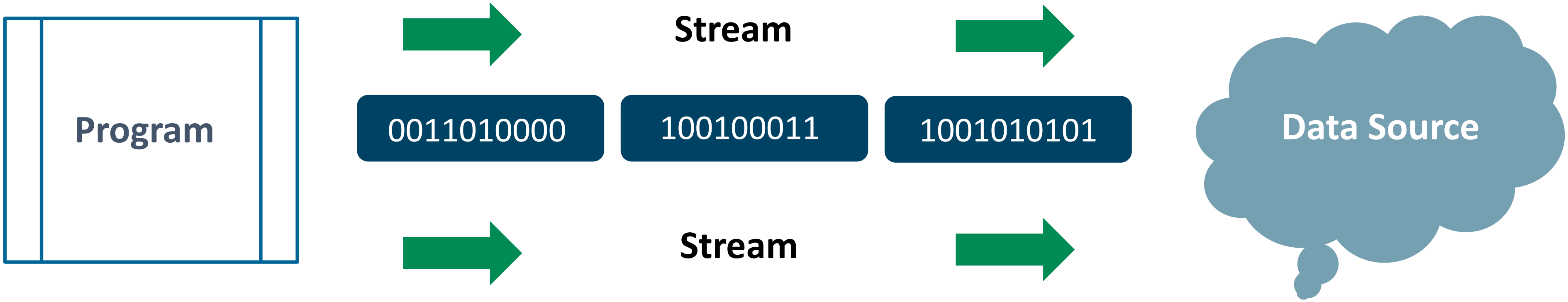
Input stream

A program uses an Input Stream to **read** data from a source.



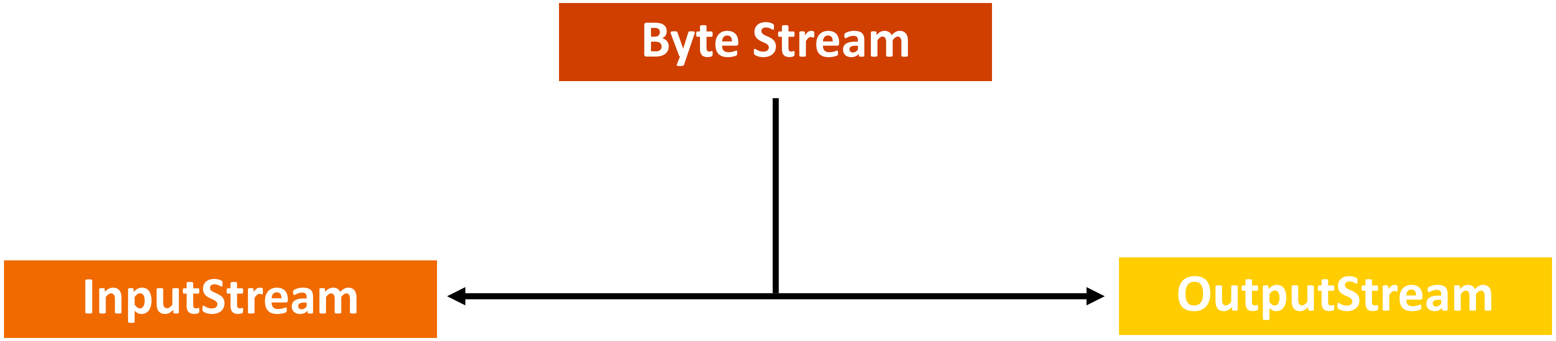
Output stream

A program uses an Output Stream to **write** data from a source.



Types of **byte stream**

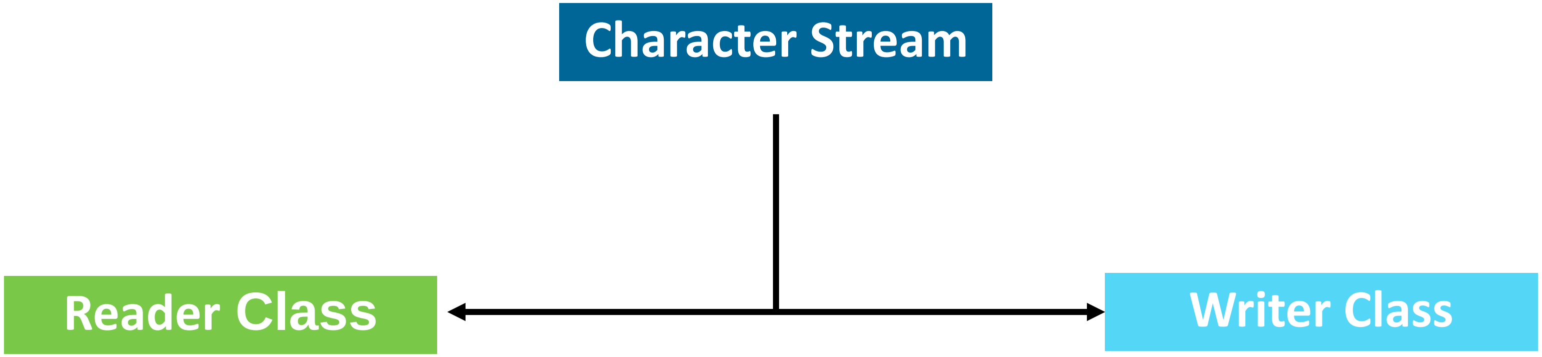
Byte stream is defined by two abstract classes:



These have several concrete classes that handle various devices such as disk files, network connection, and so on.

Types of **character stream**

Character stream is defined by two abstract classes:



These have several concrete classes that handle UNICODE character.

Reading **characters** and **strings**

Reading Characters:

```
class CharRead
{
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();       //Reading character
 }
}
```

Reading Strings:

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();          //Reading String
  System.out.println(text);
 }
}
```

An overview of I / O

When dealing with Input / Output two important things have to be identified:

Source

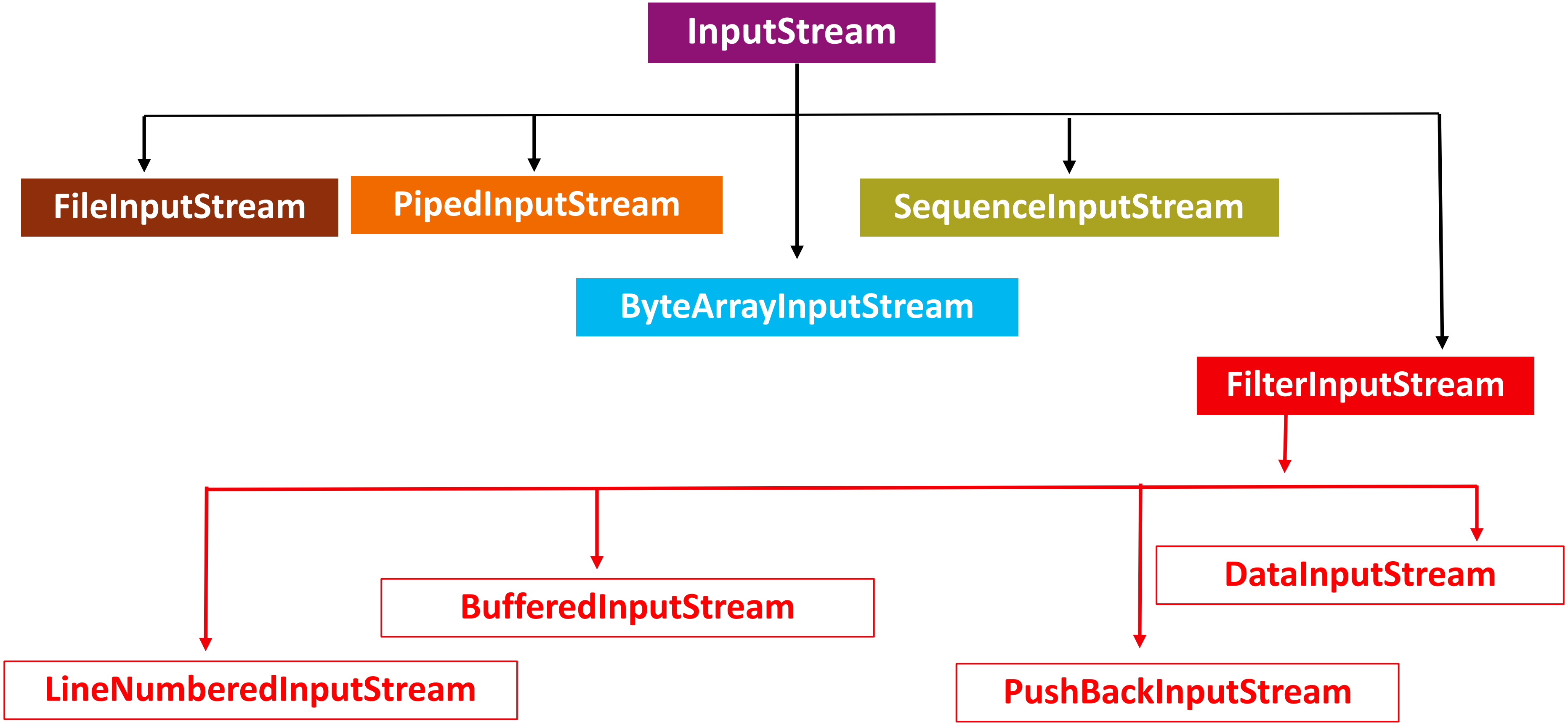
- From where **data** is read
- Associated with **InputStream**
- Uses **File, Socket, Array,** and **Thread** sub classes

Sink / Destination

- Where data is **written** to
- Associated with **OutputStream**
- Uses **File, Socket, Array,** and **Thread** sub classes

The InputStream class

InputStream class is an **abstract base class,** providing **minimal** programming interface and a **partial** implementation of input streams in Java.



Frequently used methods of InputStream

Frequently used methods of `InputStream` are:

```
public abstract int read() throws IOException
```

```
public int read(byte [] b) throws IOException
```

```
public long skip(long n) throws IOException
```

```
public void close() throws IOException
```

BufferedInputStream and its methods

- BufferedInputStreams **read data** from a **memory area** called **buffer**.
- Constructor used : **`BufferedInputStream(InputStream)`**

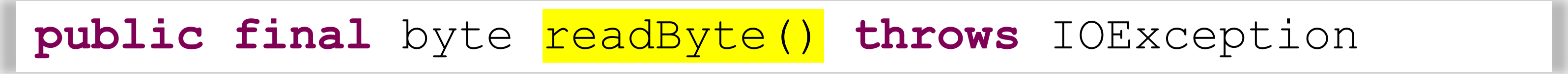
```
public int read() throws IOException
```

```
public int read(byte[] b, int off, int len) throws IOException
```

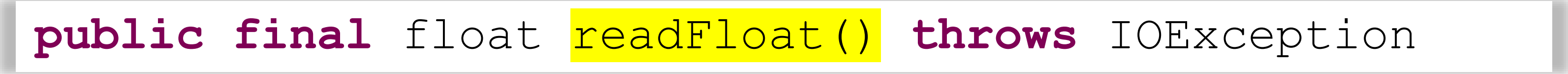
```
public void close() throws IOException
```

DataInputStream and its methods

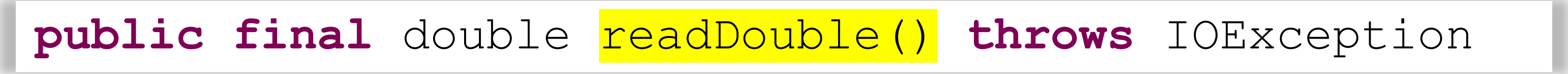
- DataInputStream allows an application to **read** primitive **Java data** types in a **machine – independent way**.
- Constructor used: **`DataInputStream(InputStream in)`**



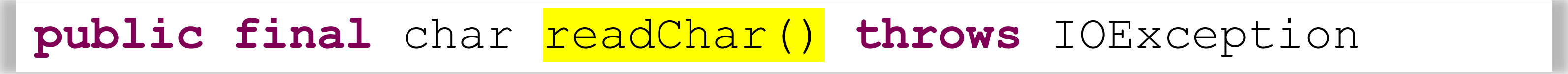
```
public final byte readByte() throws IOException
```



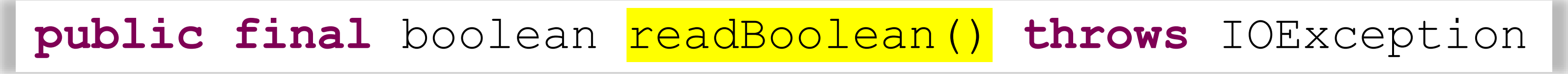
```
public final float readFloat() throws IOException
```



```
public final double readDouble() throws IOException
```



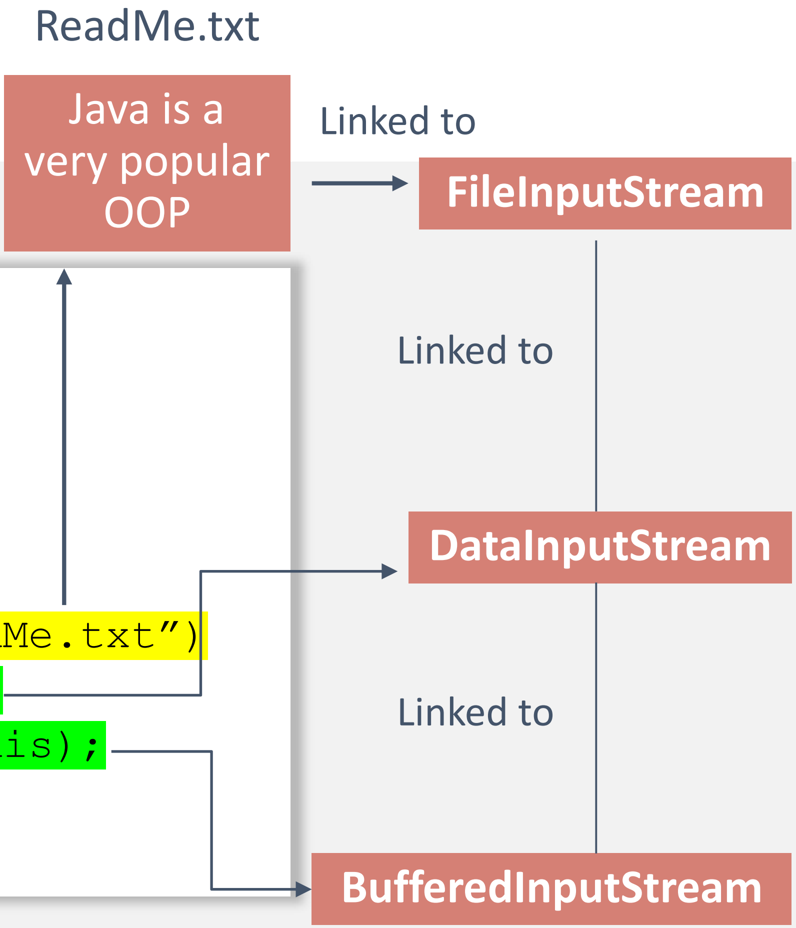
```
public final char readChar() throws IOException
```



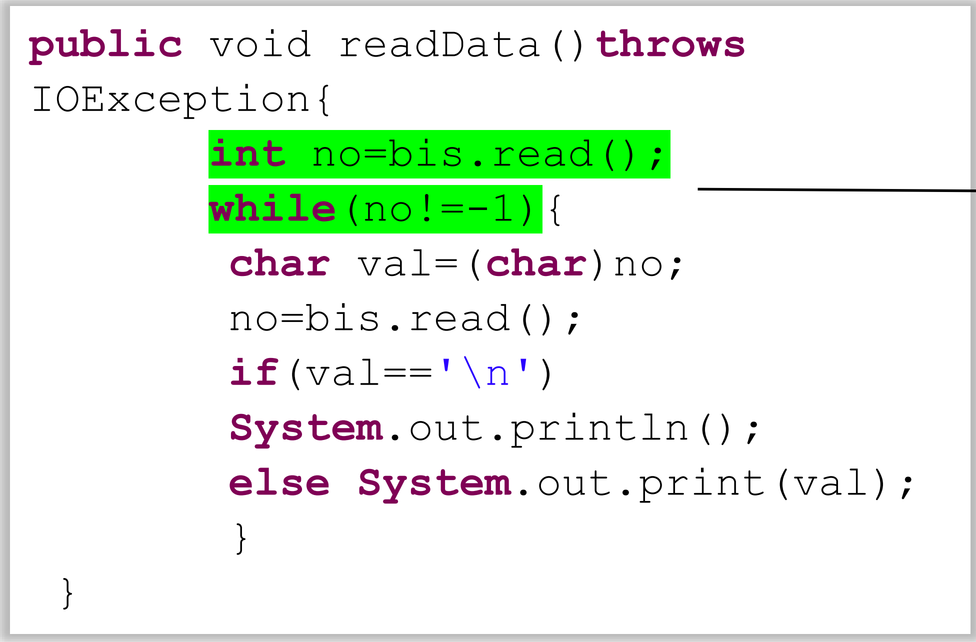
```
public final boolean readBoolean() throws IOException
```

Example for **reading** from a **File** (1 of 3)

```
import java.io.*;
public class ReadFile{
     FileInputStream fis;
     DataInputStream dis;
     BufferedInputStream bis;
     ReadFile() throws IOException{
          fis=new FileInputStream(“ReadMe.txt”)
          dis=new DataInputStream(fis);
          bis=new BufferedInputStream(dis);
                    }
     }
```



Example for **reading** from a **File** (2 of 3)



```
public void readData() throws
IOException{
        int no=bis.read();
        while(no!=-1){
         char val=(char)no;
         no=bis.read();
         if(val=='\n')
         System.out.println();
         else System.out.print(val);
         }
  }
```

Use the read() method of BufferedInputStream.

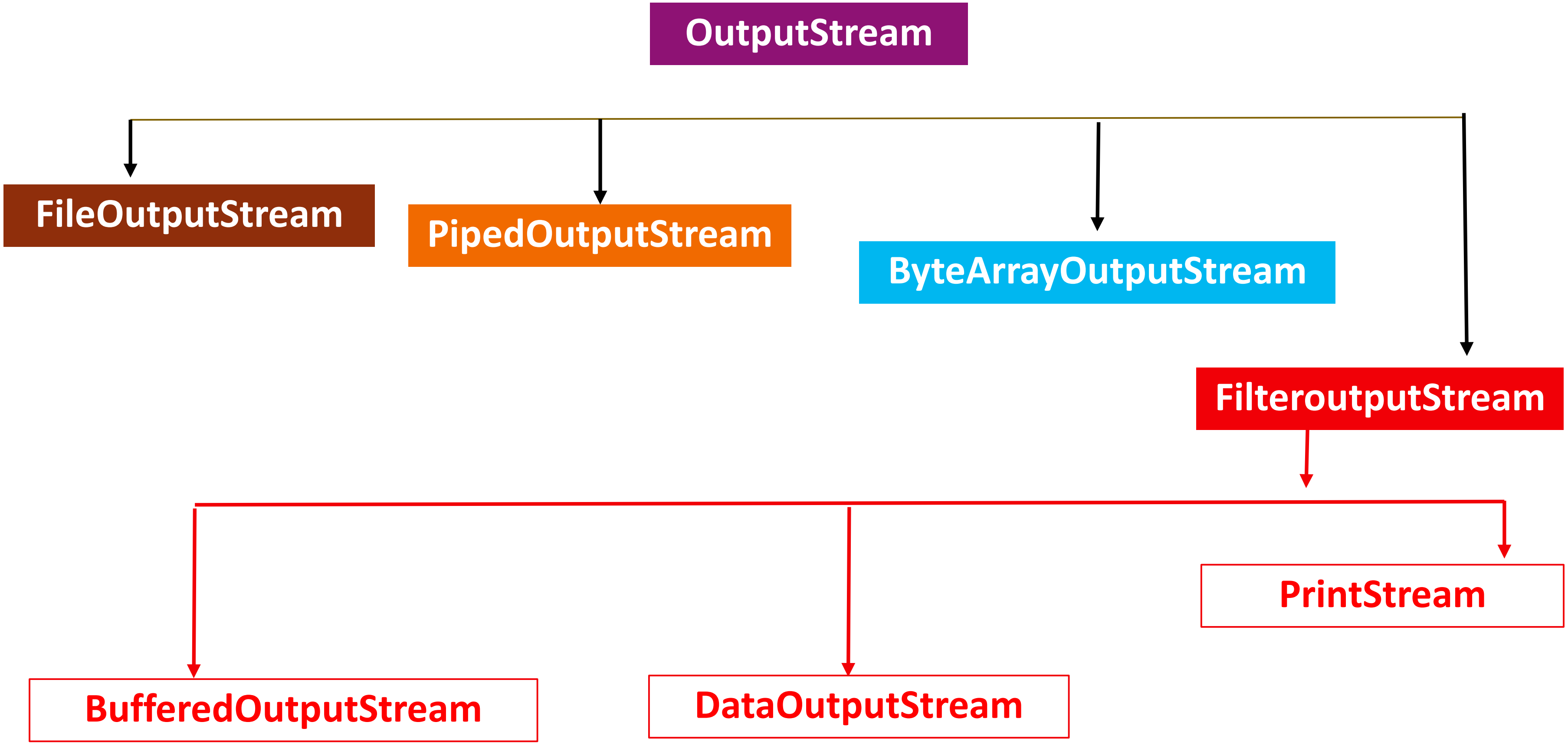
Example for **reading** from a **File** (3 of 3)

Code for the example:

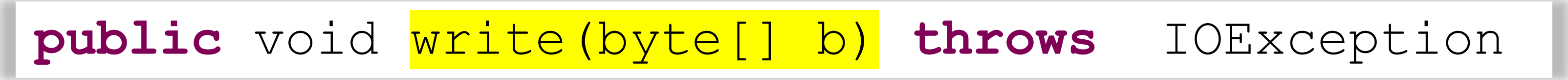
```java
public static void main(String s[])throws IOException
    {
        ReadFile f=new ReadFile();
        f.readData();
    }
```

The OutputStream class

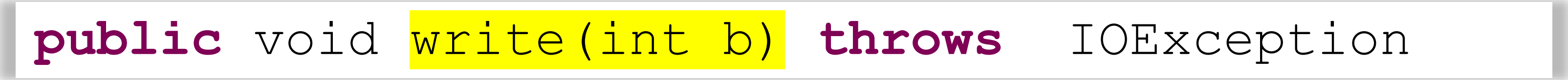
OutputStream class is an **abstract super class,** providing **minimal** programming interface and a **partial** implementation of output streams in Java.



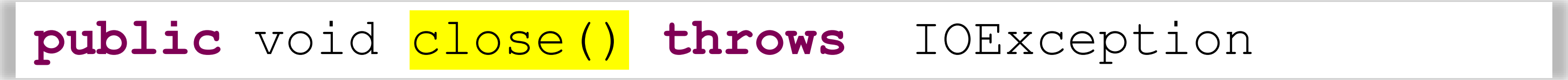
Frequently used **methods** of **OutputStream**



```
public void write(byte[] b) throws  IOException
```



```
public void write(int b) throws  IOException
```



```
public void close() throws  IOException
```

BufferedOutputStream and its methods

- BufferedOutPutStream allows to write to a stream without causing a write every time.
- The constructors used are: **`BufferedOutputStream()`**

  `BufferedOutputStream(int buffer)`

```
public void flush() throws IOException
```

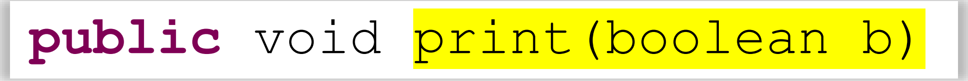
```
public void write(int i) throws IOException
```

```
public void write(byte [] b) throws IOException
```

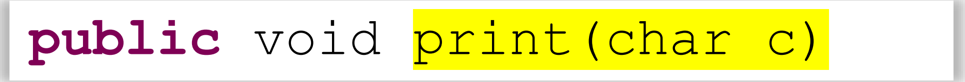
PrintStream and its methods

- PrintStream **adds functionality** to **another output stream**.
- The constructors used are: **`PrintStream(OutputStream out)`**

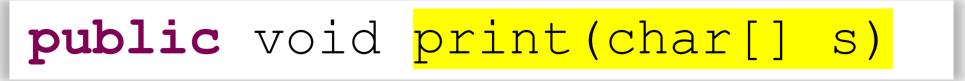
  `PrintStream(OutputStream out,boolean autoflush)`



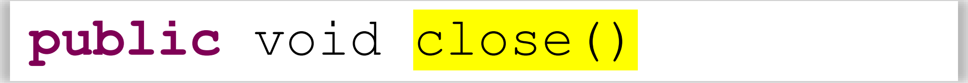
```
public void print(boolean b)
```



```
public void print(char c)
```



```
public void print(char[] s)
```



```
public void close()
```

Character stream and its uses

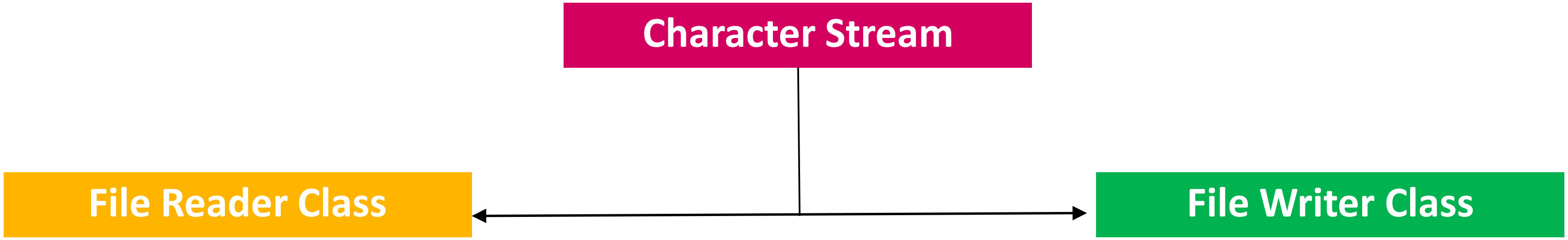
Character Stream

- Similar to byte streams
- Contain 16-bit UNICODE characters
- Has 2 classes: Reader and Writer
- Readers and Writers support same operations as InputStreams and OutputStreams

Why use Character Stream

- Make it easy to write programs
- Not dependent on a specific character coding
- Easy to internationalize
- More efficient than byte streams
- The classes are oriented around buffer-at-a-time read and write operations

Reader and writer classes



File Reader Class

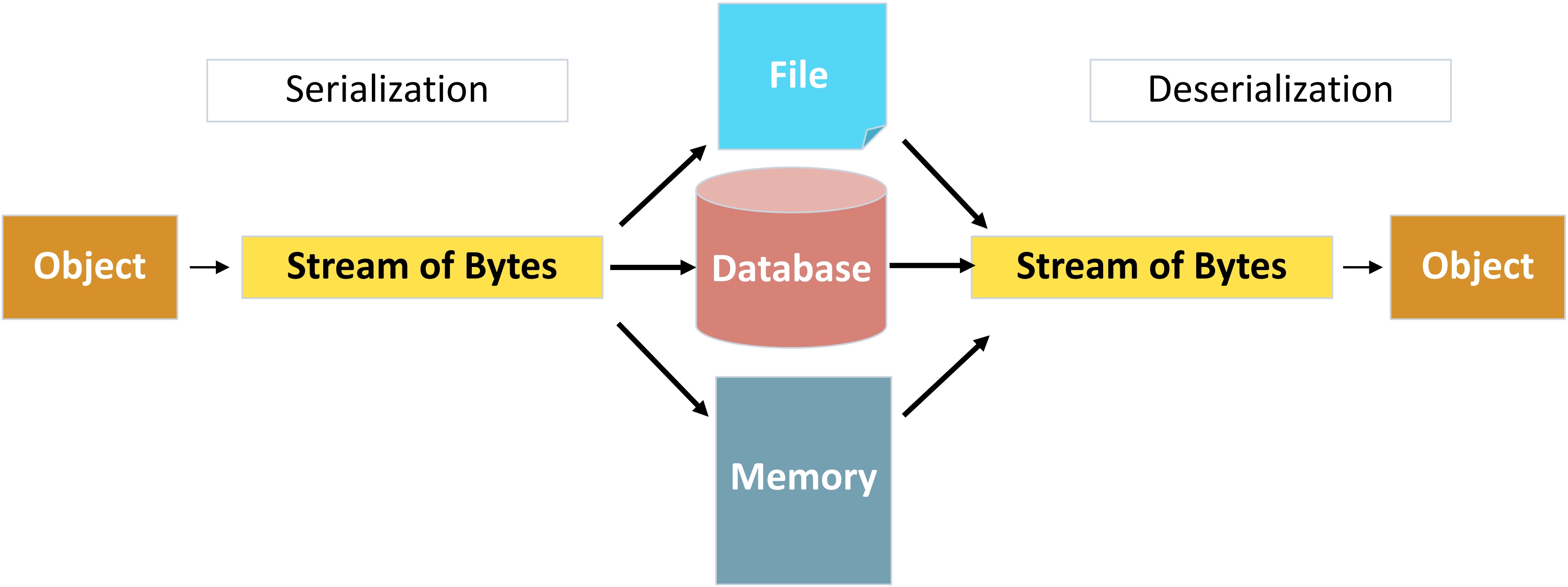
- Creates a **Reader** that can be used to **read a file**
- Constructors used:
  - FileReader(String filePath)
  - FileReader(File fileObject)

File Writer Class

- Creates a **Writer** that can be used **to write a file**
- Constructors used:
  - FileWriter(String filePath)
  - FileWriter(File fileObject)
  - FileWriter(String filePath, boolean append)
  - FileWriter(File fileObject, boolean append)

Java **object serialization**

Java object serialization is a **process** of **converting** an **object** into a **sequence of bytes** that can be persisted to a **disk, database**, or can be **sent through streams**.



The **reverse process** of **creating object** from **sequence of bytes** is **Deserialization**.

Serializing objects (1 of 3)

The **method** defined below **serializes an object** and **sends** it to the **OutputStream:**

```
public final void writeObject(Object x) throws IOException
```

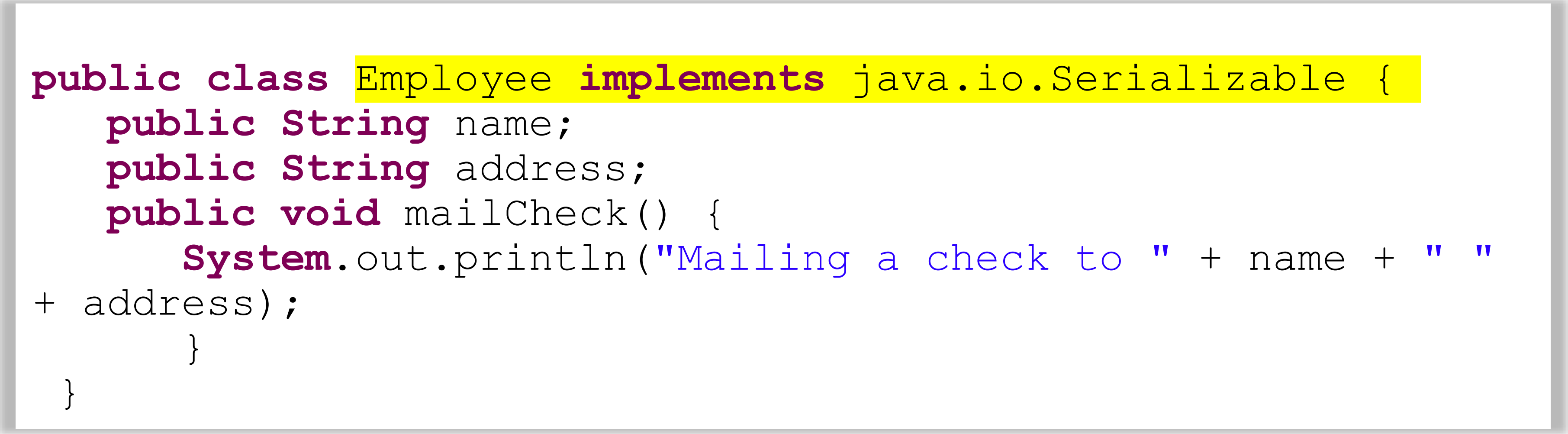
The **ObjectInputStream** class contains the following **method** for **deserializing** an object:

Note: The return value is Object, so casting to an apt data type is required.

```
public final Object readObject() throws IOException,ClassNotFoundException
```

Serializing objects (2 of 3)

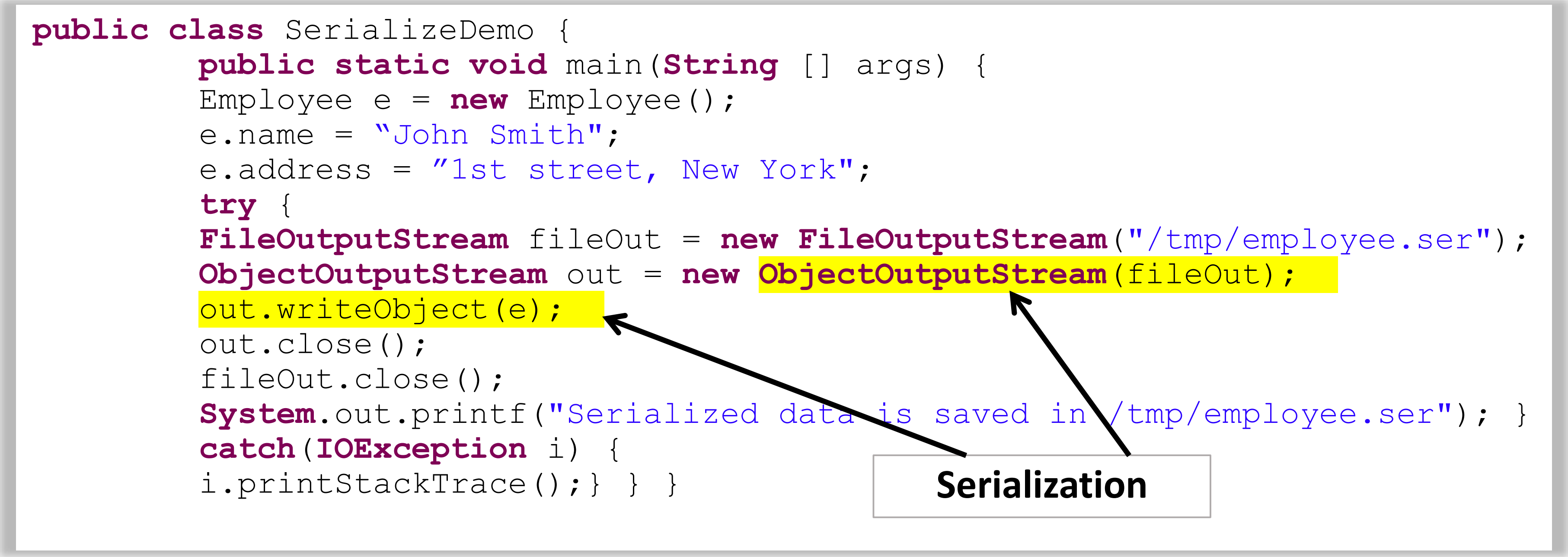
The class **Employee** implements the Serializable Interface.



```
public class Employee implements java.io.Serializable {
   public String name;
   public String address;
   public void mailCheck() {
      System.out.println("Mailing a check to " + name + " "
+ address);
      }
  }
```

Serializing objects (3 of 3)

Class **`Employee`** implements **Serializable Interface** and its object can be serialized.



```
public class SerializeDemo {
        public static void main(String [] args) {
        Employee e = new Employee();
        e.name = “John Smith";
        e.address = ”1st street, New York";
        try {
        FileOutputStream fileOut = new FileOutputStream("/tmp/employee.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(e);
        out.close();
        fileOut.close();
        System.out.printf("Serialized data is saved in /tmp/employee.ser"); }
        catch(IOException i) {
        i.printStackTrace();} } }
```

Serialization

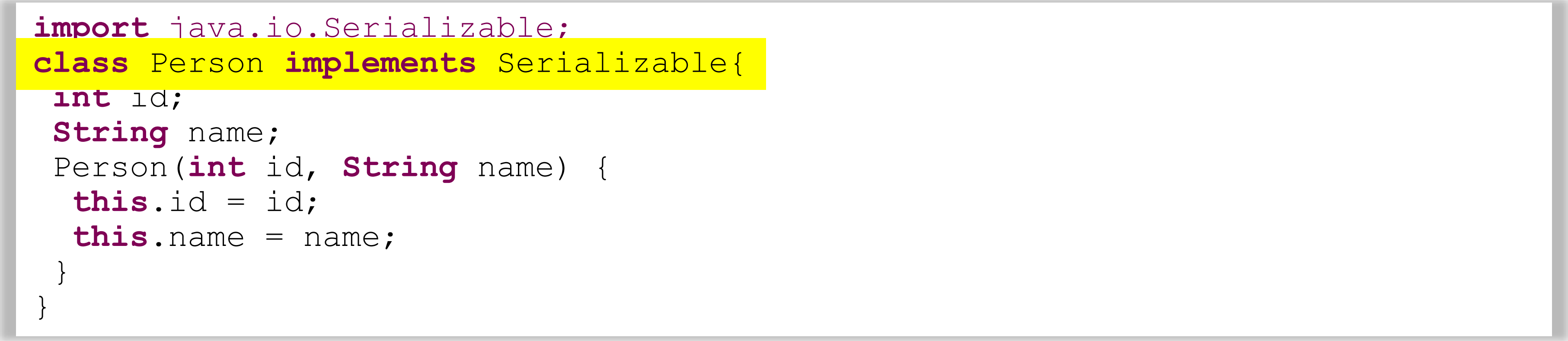
The static and transient keyword

Example of a class with static and transient variables:

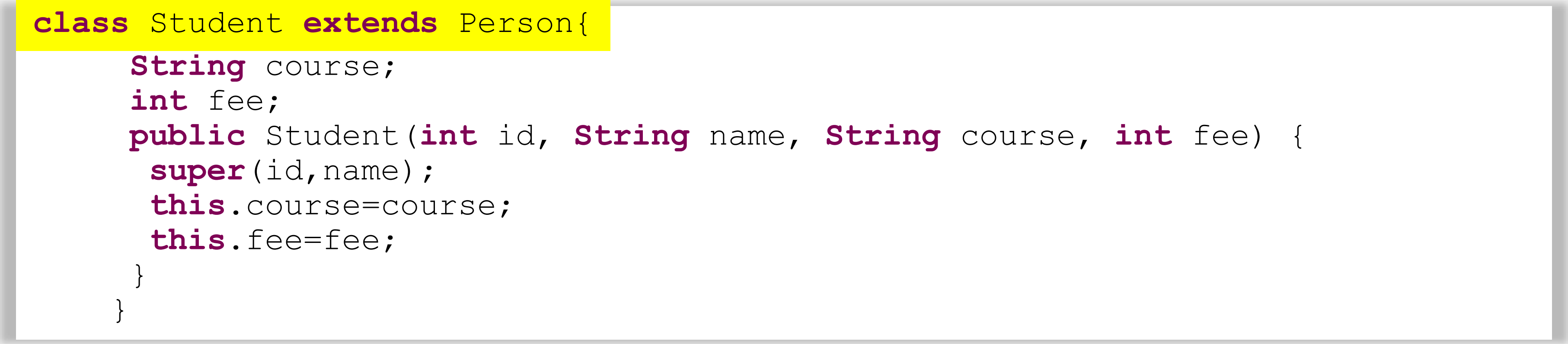
```
import java.io.Serializable;
    public class Student implements Serializable{
     int id;
     String name;
     transient int age;  //it will not be serialized
     static String schoolName = “Andrews School”; //not serialized

     public Student(int id, String name,int age, String school) {
      this.id = id;
      this.name = name;
      this.age=age;
      this.schoolName=schoolName;
     }
    }
```

Inheritance in serialization (IS – A relationship)

When a class implements serializable, all its sub classes will also be serializable.



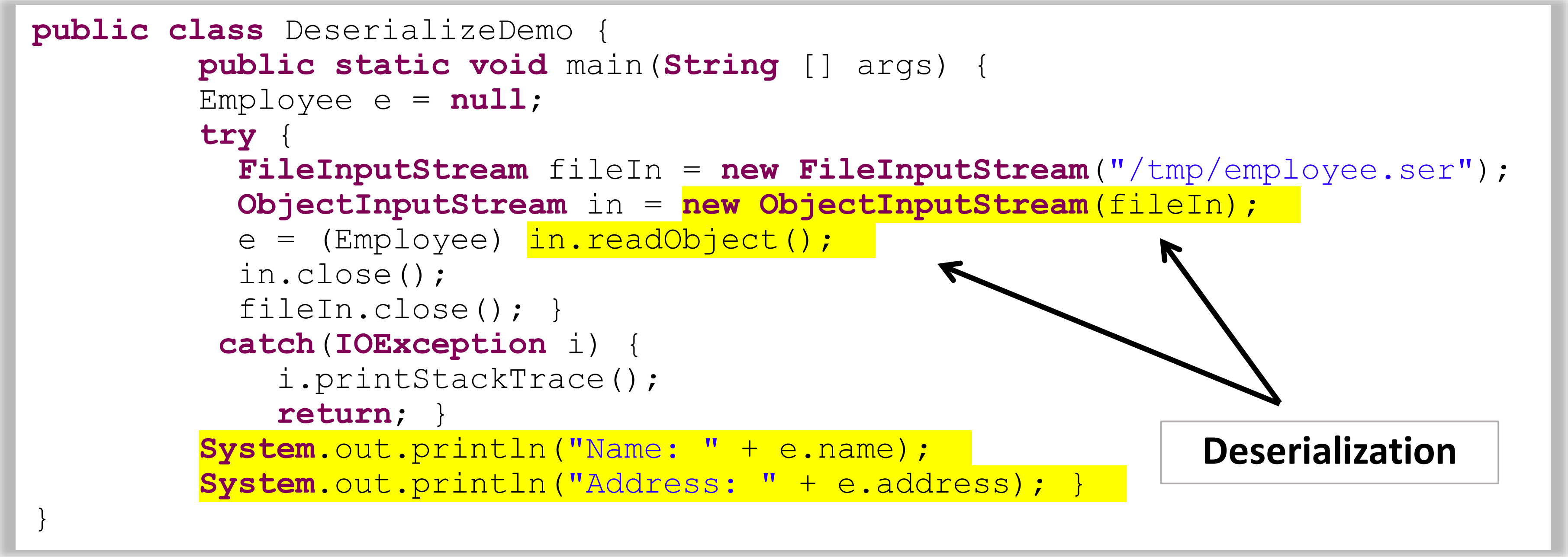
```java
import java.io.Serializable;
class Person implements Serializable{
 int id;
 String name;
 Person(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```



```java
class Student extends Person{
     String course;
     int fee;
     public Student(int id, String name, String course, int fee) {
      super(id,name);
      this.course=course;
      this.fee=fee;
     }
    }
```

Deserializing objects

Data is saved in the **`Employee`** object.



```
public class DeserializeDemo {
        public static void main(String [] args) {
        Employee e = null;
        try {
          FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
          ObjectInputStream in = new ObjectInputStream(fileIn);
          e = (Employee) in.readObject();
          in.close();
          fileIn.close(); }
         catch(IOException i) {
            i.printStackTrace();
            return; }
        System.out.println("Name: " + e.name);
        System.out.println("Address: " + e.address); }
}
```

Spot quiz

04

Which of these classes are used by character streams for input and output operations?

A InputStream

B ReadStream

C InputOutputStream

D Writer

Spot quiz

05

Which of these is method is used for writing bytes to an outputstream?

A `put()`

B `print()`

C `printf()`

D `write()`

Spot quiz

06

Which of these is a process of converting an object into a sequence of bytes that can be sent through streams?

A Garbage collection

B File Filtering

C Externalization

D Serialization

Questions?

03 File I / O (Input / Output)

What is **file class**?

A file class is an **abstract** representation of **file** and **directory pathnames**.

```
public class File
      extends Object
            implements Serializable, Comparable<File>
```

NOTE: It can not be used to read/write the content of files.

How to **instantiate** file object?

A file object can be instantiated in the following way:

```
File file = new File("c:\\data\\input-file.txt");
```

Constructors used to create file objects

The constructors commonly used to create file objects are:

```
File(File parent, String child)
```

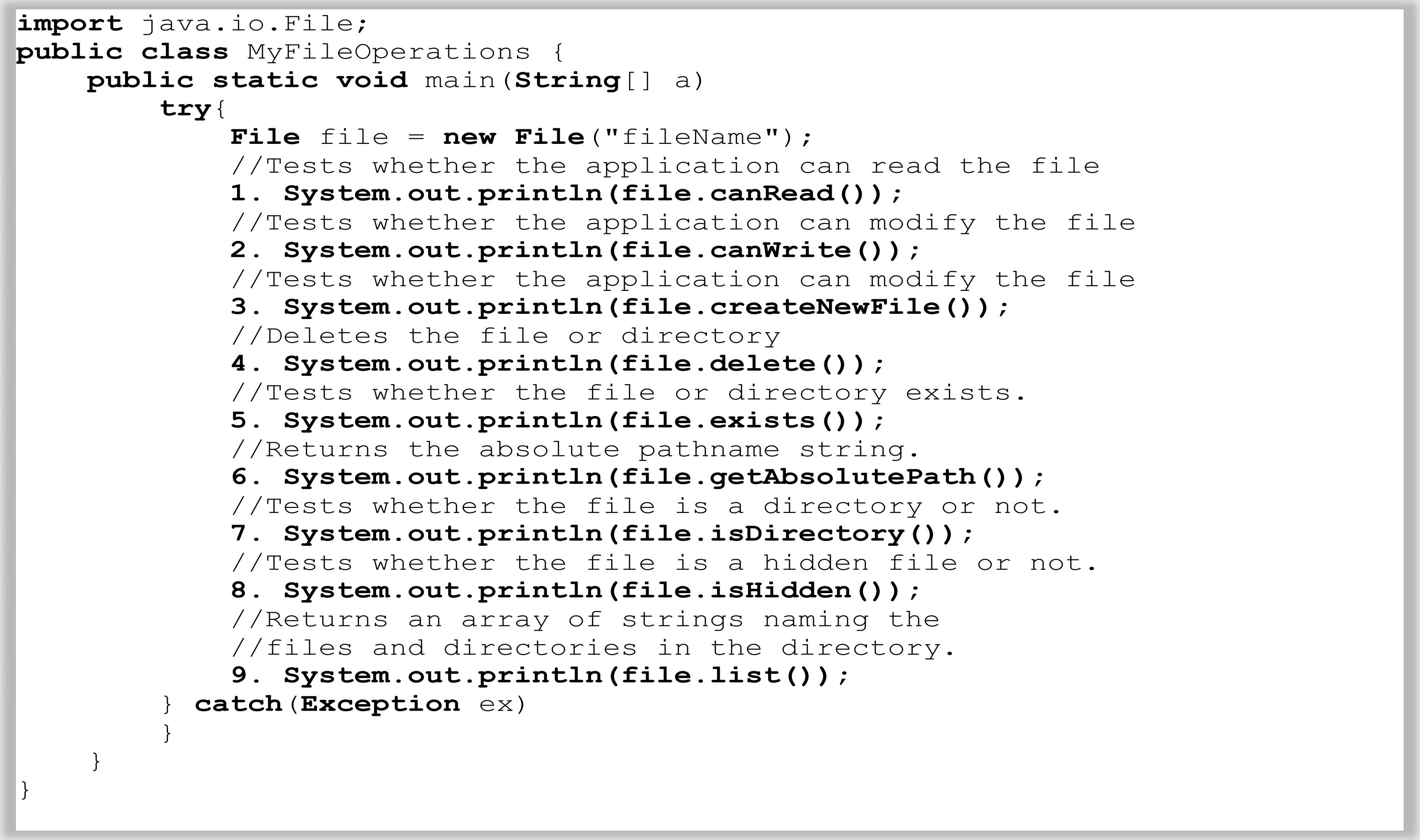
```
string.File(String pathname)
```

```
File(String parent, String child)
```

```
File(URI uri)
```

File methods

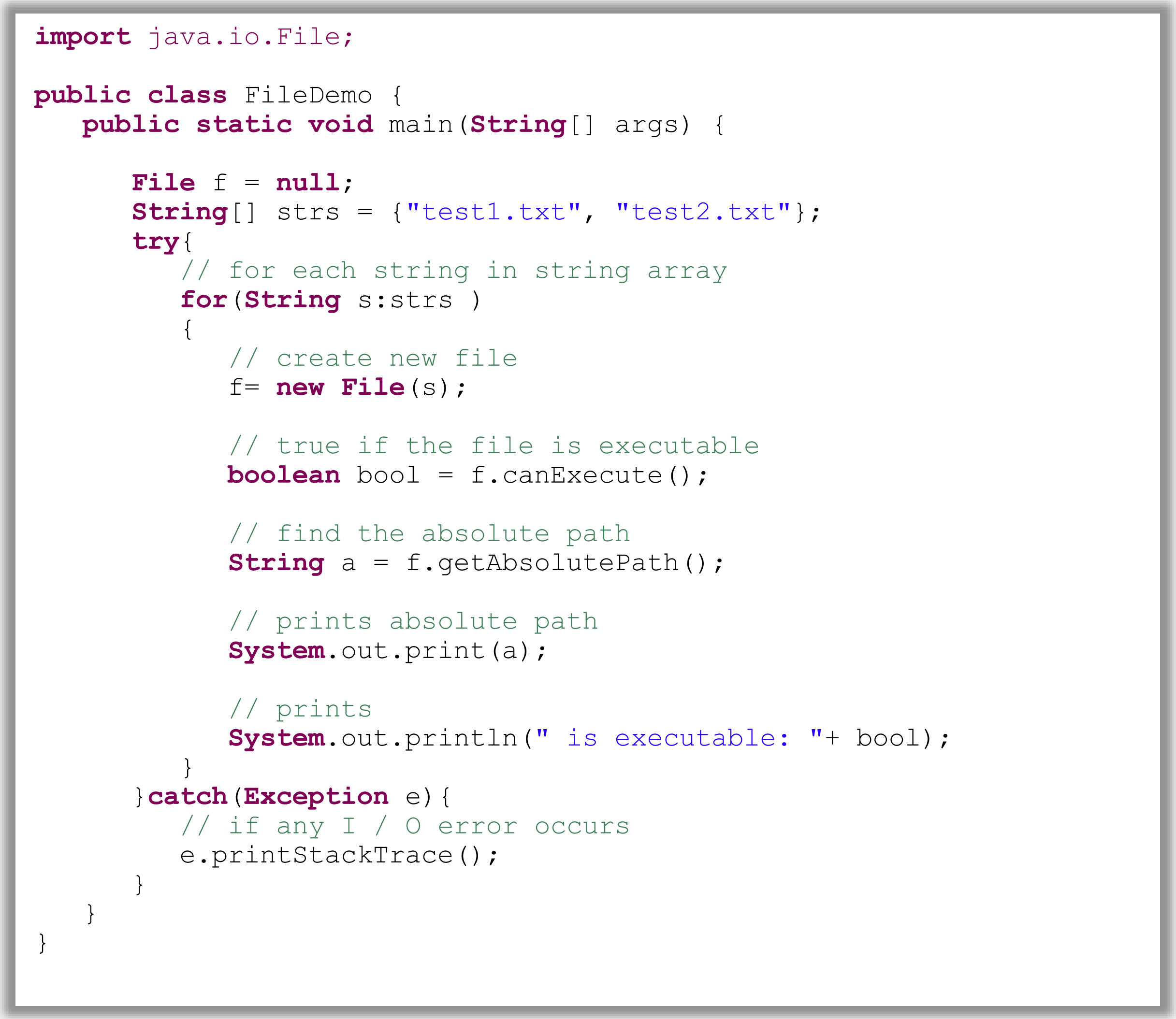
The file methods are:



```
import java.io.File;
public class MyFileOperations {
    public static void main(String[] a)
        try{
            File file = new File("fileName");
            //Tests whether the application can read the file
            1. System.out.println(file.canRead());
            //Tests whether the application can modify the file
            2. System.out.println(file.canWrite());
            //Tests whether the application can modify the file
            3. System.out.println(file.createNewFile());
            //Deletes the file or directory
            4. System.out.println(file.delete());
            //Tests whether the file or directory exists.
            5. System.out.println(file.exists());
            //Returns the absolute pathname string.
            6. System.out.println(file.getAbsolutePath());
            //Tests whether the file is a directory or not.
            7. System.out.println(file.isDirectory());
            //Tests whether the file is a hidden file or not.
            8. System.out.println(file.isHidden());
            //Returns an array of strings naming the
            //files and directories in the directory.
            9. System.out.println(file.list());
        } catch(Exception ex)
        }
    }
}
```

File object

This example demonstrates File object:



```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {

      File f = null;
      String[] strs = {"test1.txt", "test2.txt"};
      try{
         // for each string in string array
         for(String s:strs )
         {
            // create new file
            f= new File(s);

            // true if the file is executable
            boolean bool = f.canExecute();

            // find the absolute path
            String a = f.getAbsolutePath();

            // prints absolute path
            System.out.print(a);

            // prints
            System.out.println(" is executable: "+ bool);
         }
      }catch(Exception e){
         // if any I / O error occurs
         e.printStackTrace();
      }
   }
}
```

Spot quiz

07

Which of the following syntax creates a new File instance by converting the given pathname string to an abstract pathname ?

A File(URI uri)

B File(String parent, String child)

C string.File(String pathname)

D File(parent, String child)

Questions?

A c t i v i t y

Day 04 Practice Exercises:

Exceptions

1. Use Java Exception handling to enhance the exercise on Shapes to get input from user and validate the input.

I/O

1. Write a program the reads an input file ("input.txt") and creates a new file and copies the content to an output file ("output.txt").
2. Write a program that reads an input file ("input.txt") and outputs the content in console. New lines should also be outputted. The program should also output the number of lines and the number of words. The given file ("input. txt") has 5 lines and 108 words in total. (no more no less)