CST 205 : MODULE 3

Mrs. Preema Theresa Varghese

Assistant Professor

AI & DS

Sjcet, Palai

Module 3

- Packages and Interfaces
  - Defining Package
  - CLASSPATH
  - Access Protection
  - Importing Packages
  - Interfaces
- Exception Handling
  - Checked Exceptions
  - Unchecked Exceptions
  - try Block and catch Clause
  - Multiple catch Clauses
  - Nested try Statements
  - Throw
  - throws and finally

- Input/Output
  - I/O Basics
  - Reading Console Input
  - Writing Console Output
  - PrintWriter Class
  - Object Streams and Serialization
  - Working with Files

Packages

- Java package is a mechanism of grouping similar type of classes, interfaces, and sub-classes collectively based on functionality.
- Advantages
  - **Re-usability**: The classes contained in the packages of another program can be easily reused
  - **Name Conflicts**: Packages help us to uniquely identify a class, for example, we can have company.sales.Employee and company.marketing.Employee classes
  - **Controlled Access**: Offers access protection such as protected classes, default classes and private class
  - **Data Encapsulation**: They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only
  - **Maintenance**: With packages, you can organize your project better and easily locate related classes

- **Types of Packages in Java**
- Based on whether the package is defined by the user or not, packages are divided into two categories:
  - Built-in Packages
  - User Defined Packages

Defining Package

- package command is used to create the package
- If you omit the package statement, the class names are put into the default package, which has no name.

- This is the general form of the package statement:

```
package pkg;
```

- For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

-

- You can create a hierarchy of packages.
- To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

- For example, a package declared as

package com.sjcet.oopj;

-

CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
  - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
  - Third, you can use the -classpath option with java and javac to specify the path to your classes.

- **Set CLASSPATH System Variable**
- To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) −
  - In Windows → C:\> set CLASSPATH
  - In UNIX → % echo $CLASSPATH
- To delete the current contents of the CLASSPATH variable, use −
  - In Windows → C:\> set CLASSPATH =
  - In UNIX → % unset CLASSPATH; export CLASSPATH
- To set the CLASSPATH variable −
  - In Windows → set CLASSPATH = C:\users\jack\java\classes
  - In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

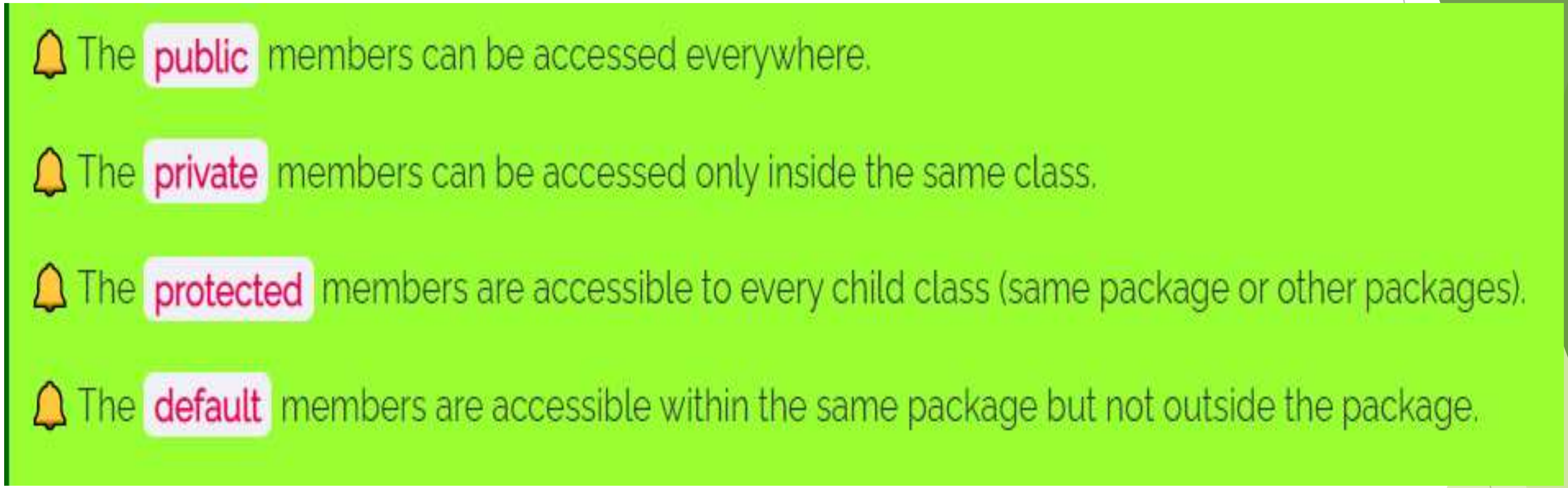
Access Protection

- In java, the access modifiers define the accessibility of the class and its members.
- For example, private members are accessible within the same class members only.
- Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages.
- Packages are meant for encapsulating, it works as containers for classes and other subpackages.
- Class acts as containers for data and methods.
- There are four categories, provided by Java regarding the visibility of the class members between classes and packages:
- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

- Java has four access modifiers, and they are default, private, protected, and public.
- In java, the package is a container of classes, sub-classes, interfaces, and sub-packages.
- The class acts as a container of data and methods.
- So, the access modifier decides the accessibility of class members across the different packages.
- In java, the accessibility of the members of a class or interface depends on its access specifiers.
- The following table provides information about the visibility of both data members and methods
- Simply remember, private cannot be seen outside of its class, public can be access from anywhere, and protected can be accessible in subclass only in the hierarchy.

Access control for members of class and interface in java

Accessibility Location / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

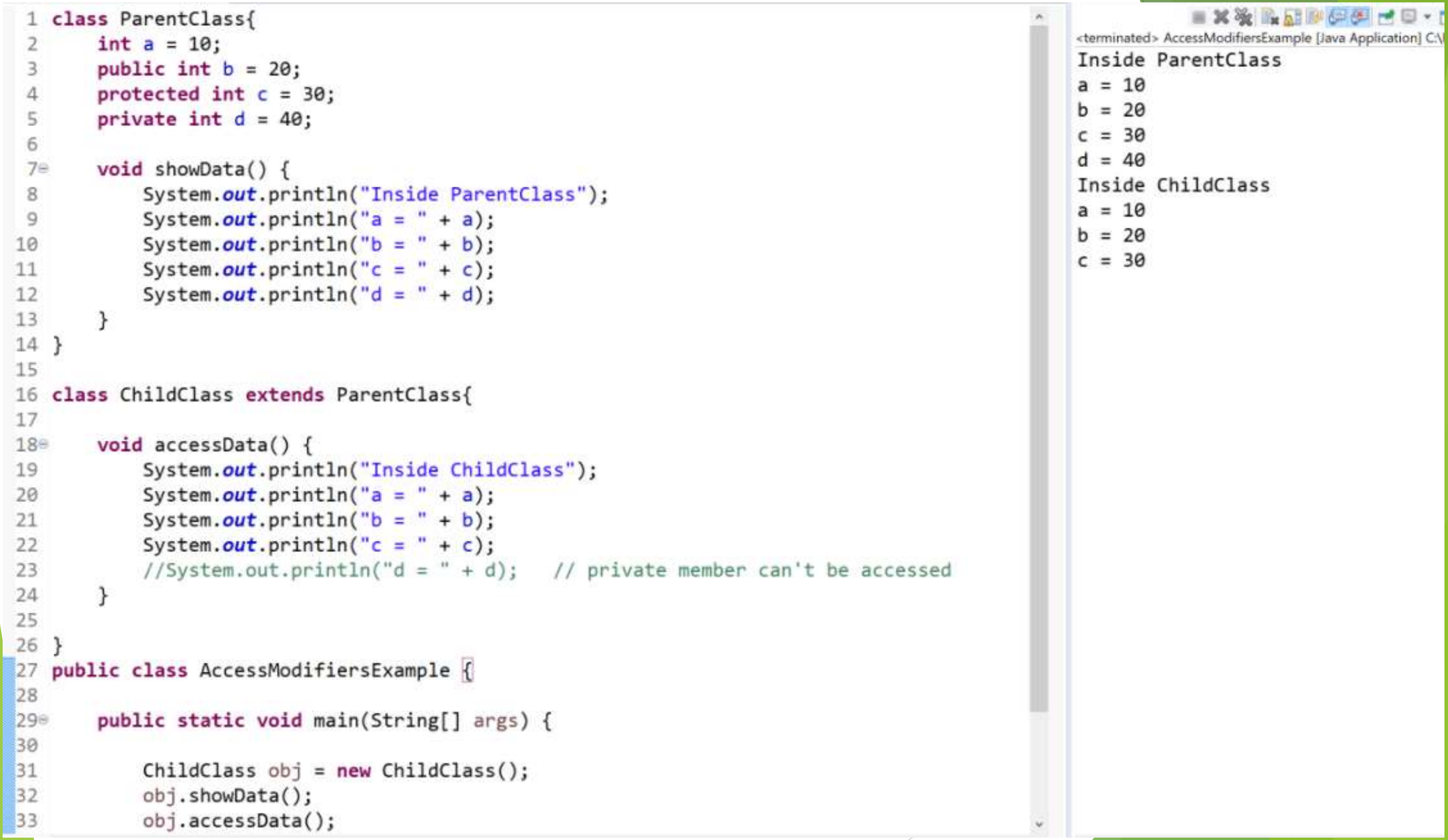


🔔 The `public` members can be accessed everywhere.

🔔 The `private` members can be accessed only inside the same class.

🔔 The `protected` members are accessible to every child class (same package or other packages).

🔔 The `default` members are accessible within the same package but not outside the package.



```java
class ParentClass{
    int a = 10;
    public int b = 20;
    protected int c = 30;
    private int d = 40;

    void showData() {
        System.out.println("Inside ParentClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

class ChildClass extends ParentClass{

    void accessData() {
        System.out.println("Inside ChildClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        //System.out.println("d = " + d);    // private member can't be accessed
    }

}
public class AccessModifiersExample {

    public static void main(String[] args) {

        ChildClass obj = new ChildClass();
        obj.showData();
        obj.accessData();
```

```
<terminated> AccessModifiersExample [Java Application] C:\
Inside ParentClass
a = 10
b = 20
c = 30
d = 40
Inside ChildClass
a = 10
b = 20
c = 30
```

Importing Packages

- In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

- This is the general form of the import statement:

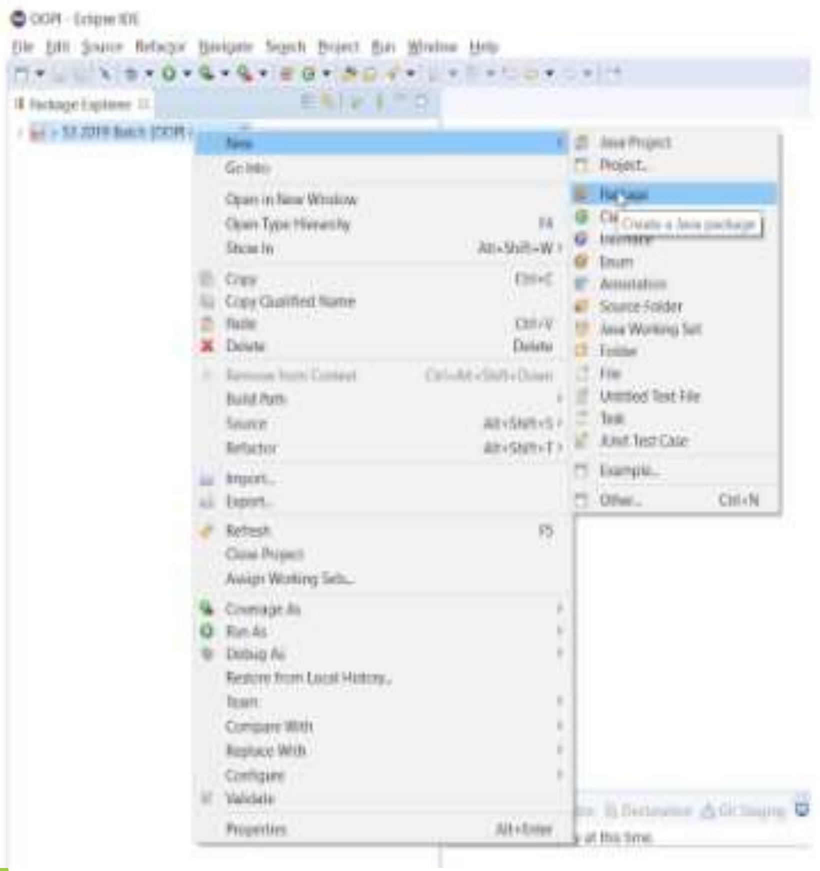
  import *pkg1* [.*pkg2*].(*classname* | *);

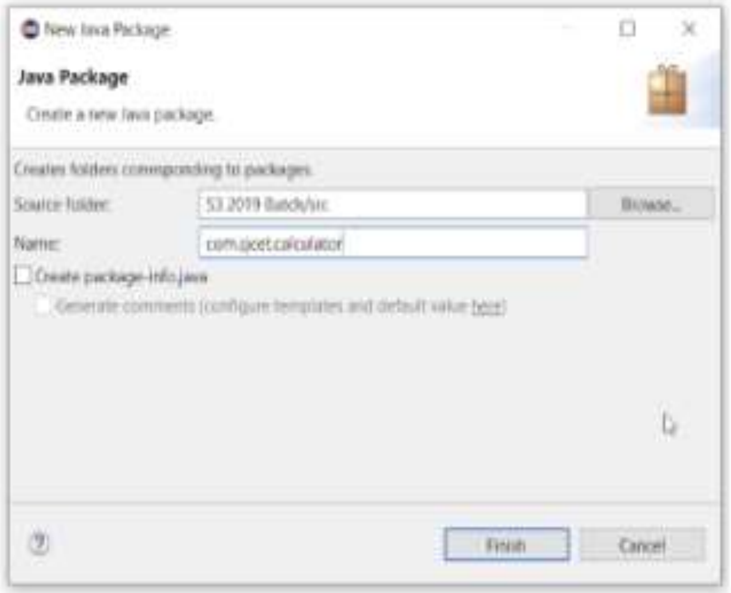
- Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

- Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

```
import java.util.Date;
import java.io.*;
```

How to create a package in Eclipse IDE




New Java Package

Java Package

Create a new Java package.

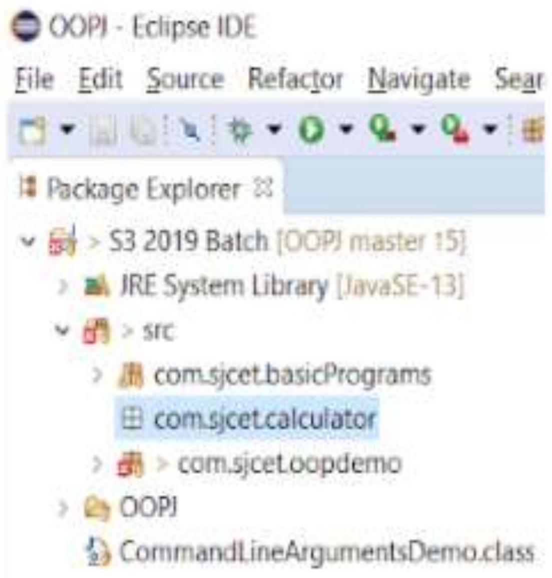
Creates folders corresponding to packages.

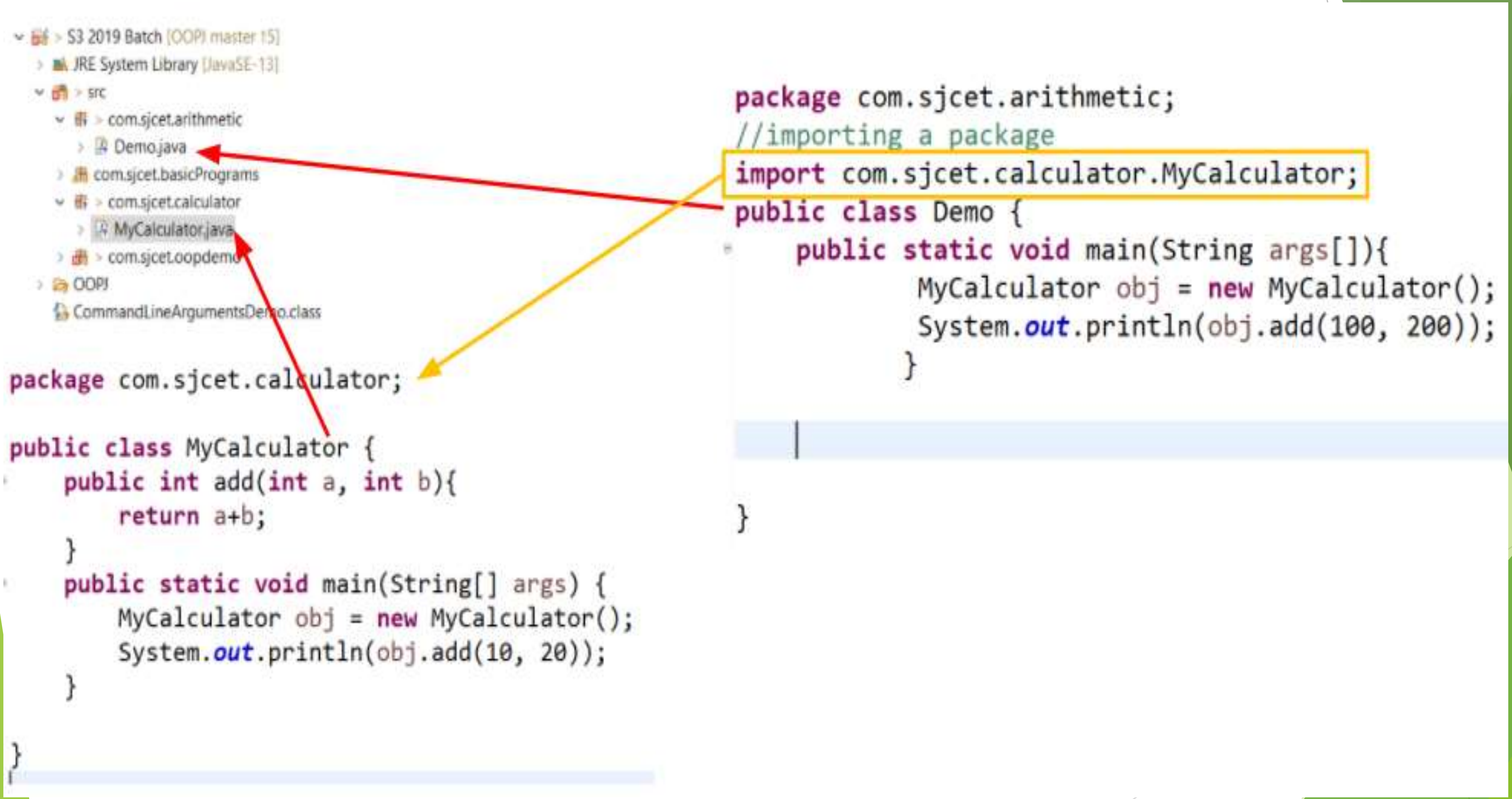
Source folder: S3 2019 Batch/src

Name: com.sjcet.calculator

[ ] Create package-info.java

[ ] Generate comments (configure templates and default value here)


OOPI - Eclipse IDE
File Edit Source Refactor Navigate Sear
Package Explorer
> S3 2019 Batch [OOPJ master ↑5]
JRE System Library [JavaSE-13]
> src
com.sjcet.basicPrograms
com.sjcet.calculator
> com.sjcet.oopdemo
OOPJ
CommandLineArgumentsDemo.class



```
S3 2019 Batch [OOPJ master 15]
  JRE System Library [JavaSE-13]
  src
    com.sjcet.arithmetic
      Demo.java
    com.sjcet.basicPrograms
    com.sjcet.calculator
      MyCalculator.java
    com.sjcet.oopdemo
  OOPJ
  CommandLineArgumentsDemo.class
```

```java
package com.sjcet.calculator;

public class MyCalculator {
    public int add(int a, int b){
        return a+b;
    }
    public static void main(String[] args) {
        MyCalculator obj = new MyCalculator();
        System.out.println(obj.add(10, 20));
    }

}
```

```java
package com.sjcet.arithmetic;
//importing a package
import com.sjcet.calculator.MyCalculator;
public class Demo {
    public static void main(String args[]){
            MyCalculator obj = new MyCalculator();
            System.out.println(obj.add(100, 200));
          }


}
```

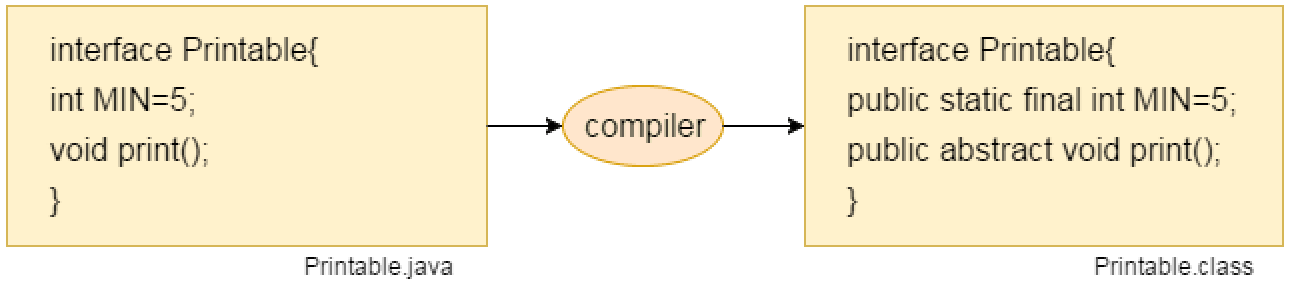
Interfaces

- In the Java programming language, an interface is a reference type, similar to a class.
- *interface* keyword is used to define an interface
- Contains
  - Constants
  - Method signatures
  - Default methods
  - Static methods
  - Nested types.

```
interface <interface_name> {

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

- Method bodies exist only for default methods and static methods.

- The variables declared in an interface are public, static & final by default.
- Methods declared in interface are by default abstract
-



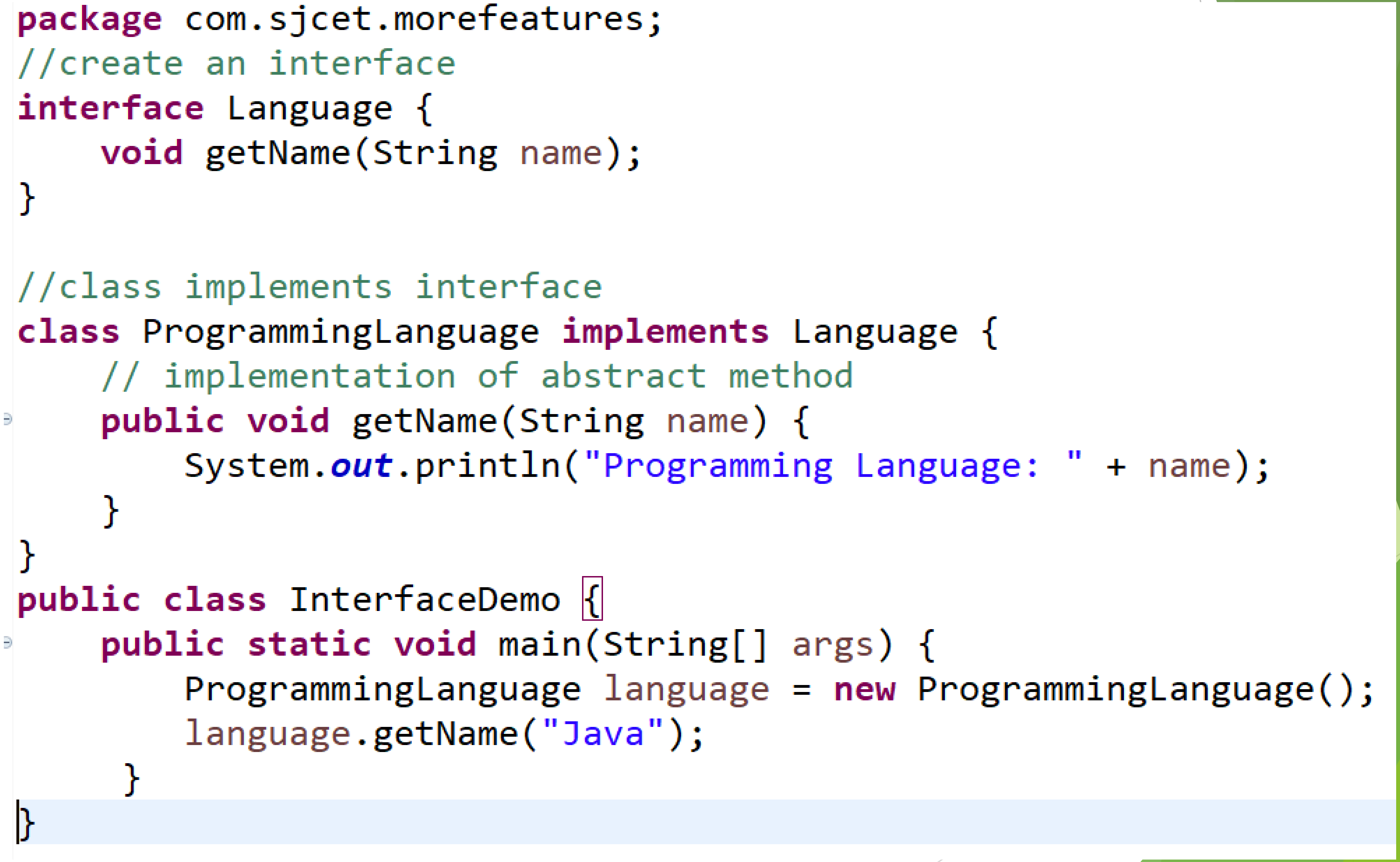
```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

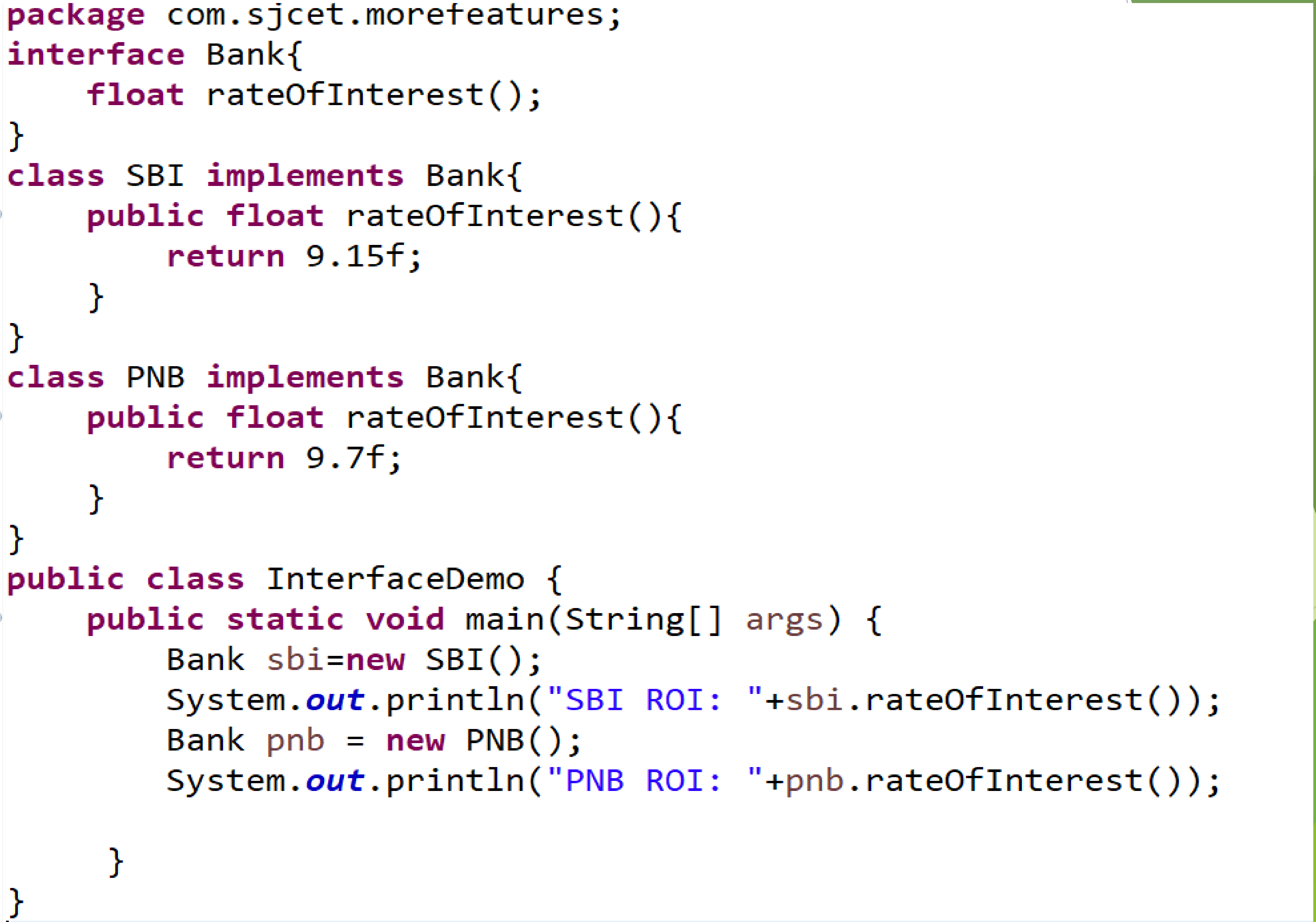
- **Implementing an Interface**
- Like abstract classes, we cannot create objects of an interface. However, we can implement an interface.
- We use the *implements* keyword to implement an interface

```
// create an interface
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}
```

Interfaces in Java Example

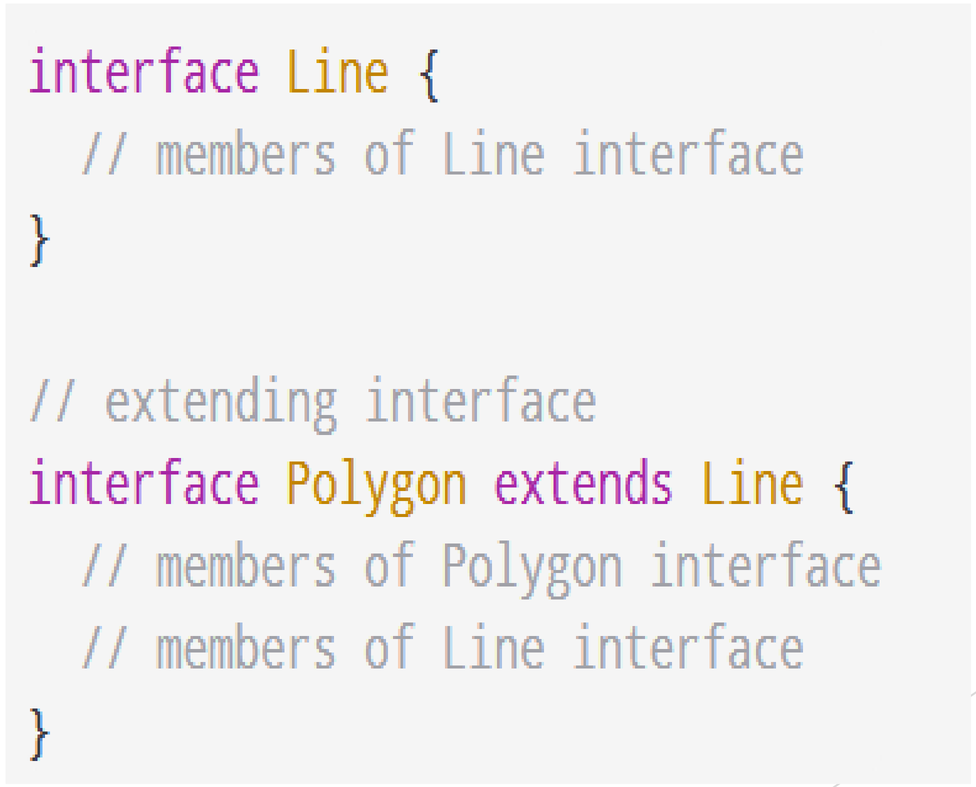


```java
package com.sjcet.morefeatures;
//create an interface
interface Language {
    void getName(String name);
}

//class implements interface
class ProgrammingLanguage implements Language {
    // implementation of abstract method
    public void getName(String name) {
        System.out.println("Programming Language: " + name);
    }
}
public class InterfaceDemo {
    public static void main(String[] args) {
        ProgrammingLanguage language = new ProgrammingLanguage();
        language.getName("Java");
     }
}
```

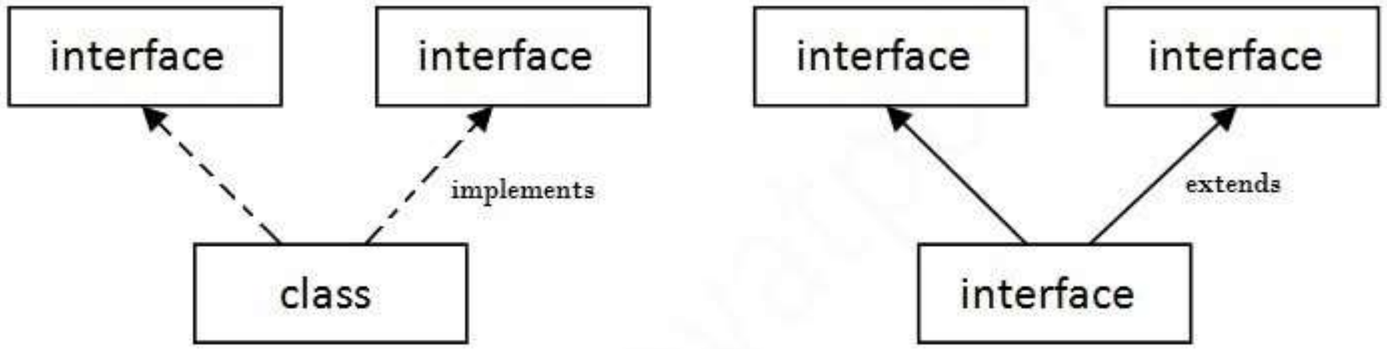


```java
package com.sjcet.morefeatures;
interface Bank{
    float rateOfInterest();
}
class SBI implements Bank{
    public float rateOfInterest(){
        return 9.15f;
    }
}
class PNB implements Bank{
    public float rateOfInterest(){
        return 9.7f;
    }
}
public class InterfaceDemo {
    public static void main(String[] args) {
        Bank sbi=new SBI();
        System.out.println("SBI ROI: "+sbi.rateOfInterest());
        Bank pnb = new PNB();
        System.out.println("PNB ROI: "+pnb.rateOfInterest());

     }
}
```

- **Extending an Interface**
- Similar to classes, interfaces can extend other interfaces.
- The extends keyword is used for extending interfaces. For example
- Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon



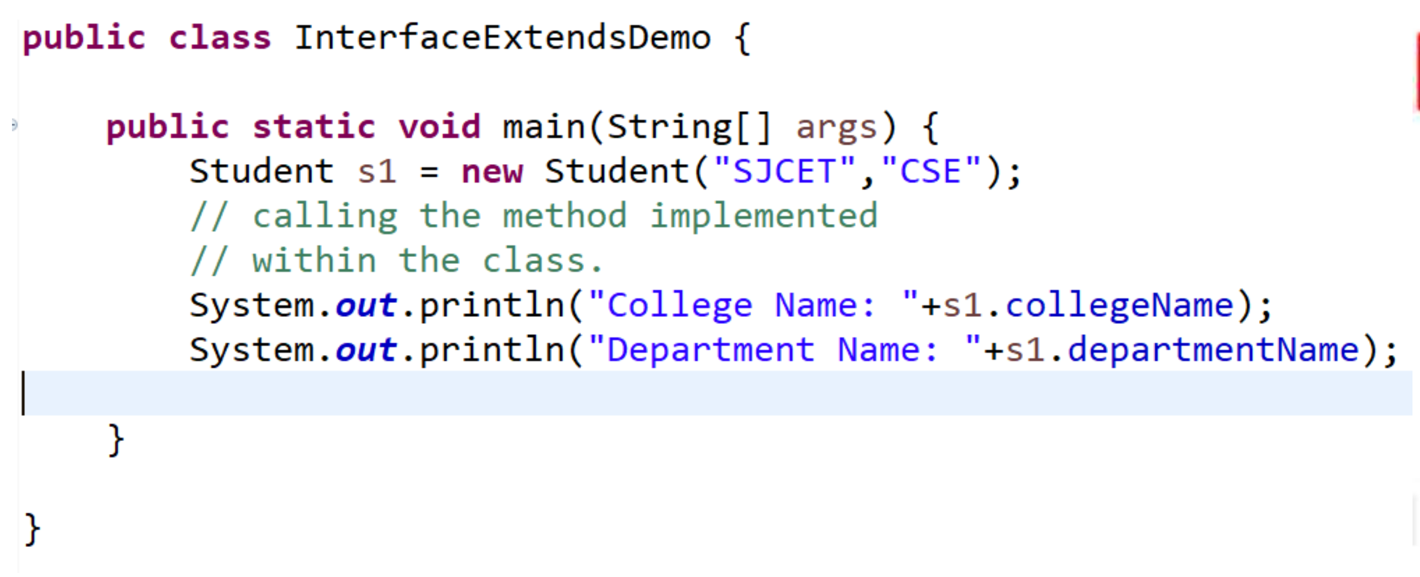
```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

- **Multiple inheritance in Java by interface**
- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.
- 

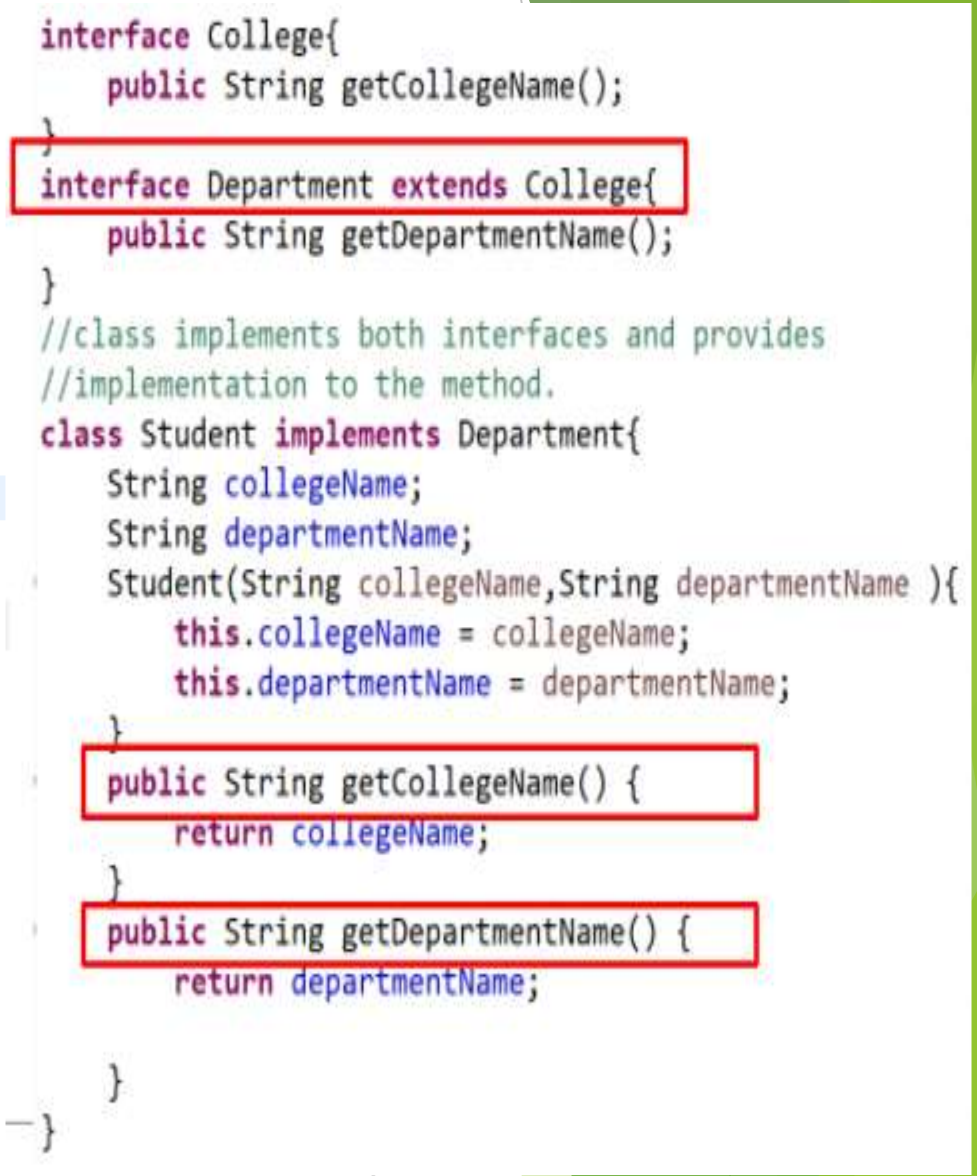
Multiple Inheritance in Java

- Multiple inheritance is not supported through class in java, but it is possible by an interface, why?
-

Extending an Interface Demo



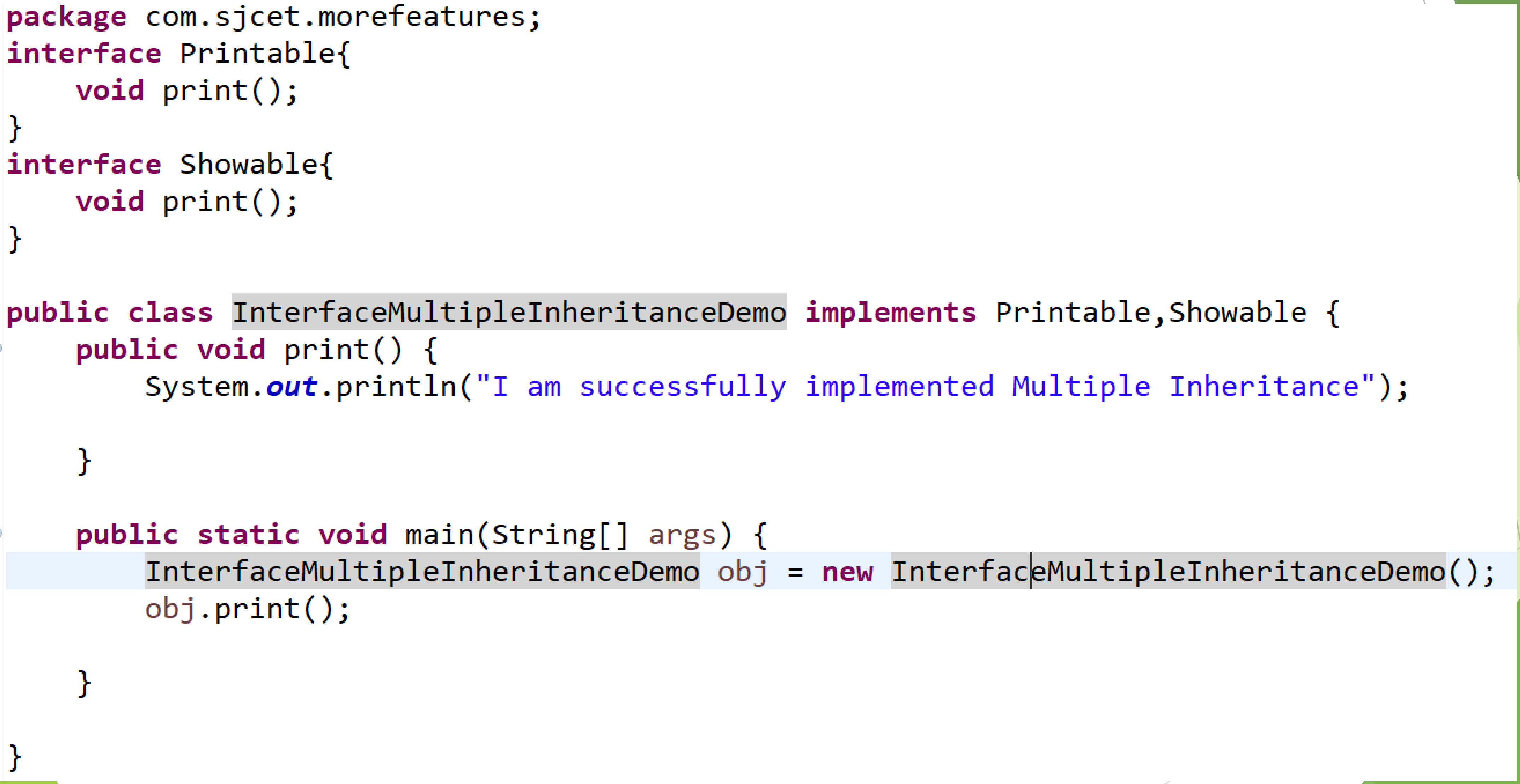
```
public class InterfaceExtendsDemo {

    public static void main(String[] args) {
        Student s1 = new Student("SJCET","CSE");
        // calling the method implemented
        // within the class.
        System.out.println("College Name: "+s1.collegeName);
        System.out.println("Department Name: "+s1.departmentName);

    }

}
```



```
interface College{
    public String getCollegeName();
}
interface Department extends College{
    public String getDepartmentName();
}
//class implements both interfaces and provides
//implementation to the method.
class Student implements Department{
    String collegeName;
    String departmentName;
    Student(String collegeName,String departmentName ){
        this.collegeName = collegeName;
        this.departmentName = departmentName;
    }
    public String getCollegeName() {
        return collegeName;
    }
    public String getDepartmentName() {
        return departmentName;

    }
}
```

getCollegeName() and getDepartmentName() methods are implemented in Student class

Multiple inheritance by Interface in Java



```java
package com.sjcet.morefeatures;
interface Printable{
    void print();
}
interface Showable{
    void print();
}

public class InterfaceMultipleInheritanceDemo implements Printable,Showable {
    public void print() {
        System.out.println("I am successfully implemented Multiple Inheritance");

    }

    public static void main(String[] args) {
        InterfaceMultipleInheritanceDemo obj = new InterfaceMultipleInheritanceDemo();
        obj.print();

    }

}
```

- **Default methods in Java Interfaces**
- With the release of Java 8, we can now add methods with implementation inside an interface.
- These methods are called default methods.

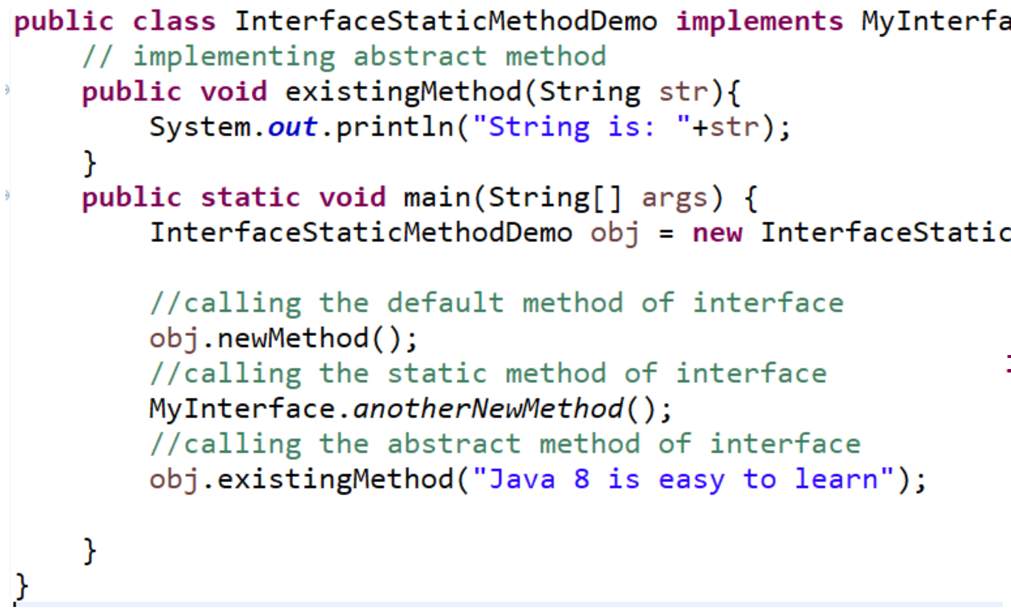
```
public default void getSides() {
// body of getSides()
}
```

-

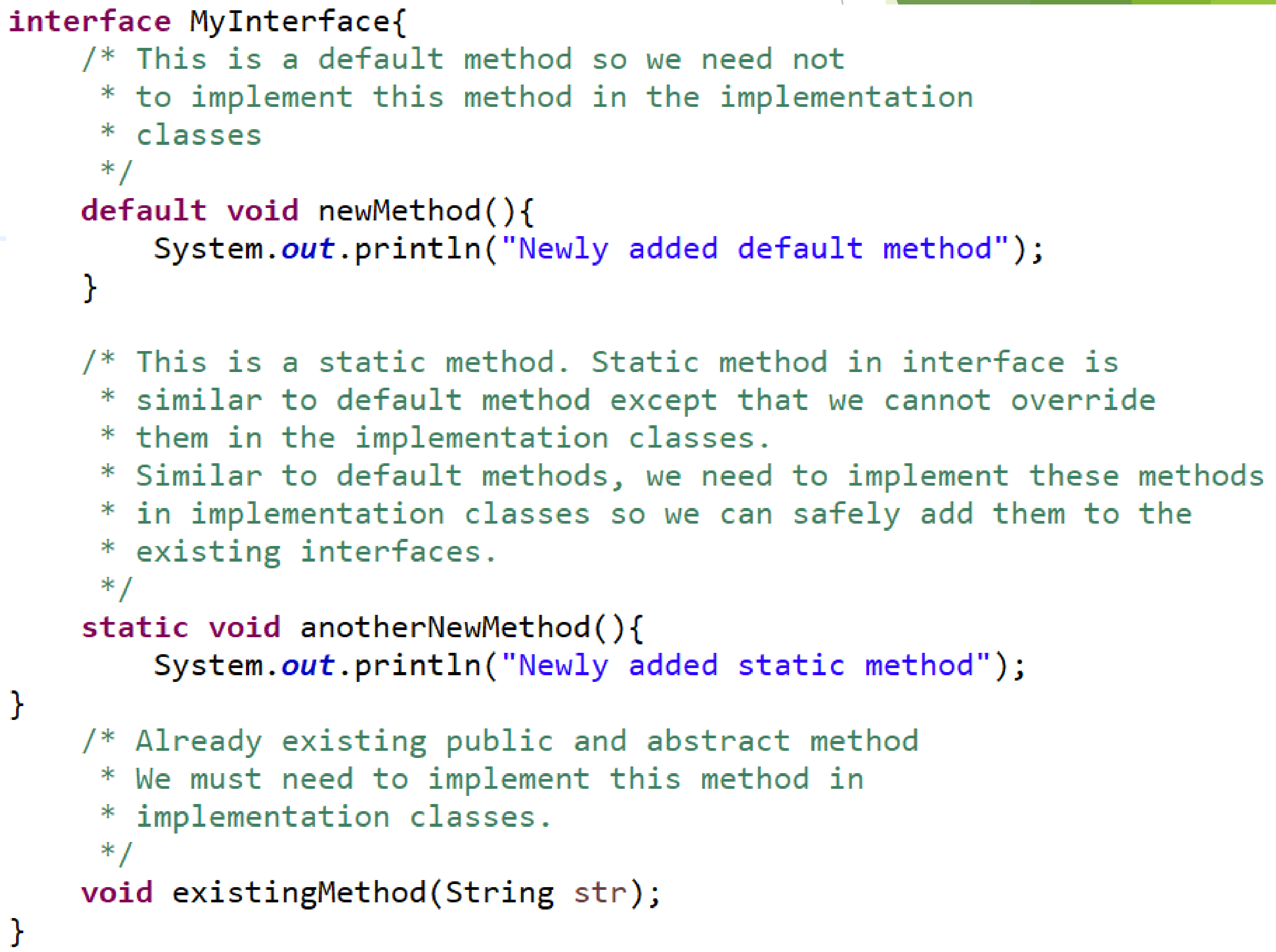
```java
public class InterfaceDefaultMethodDemo {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
            r1.getArea();
            r1.getSides();

            // create an object of Square
            Square s1 = new Square();
            s1.getArea();
            s1.getSides();
    }
}
```

```java
interface Polygon {
      void getArea();
      // default method
      default void getSides() {
        System.out.println("I can get sides of a polygon.");
      }
}
// implements the interface
class Rectangle implements Polygon {
      public void getArea() {
        int length = 6;
        int breadth = 5;
        int area = length * breadth;
        System.out.println("The area of the rectangle is " + area);
      }
      // overrides the getSides()
      public void getSides() {
        System.out.println("I have 4 sides.");
      }
}
    // implements the interface
    class Square implements Polygon {
      public void getArea() {
        int length = 5;
        int area = length * length;
        System.out.println("The area of the square is " + area);
      }
    }
```

- **Static method in Interface**
- static methods in interface are similar to default method so we need not to implement them in the implementation classes.
- We can safely add them to the existing interfaces without changing the code in the implementation classes



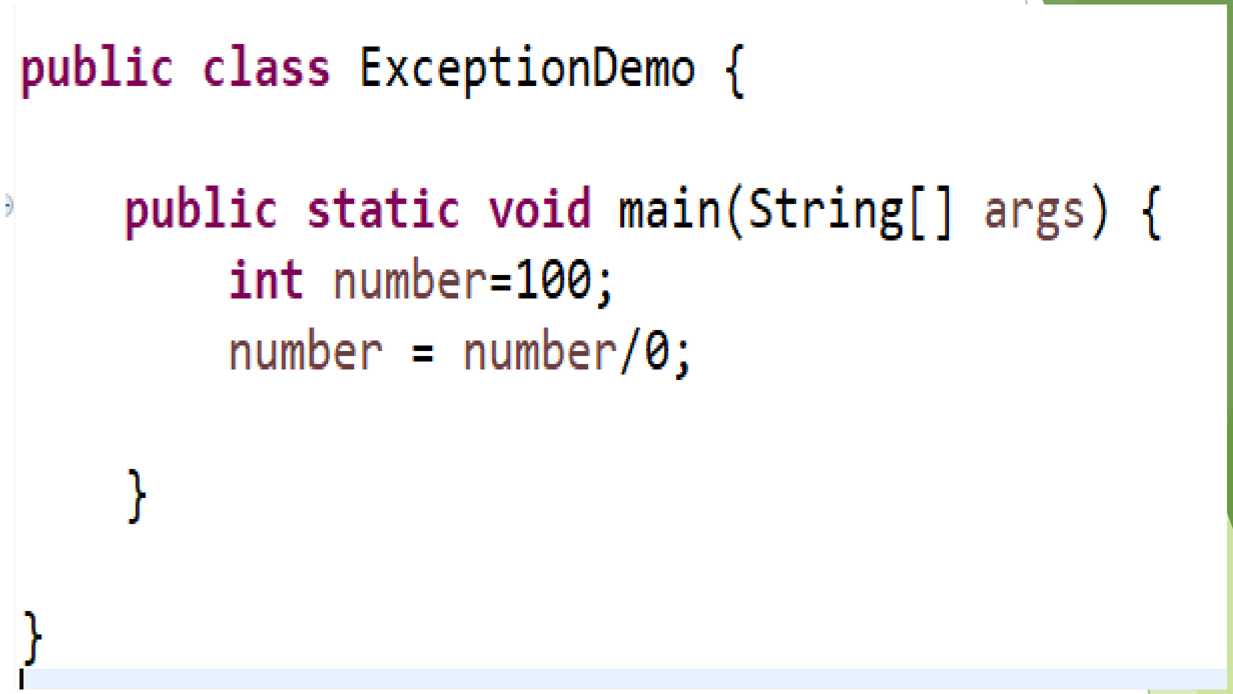
```java
public class InterfaceStaticMethodDemo implements MyInterface {
    // implementing abstract method
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }
    public static void main(String[] args) {
        InterfaceStaticMethodDemo obj = new InterfaceStaticMethodDemo();

        //calling the default method of interface
        obj.newMethod();
        //calling the static method of interface
        MyInterface.anotherNewMethod();
        //calling the abstract method of interface
        obj.existingMethod("Java 8 is easy to learn");

    }
}
```



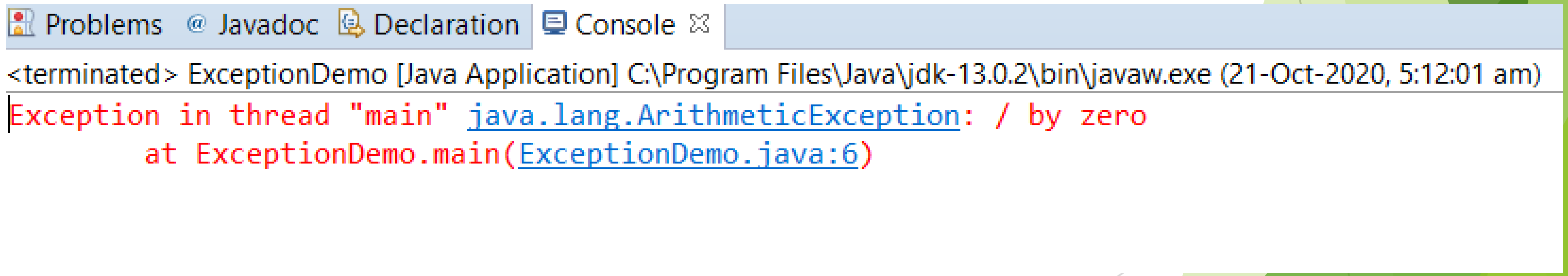
```java
interface MyInterface{
    /* This is a default method so we need not
     * to implement this method in the implementation
     * classes
     */
    default void newMethod(){
        System.out.println("Newly added default method");
    }

    /* This is a static method. Static method in interface is
     * similar to default method except that we cannot override
     * them in the implementation classes.
     * Similar to default methods, we need to implement these methods
     * in implementation classes so we can safely add them to the
     * existing interfaces.
     */
    static void anotherNewMethod(){
        System.out.println("Newly added static method");
}
    /* Already existing public and abstract method
     * We must need to implement this method in
     * implementation classes.
     */
    void existingMethod(String str);
}
```

Exception Handling

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- When an exception occurs program execution gets terminated.
- In such cases we get a system generated error message.
- The good thing about exceptions is that they can be handled in Java.



```
public class ExceptionDemo {

    public static void main(String[] args) {
        int number=100;
        number = number/0;

    }

}
```



```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:12:01 am)
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

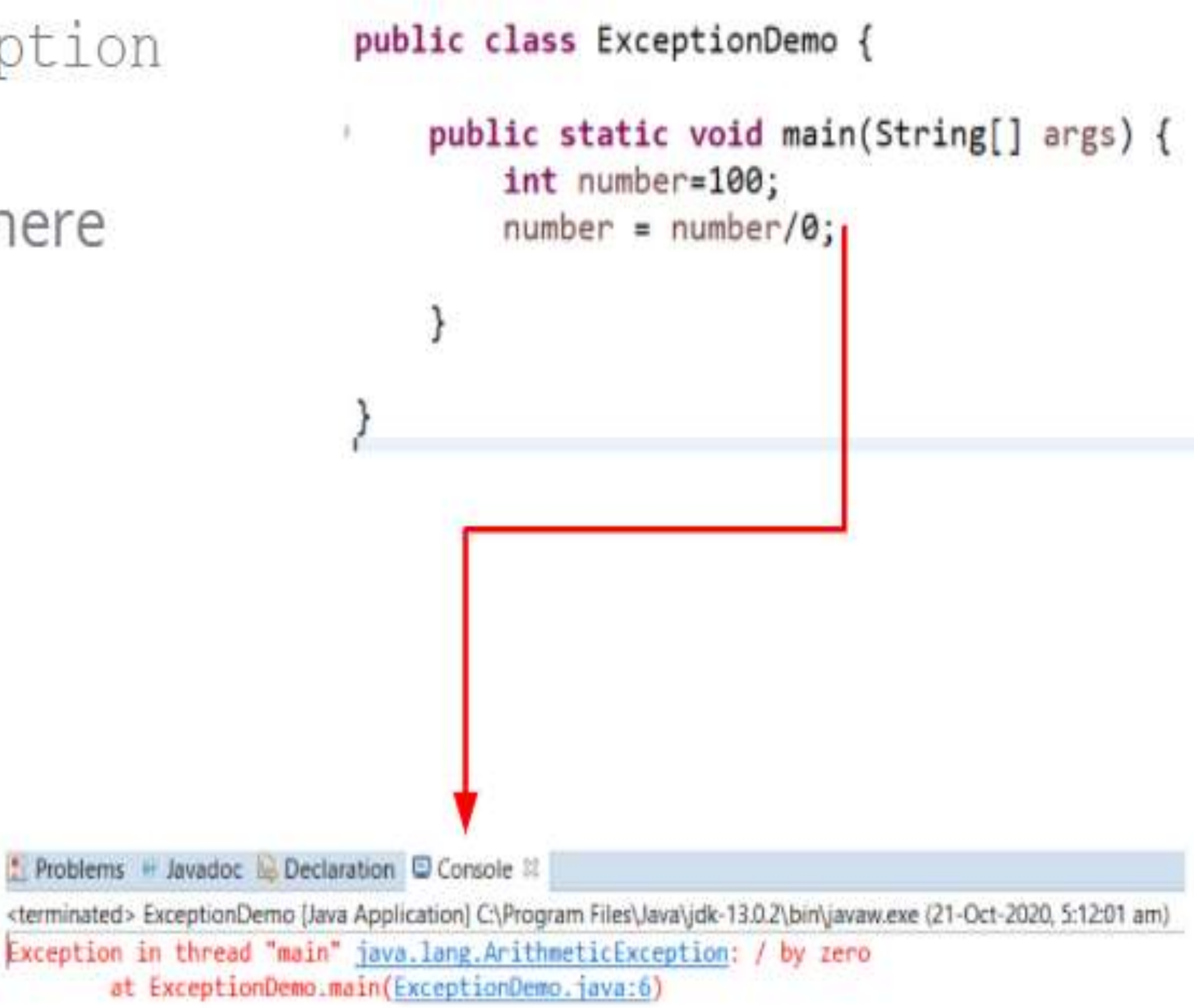
- **Why an exception occurs?**
- There can be several reasons that can cause a program to throw exception.
- For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.
- **Difference between error and exception**
- **Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.
- **Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

Common scenarios where exceptions may occur

Scenario where `ArithmeticException` occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

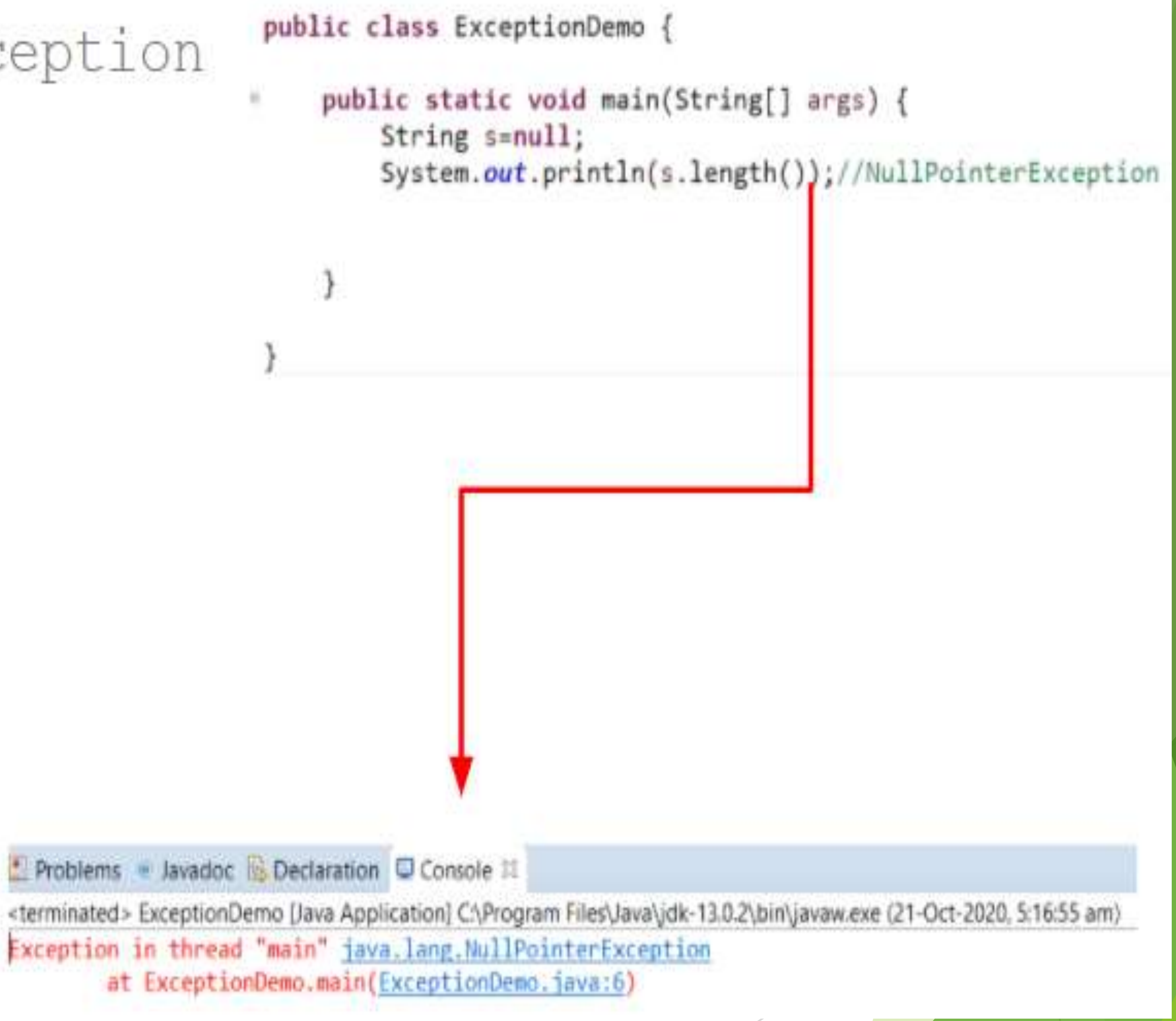


```
public class ExceptionDemo {

    public static void main(String[] args) {
        int number=100;
        number = number/0;

    }

}
```

```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:12:01 am)
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

- Scenario where `NullPointerException` occurs

```
String s=null;
System.out.println(s.length());
```

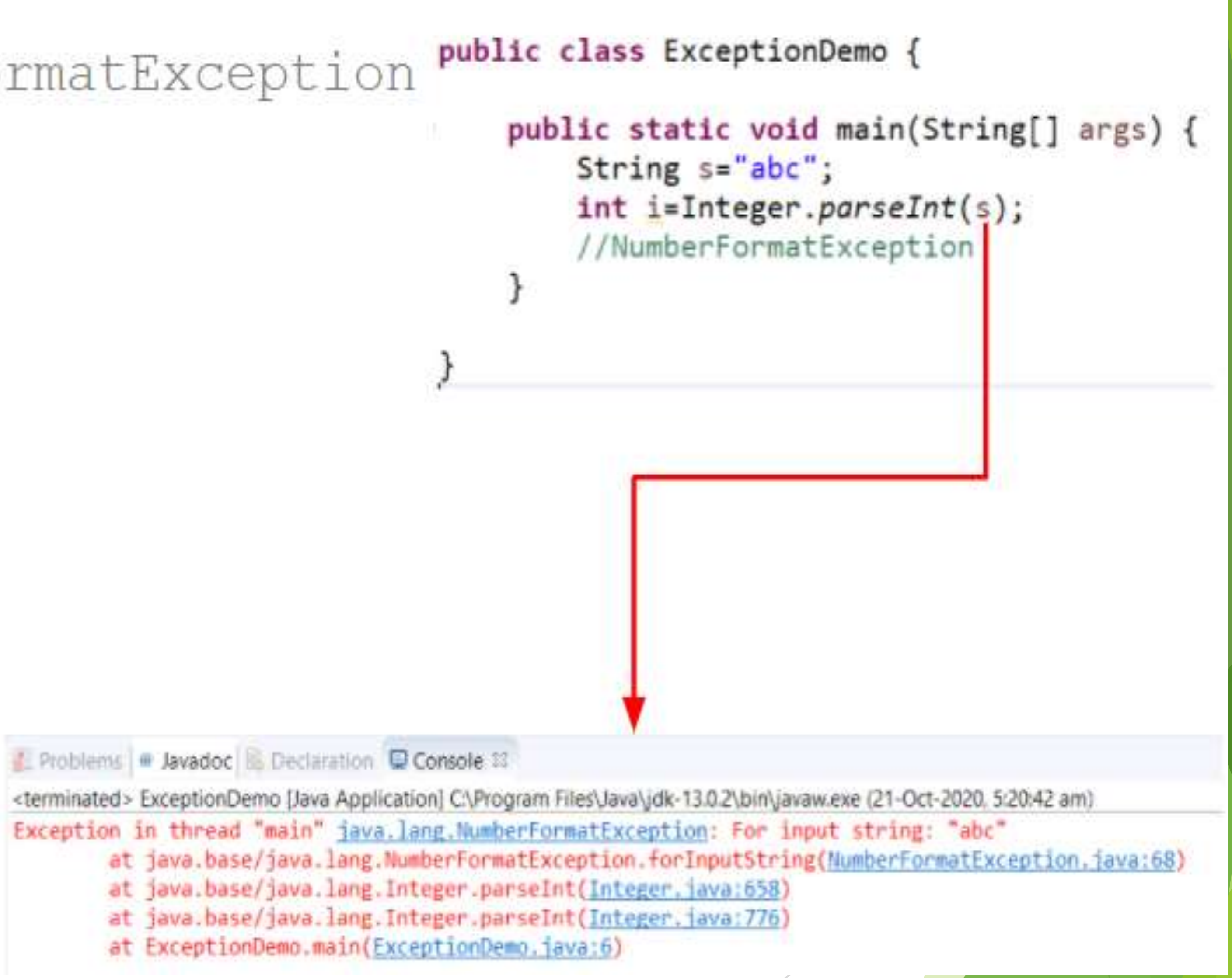


```
public class ExceptionDemo {

    public static void main(String[] args) {
        String s=null;
        System.out.println(s.length());//NullPointerException

    }
}
```

```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:16:55 am)
Exception in thread "main" java.lang.NullPointerException
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

- Scenario where `NumberFormatException` occurs

```
String s="abc";
int i=Integer.parseInt(s);
```



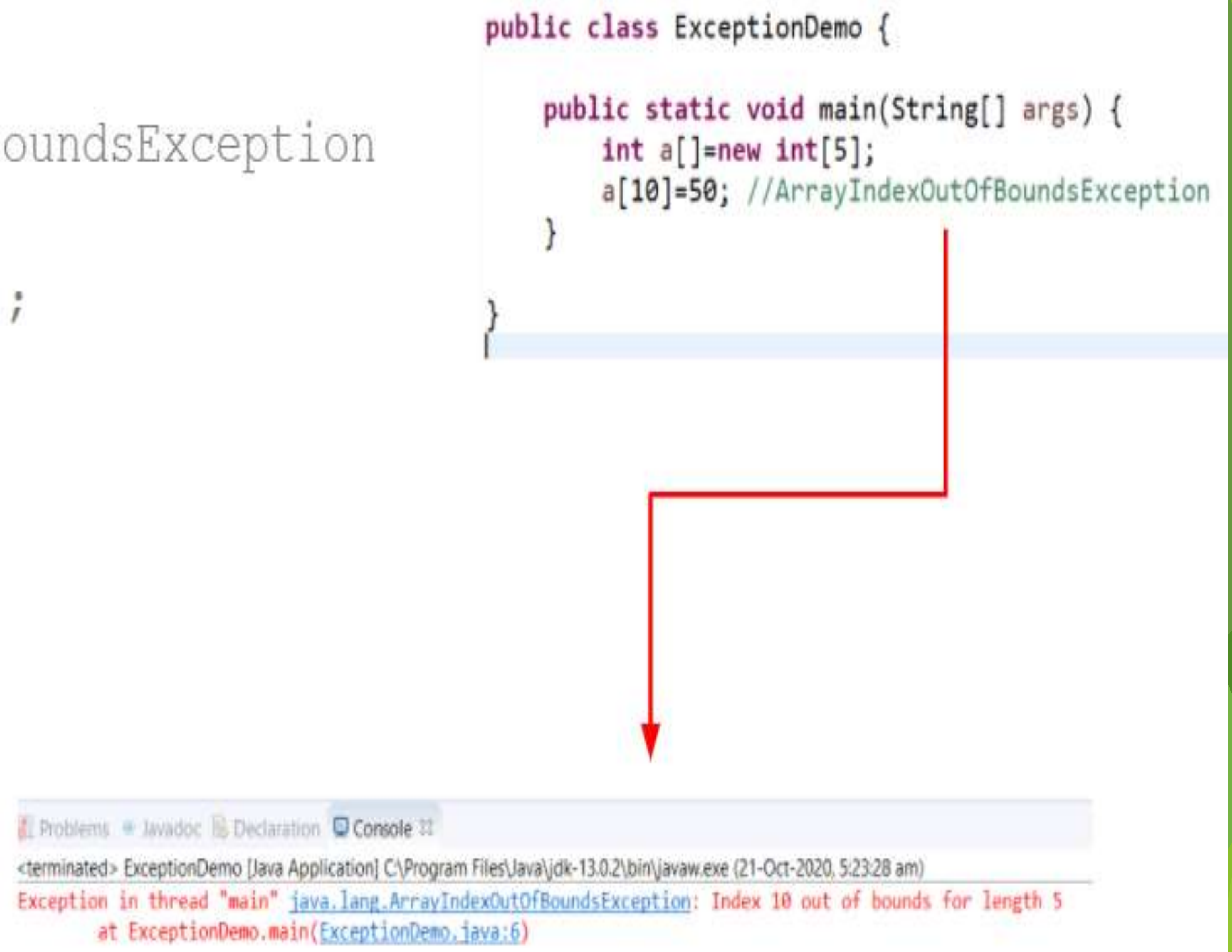
```
public class ExceptionDemo {

    public static void main(String[] args) {
        String s="abc";
        int i=Integer.parseInt(s);
        //NumberFormatException
    }

}
```

Problems | Javadoc | Declaration | Console

```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:20:42 am)
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
	at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:68)
	at java.base/java.lang.Integer.parseInt(Integer.java:658)
	at java.base/java.lang.Integer.parseInt(Integer.java:776)
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

- Scenario where `ArrayIndexOutOfBoundsException` occurs

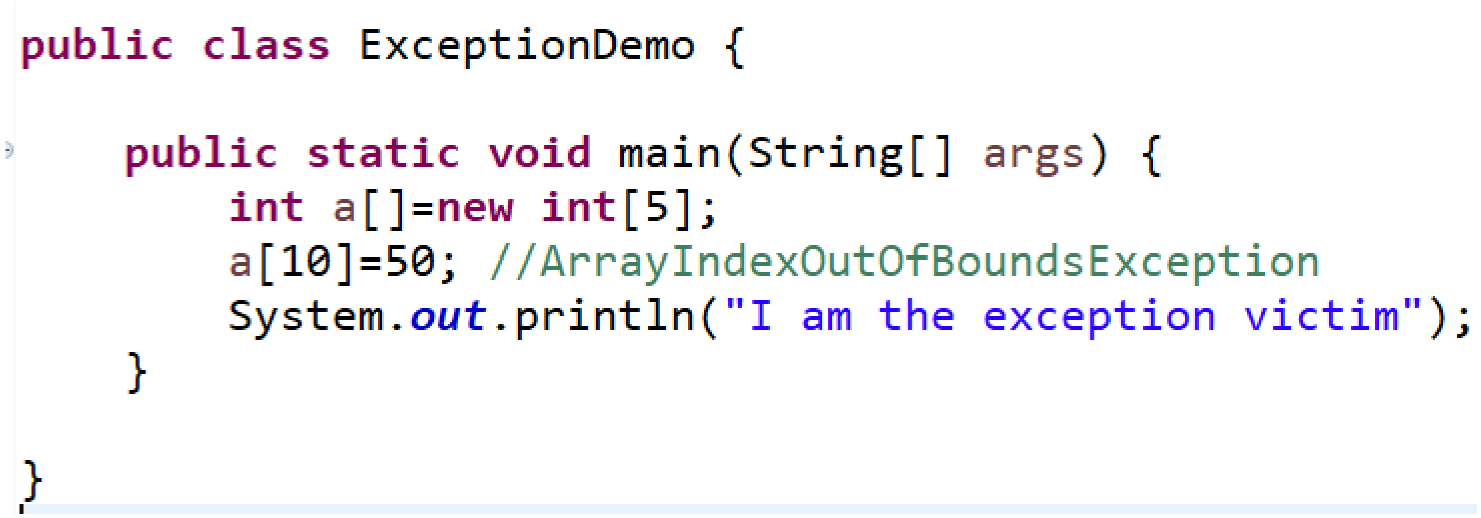
```
int a[]=new int[5];
a[10]=50;
```



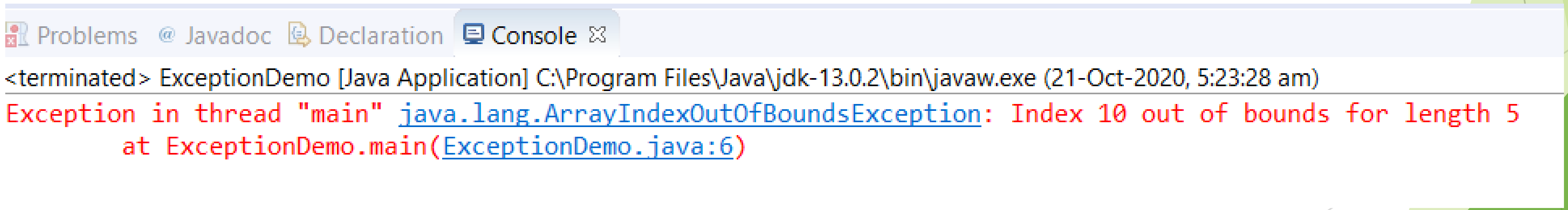
```
public class ExceptionDemo {

    public static void main(String[] args) {
        int a[]=new int[5];
        a[10]=50; //ArrayIndexOutOfBoundsException
    }

}
```

```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:23:28 am)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

- **What is exception handling?**
- Exception Handling is a mechanism to handle exceptions.
- Exception handling ensures that the flow of the program doesn't break when an exception occurs.



```java
public class ExceptionDemo {

    public static void main(String[] args) {
        int a[]=new int[5];
        a[10]=50; //ArrayIndexOutOfBoundsException
        System.out.println("I am the exception victim");
    }

}
```



```
<terminated> ExceptionDemo [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (21-Oct-2020, 5:23:28 am)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5
	at ExceptionDemo.main(ExceptionDemo.java:6)
```

- **Exception Handling Methods**
- Java provides various methods to handle the Exceptions like:
  - Try
  - catch
  - finally
  - throw
  - throws

Checked Exceptions

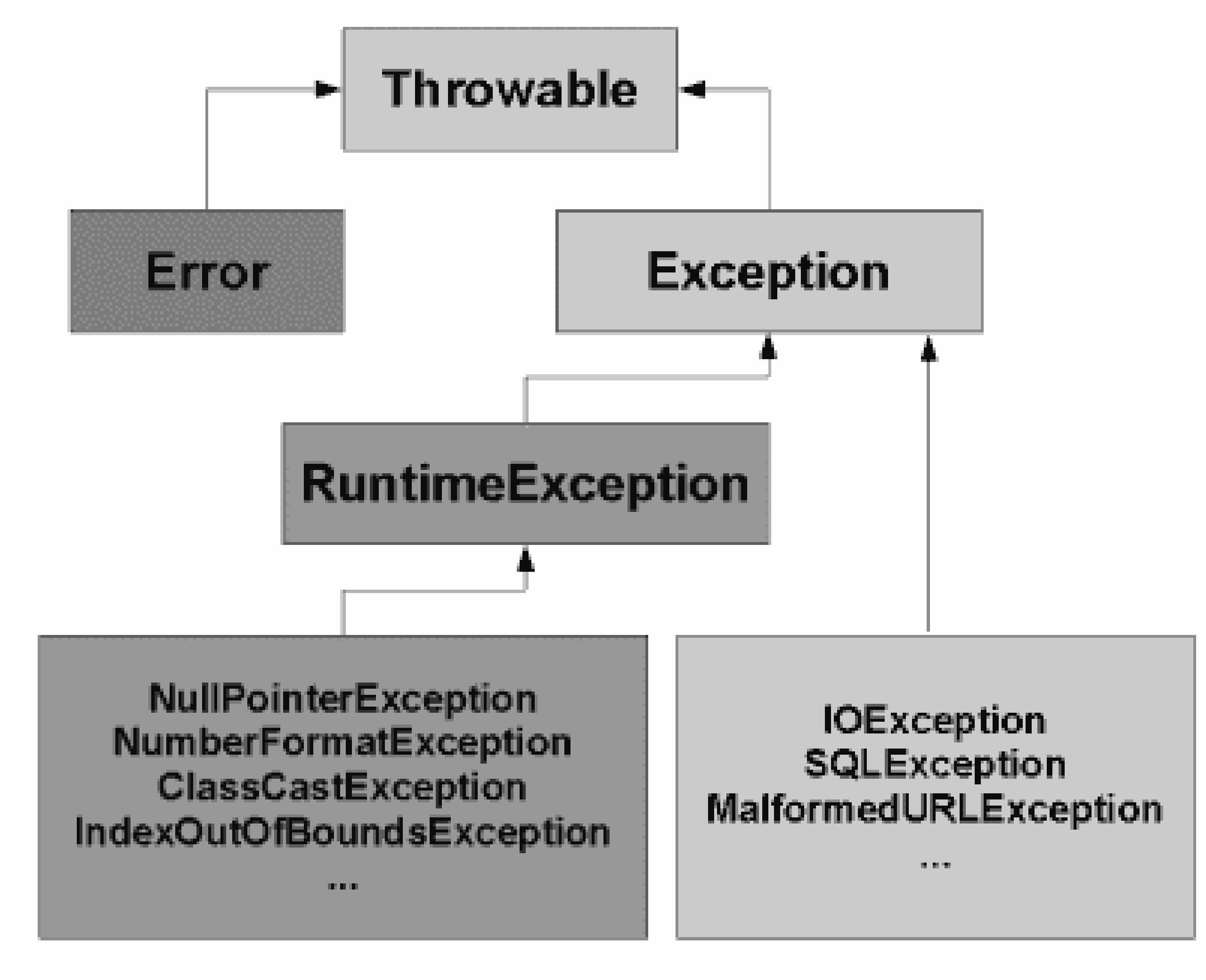
- Checked exceptions are checked at compile-time, , these are also called as compile time exceptions.
- These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
-

```java
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

```
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException;
   FileReader fr = new FileReader(file);
      ^
1 error
```

Unchecked Exceptions

- An unchecked exception is an exception that occurs at the time of execution.
- These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.


Throwable
Error
Exception
RuntimeException
NullPointerException
NumberFormatException
ClassCastException
IndexOutOfBoundsException
...
IOException
SQLException
MalformedURLException
...

try Block

- The try block contains a set of statements where an exception can occur.
- It is always followed by a catch block, which handles the exception that occurs in the associated try block.
- A try block must be followed by catch blocks or finally block or both.

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

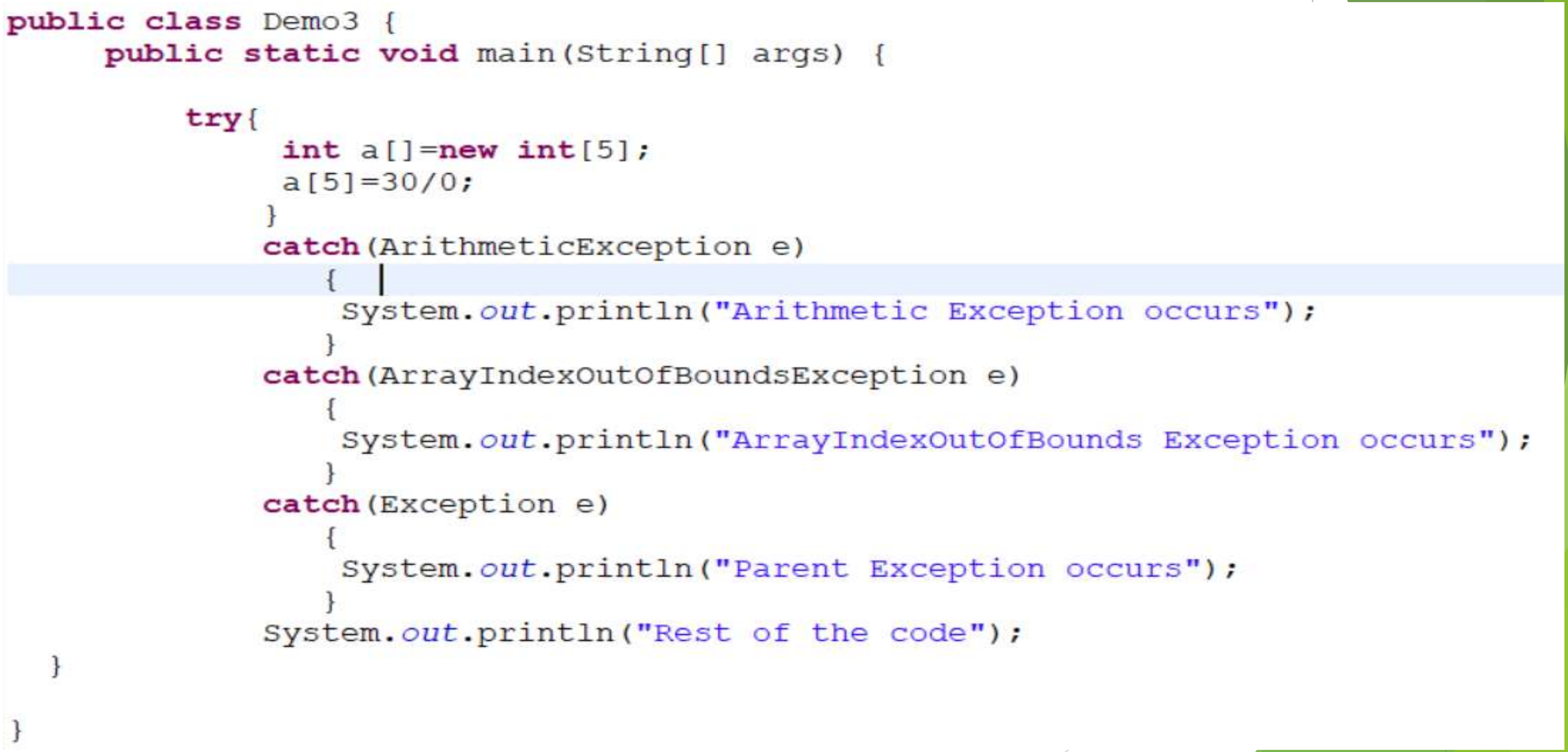
Catch block

- A catch block is where you handle the exceptions.
- This block must follow the try block and a single try block can have several catch blocks associated with it.
- You can catch different exceptions in different catch blocks.
- When an exception occurs in a try block, the corresponding catch block that handles that particular exception executes.

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

Multiple catch Clauses

- A single try block can have multiple catch blocks associated with it.



```
public class Demo3 {
     public static void main(String[] args) {

          try{
               int a[]=new int[5];
               a[5]=30/0;
              }
              catch(ArithmeticException e)
                 {
                  System.out.println("Arithmetic Exception occurs");
                 }
              catch(ArrayIndexOutOfBoundsException e)
                 {
                  System.out.println("ArrayIndexOutOfBounds Exception occurs");
                 }
              catch(Exception e)
                 {
                  System.out.println("Parent Exception occurs");
                 }
              System.out.println("Rest of the code");
   }

}
```

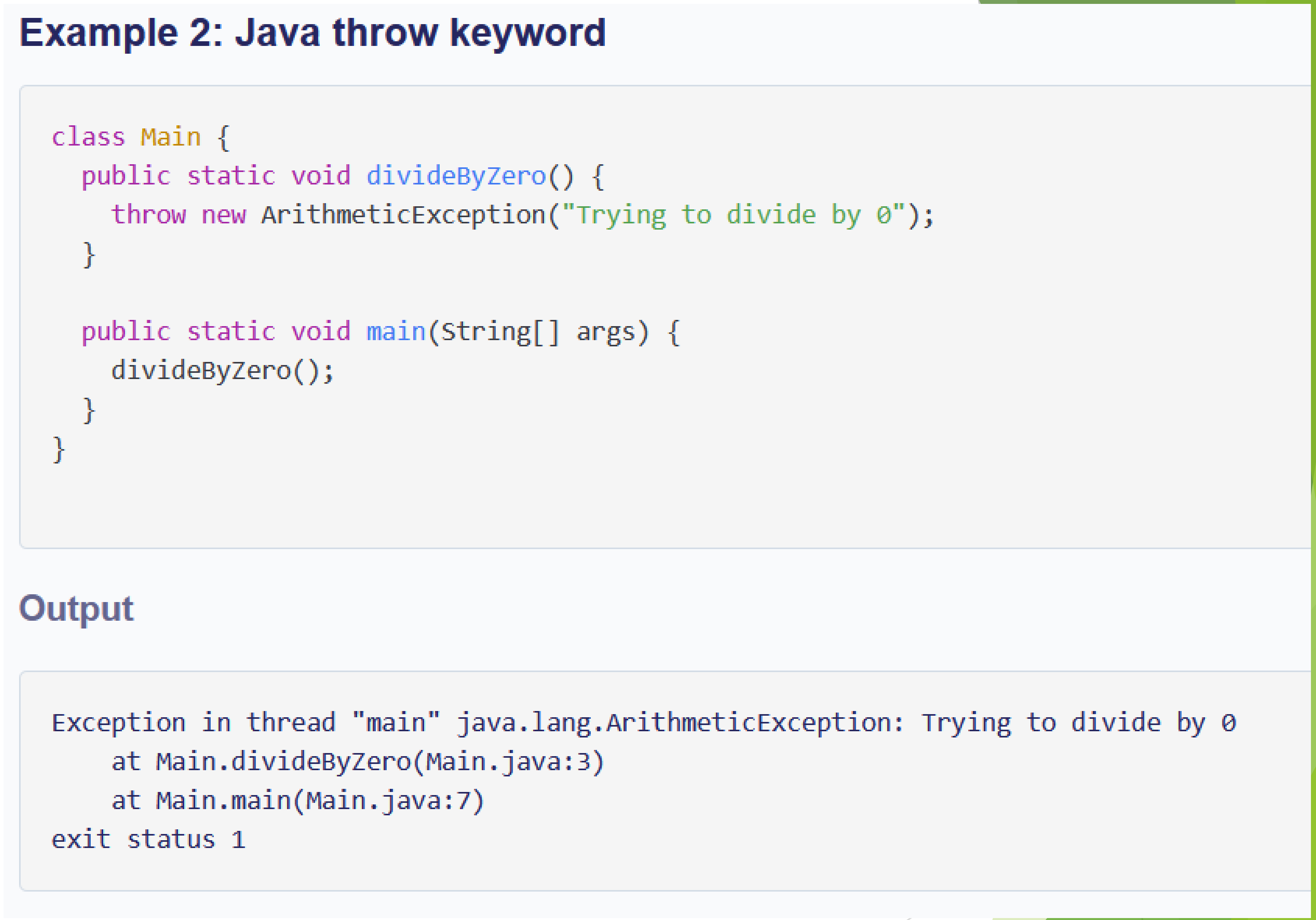
Nested try Statements

- When a try catch block is present in another try block then it is called the nested try catch block.

```
//Main try block
try {
    statement 1;
    statement 2;
    //try-catch block inside another try block
    try {
        statement 3;
        statement 4;
        //try-catch block inside nested try block
        try {
            statement 5;
            statement 6;
        }
        catch(Exception e2) {
            //Exception Message
        }
    }
    catch(Exception e1) {
         //Exception Message
    }

}
//Catch of Main(parent) try block
catch(Exception e3) {
        //Exception Message
}
```

Throw

- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.



Example 2: Java throw keyword

```
class Main {
  public static void divideByZero() {
    throw new ArithmeticException("Trying to divide by 0");
  }

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

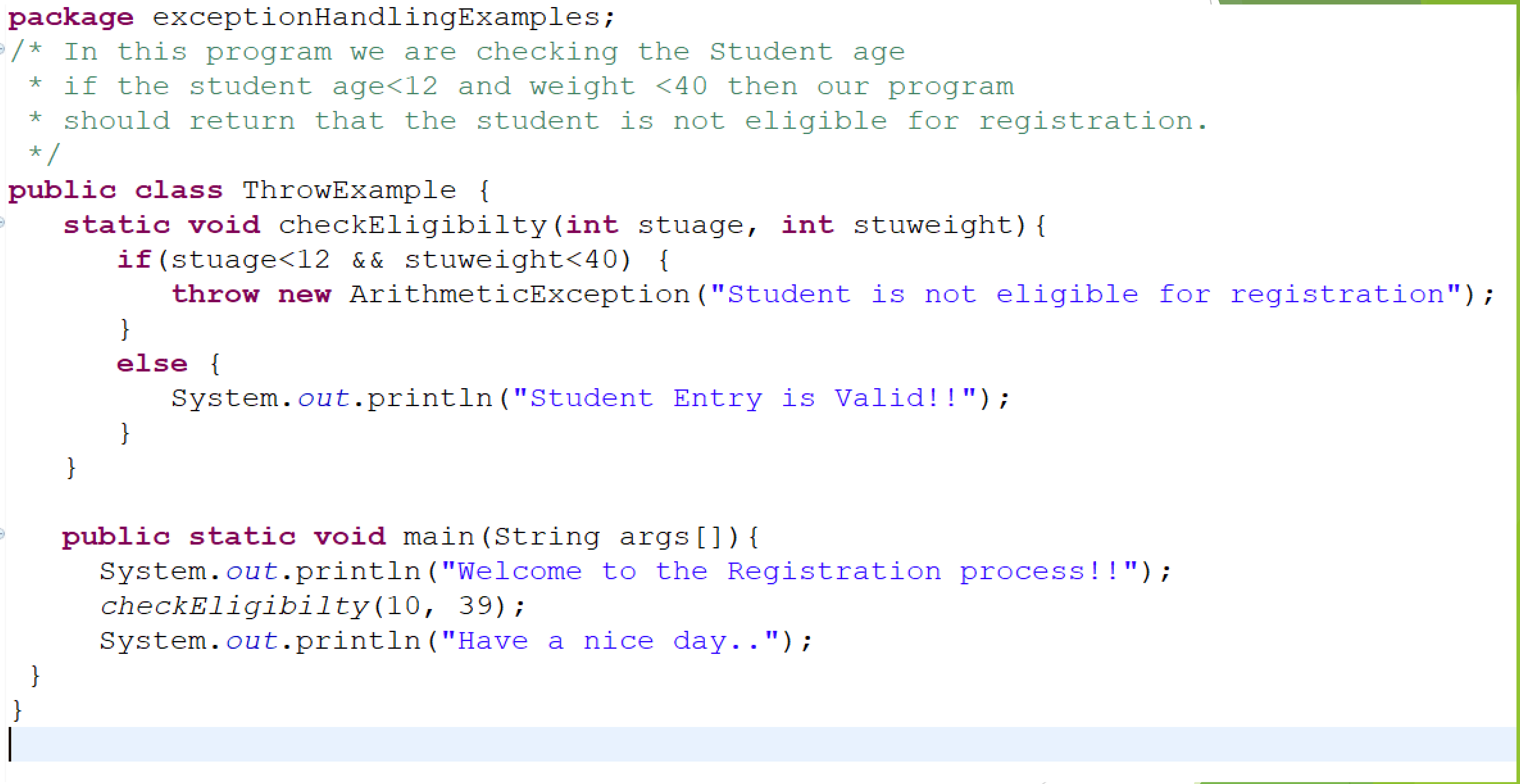
Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
    at Main.divideByZero(Main.java:3)
    at Main.main(Main.java:7)
exit status 1
```

```
Syntax :
throw ThrowableInstance
```

- We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.

- **Problem Statement**
- Let’s say we have a requirement where we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an ArithmeticException with the warning message “Student is not eligible for registration”.
-



```java
package exceptionHandlingExamples;
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program
 * should return that the student is not eligible for registration.
 */
public class ThrowExample {
   static void checkEligibilty(int stuage, int stuweight){
      if(stuage<12 && stuweight<40) {
         throw new ArithmeticException("Student is not eligible for registration");
      }
      else {
         System.out.println("Student Entry is Valid!!");
      }
   }

   public static void main(String args[]){
     System.out.println("Welcome to the Registration process!!");
     checkEligibilty(10, 39);
     System.out.println("Have a nice day..");
 }
}
```

throws

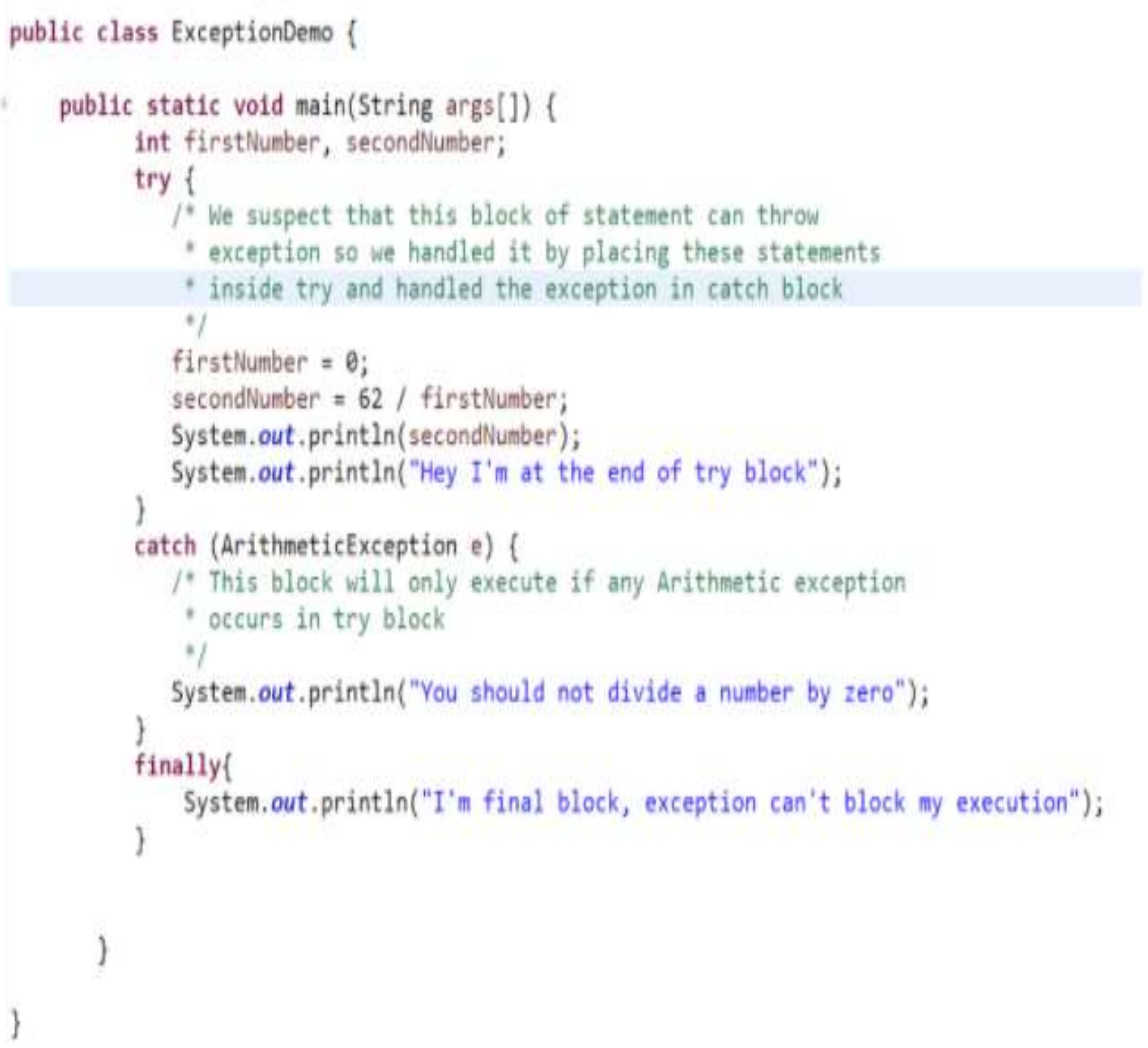
- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions.
- The caller to these methods has to handle the exception using a try-catch block.

```
public void myMethod() throws ArithmeticException,
NullPointerException
{
  // Statements that might throw an exception
}

public static void main(String args[]) {
  try {
    myMethod();
  }
  catch (ArithmeticException e) {
    // Exception handling statements
  }
  catch (NullPointerException e) {
    // Exception handling statements
  }
}
```

finally

- A finally block contains all the crucial statements that must be executed whether an exception occurs or not.
- The statements present in this block will always execute, regardless an exception occurs in the try block or not such as closing a connection, stream etc.

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// optional
finally {
// block of code to be executed after try block ends
}
```

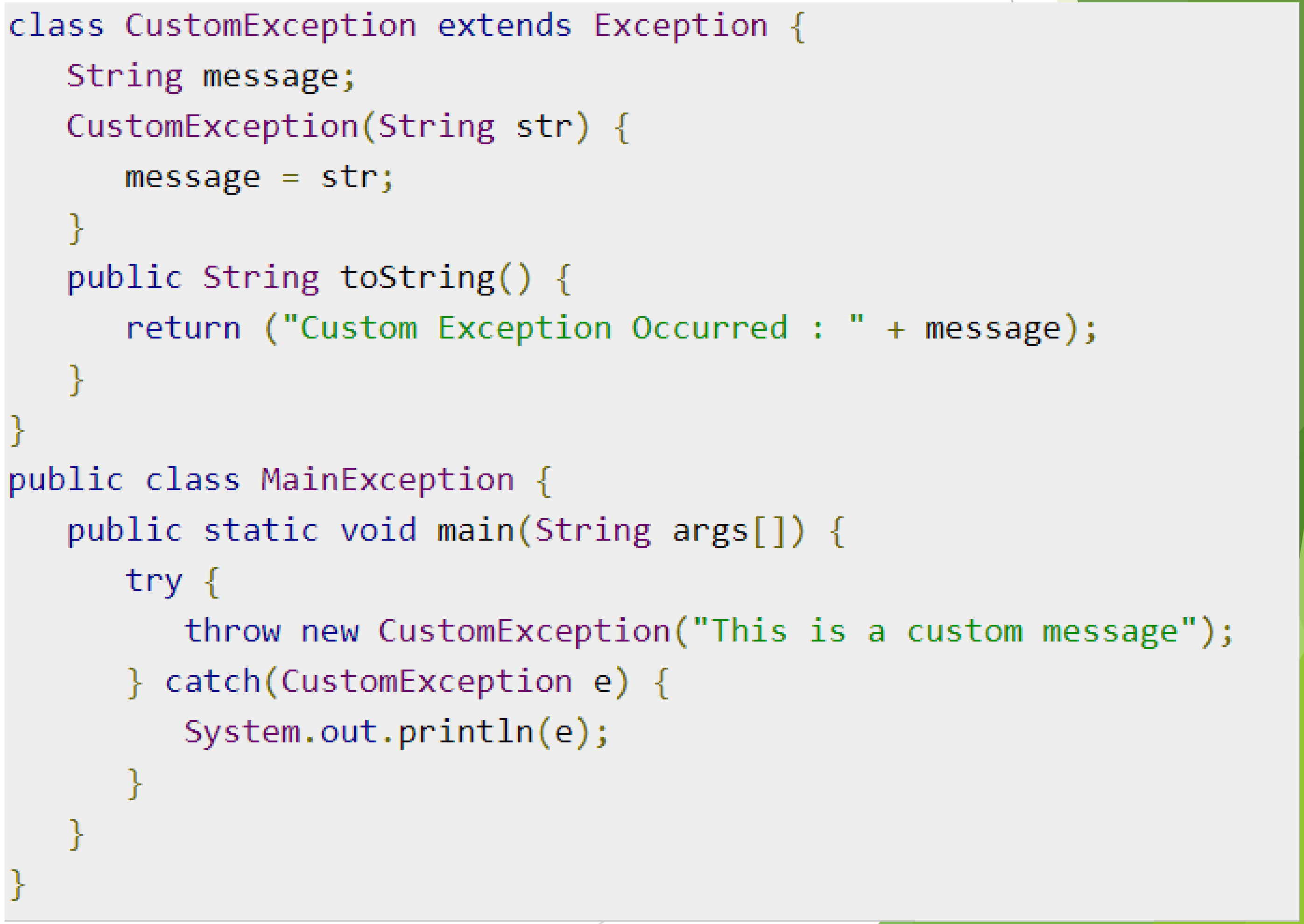


```java
public class ExceptionDemo {

    public static void main(String args[]) {
          int firstNumber, secondNumber;
          try {
             /* We suspect that this block of statement can throw
              * exception so we handled it by placing these statements
              * inside try and handled the exception in catch block
              */
             firstNumber = 0;
             secondNumber = 62 / firstNumber;
             System.out.println(secondNumber);
             System.out.println("Hey I'm at the end of try block");
          }
          catch (ArithmeticException e) {
             /* This block will only execute if any Arithmetic exception
              * occurs in try block
              */
             System.out.println("You should not divide a number by zero");
          }
          finally{
              System.out.println("I'm final block, exception can't block my execution");
          }


    }

}
```

Output

```
You should not divide a number by zero
I'm final block, exception can't block my
execution
```

- **How to create custom exceptions in Java**
- Sometimes it is required to develop meaningful exceptions based on the application requirements.
- We can create our own exceptions by extending Exception class in Java



```java
class CustomException extends Exception {
   String message;
   CustomException(String str) {
      message = str;
   }
   public String toString() {
      return ("Custom Exception Occurred : " + message);
   }
}
public class MainException {
   public static void main(String args[]) {
      try {
         throw new CustomException("This is a custom message");
      } catch(CustomException e) {
         System.out.println(e);
      }
   }
}
```

Input/Output & IO basics

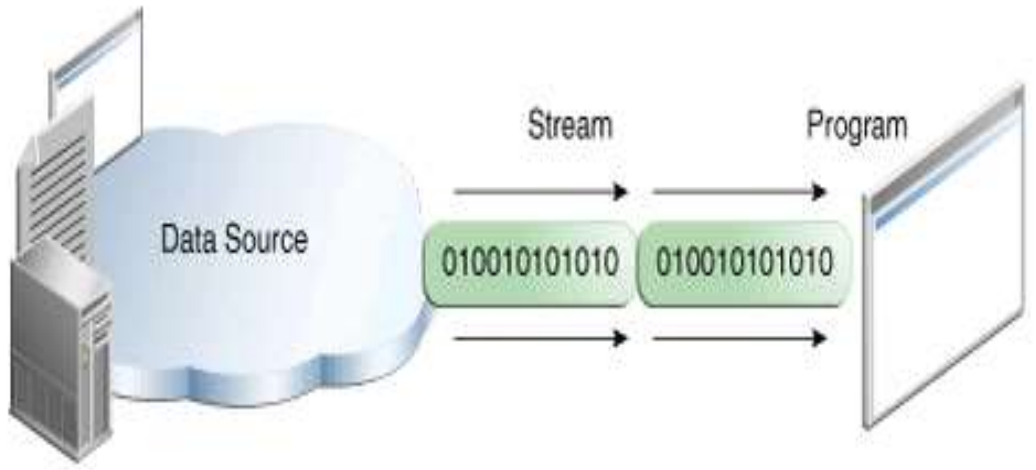
- **Stream Classes in Java**
- A stream is a communication channel that a program has with the outside world. It is used to transfer data items in succession.
- An I/O Stream represents an input source or an output destination.
- A stream is a sequence of data.

- Based on the data they handle there are two types of streams

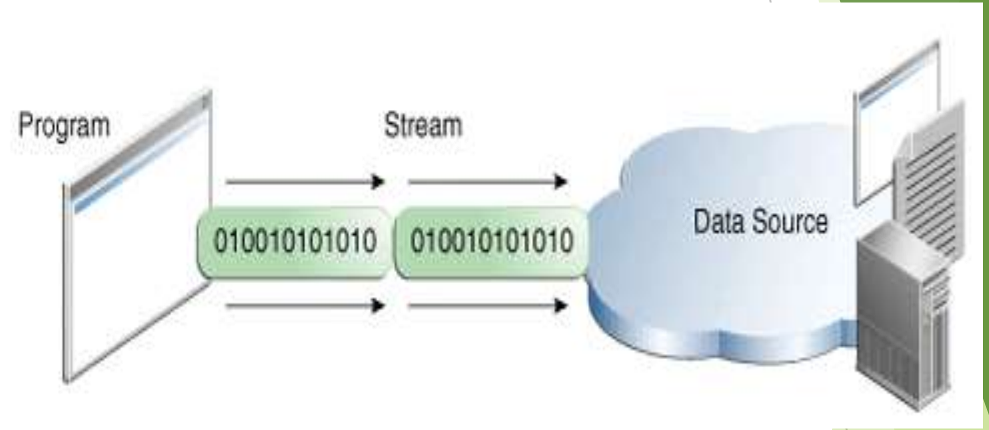
- **Byte Streams** − These handle data in bytes (8 bits)
  - Using these you can store characters, videos, audios, images etc.

- **Character Streams** − These handle data in 16 bit Unicode. Using these you can read and write text data only.

- **InputStream** − A program uses an input stream to read data from a source, one item at a time
- **OutputStream** − A program uses an output stream to write data to a destination, one item at time:

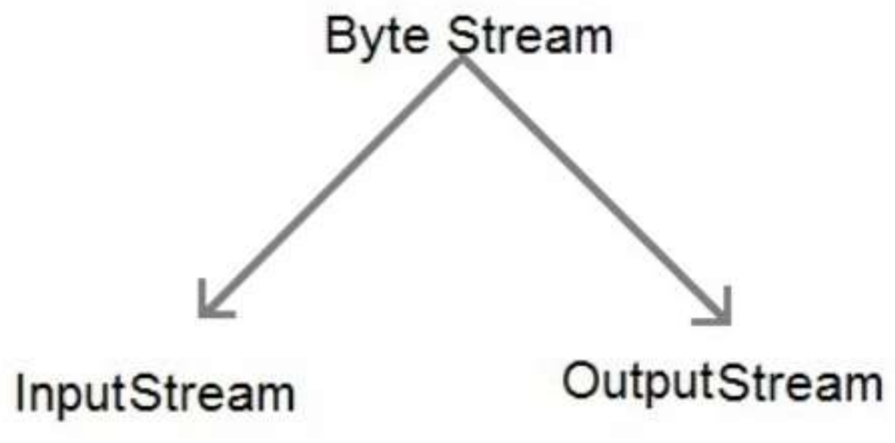


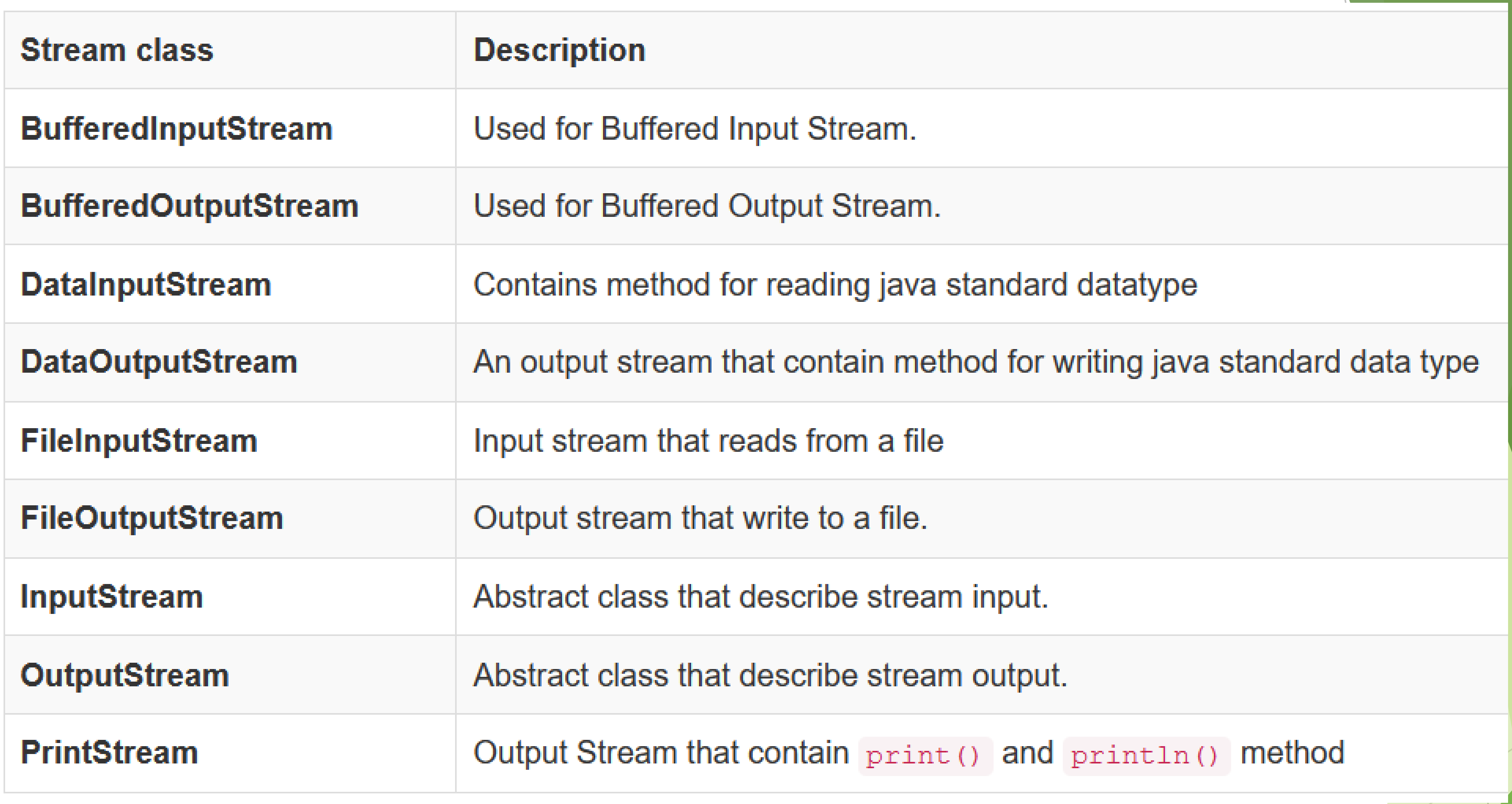
Reading information into a program



Writing information from a program

- **Java Byte Stream Classes**
- Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.
- These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.
-



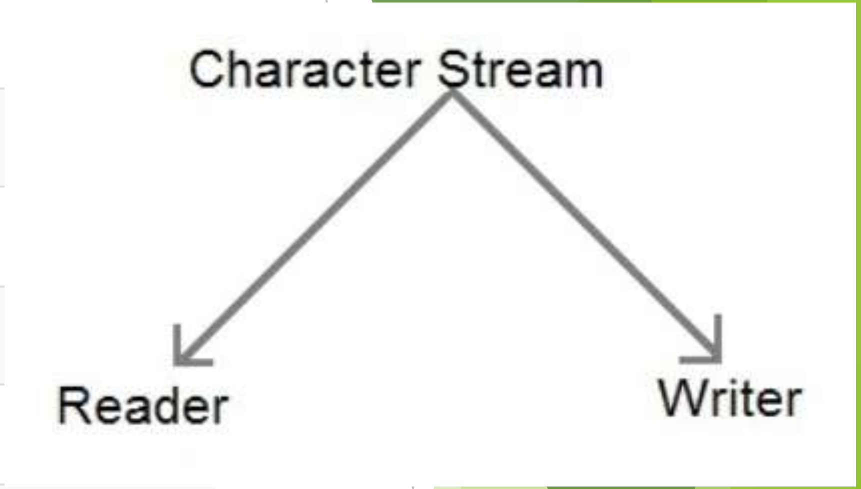


Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

- These classes define several key methods. Two most important are
  - read() : reads byte of data
  - write() : Writes byte of data.

- **Java Character Stream Classes**
- Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.

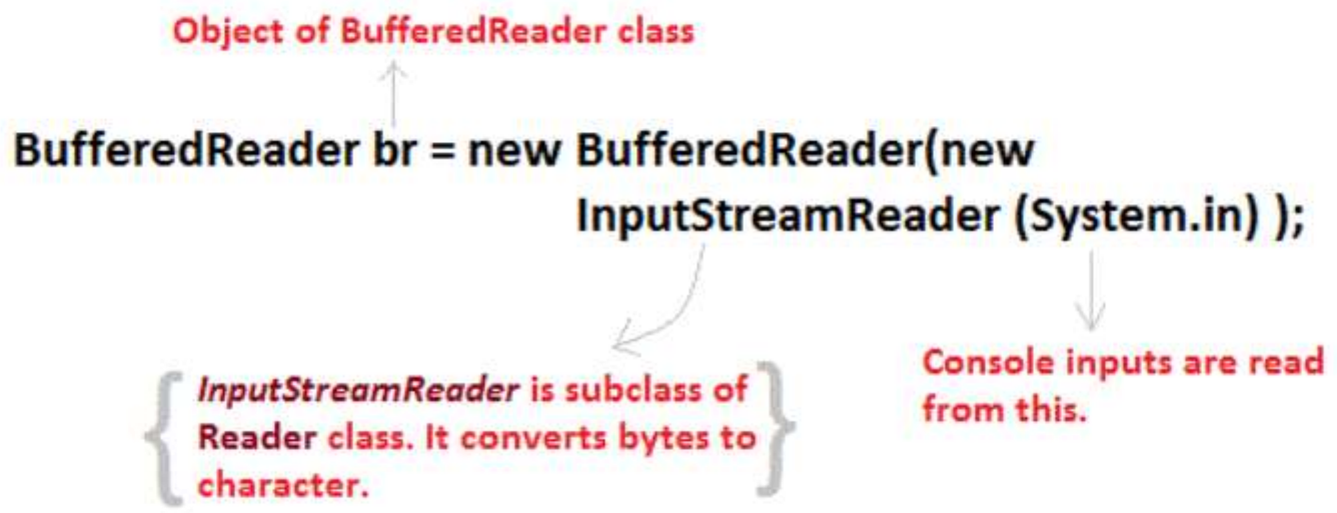
Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output


$$\begin{tikzcd}
 & \text{Character Stream} \arrow[dl] \arrow[dr] & \\
\text{Reader} & & \text{Writer}
\end{tikzcd}$$

- **The Predefined Streams**
- All Java programs automatically import the java.lang package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system.
  - long millis=System. currentTimeMillis();
- System also contains three predefined stream variables: **in**, **out**, and **err**.
- These fields are declared as **public**, **static**, and **final** within System.
- This means that they can be used by any other part of your program and without reference to a specific System object

Reading Console Input

- We use the object of BufferedReader class to take inputs from the keyboard.



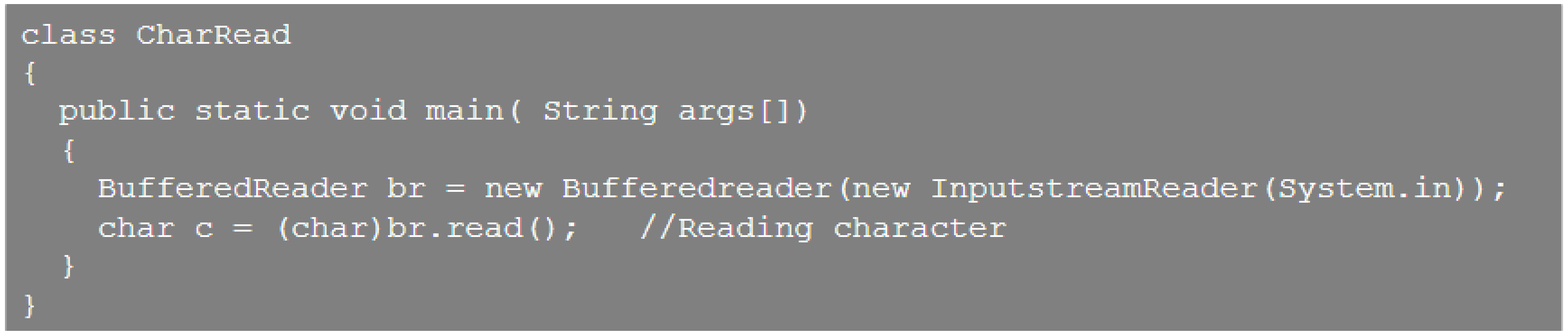
```
BufferedReader br = new BufferedReader(new
                              InputStreamReader (System.in) );
```

Object of BufferedReader class
InputStreamReader is subclass of Reader class. It converts bytes to character.
Console inputs are read from this.

- **Reading Characters**

- read() method is used with BufferedReader object to read characters.
- As this function returns integer type value has *we need to use typecasting* to convert it into char type.

  int read() throws IOException

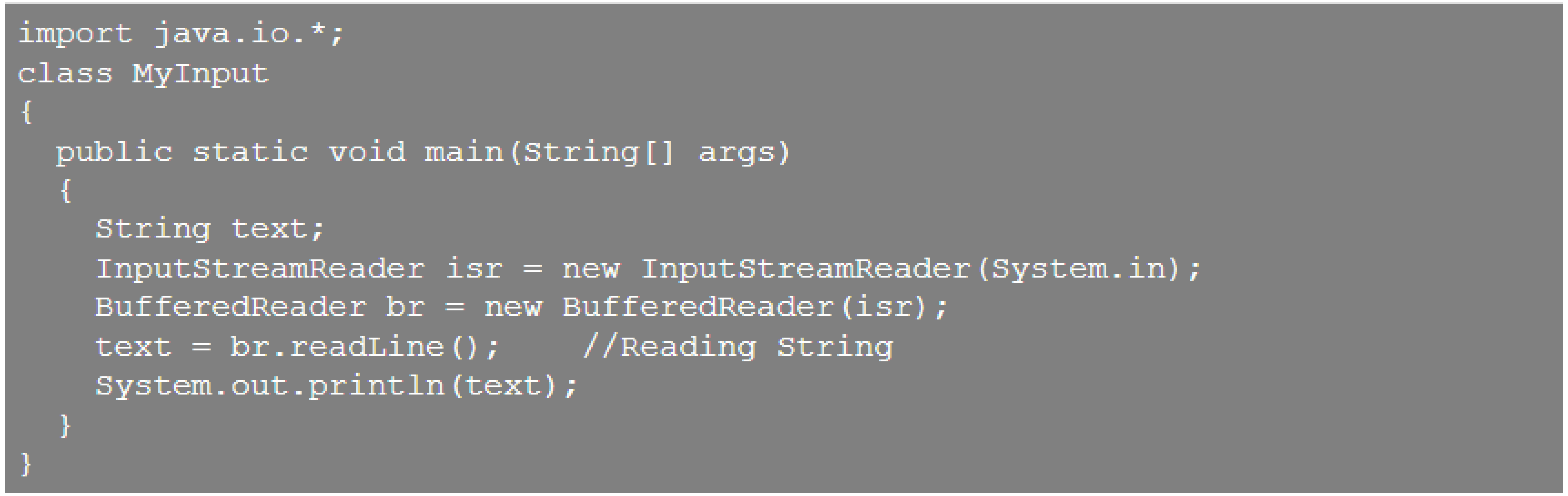


```
class CharRead
{
  public static void main( String args[])
  {
    BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
    char c = (char)br.read();    //Reading character
  }
}
```

- **Reading Strings in Java**
- To read string we have to use readLine() function with BufferedReader class's object.

  String readLine() throws IOException

-

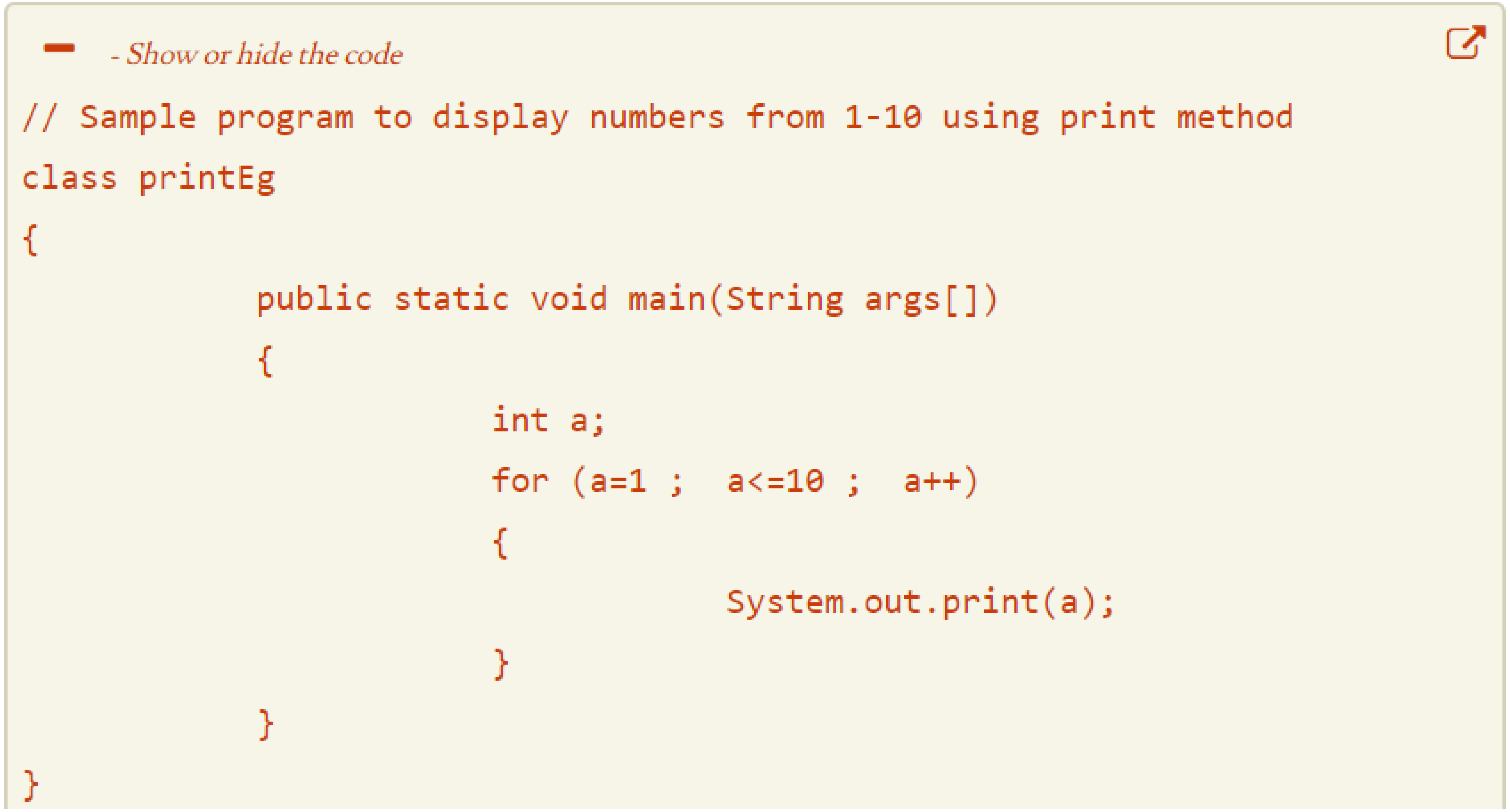


```
import java.io.*;
class MyInput
{
  public static void main(String[] args)
  {
    String text;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    text = br.readLine();    //Reading String
    System.out.println(text);
  }
}
```

Writing Console Output

- The methods used for streaming output are defined in the **PrintStream** class. The methods used for writing console output are print(), println() and write().
- Both print() and println() methods are used to direct the output to the console. These methods are defined in the **PrintStream** class and are widely used. Both these methods are used with the help of the System.out stream.
- The basic differences between print() and println() methods are as follows:
- **print**() method displays the string in the same line whereas **println**() method outputs a newline character after its execution.
- **print**() method is used for directing output to console only whereas **println**() method is used for directing output to not only console but other sources also.

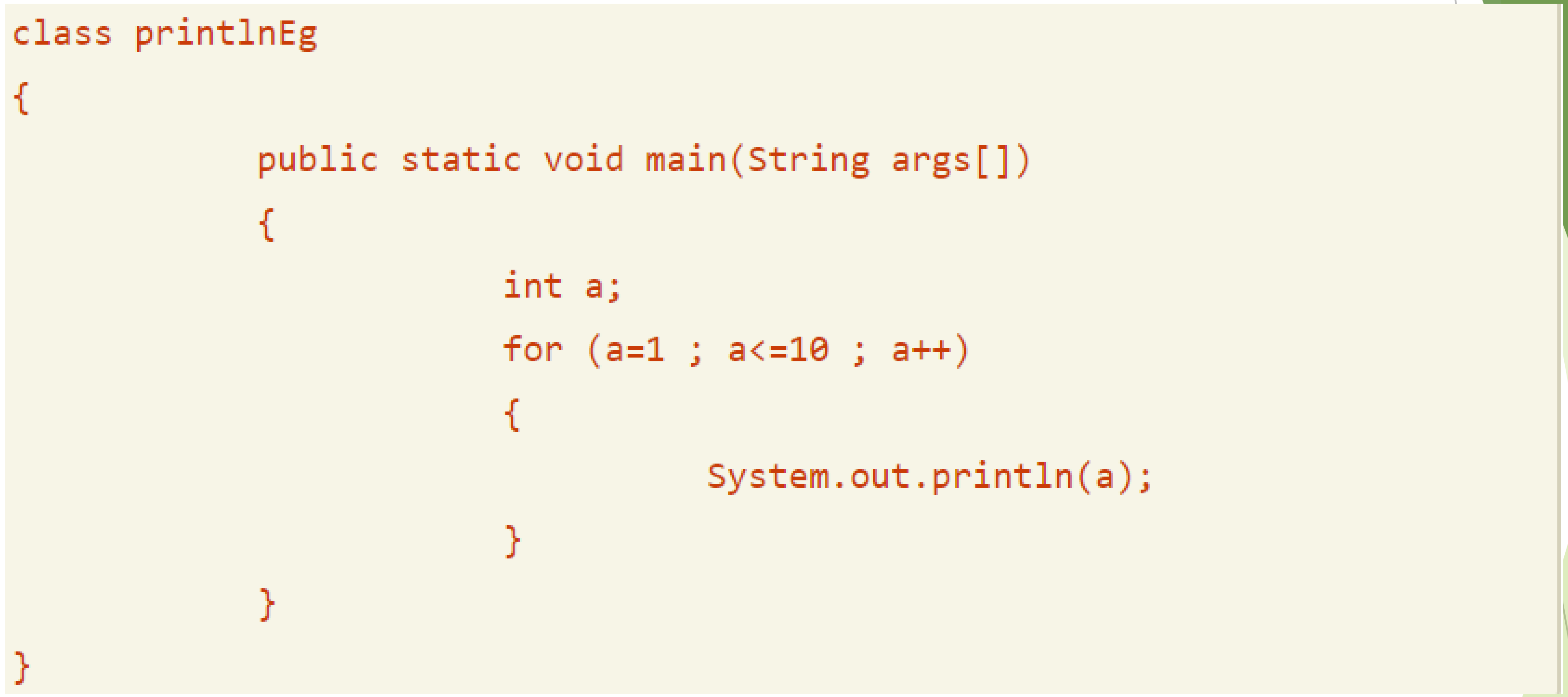
- **Example** of print() method:



```
// Sample program to display numbers from 1-10 using print method
class printEg
{
            public static void main(String args[])
            {
                        int a;
                        for (a=1 ;  a<=10 ;  a++)
                        {
                                    System.out.print(a);
                        }
            }
}
```

- Output
- 12345678910

- **Example** of println() method:



```
class printlnEg
{
            public static void main(String args[])
            {
                        int a;
                        for (a=1 ; a<=10 ; a++)
                        {
                                  System.out.println(a);
                        }
            }
}
```

- Output:

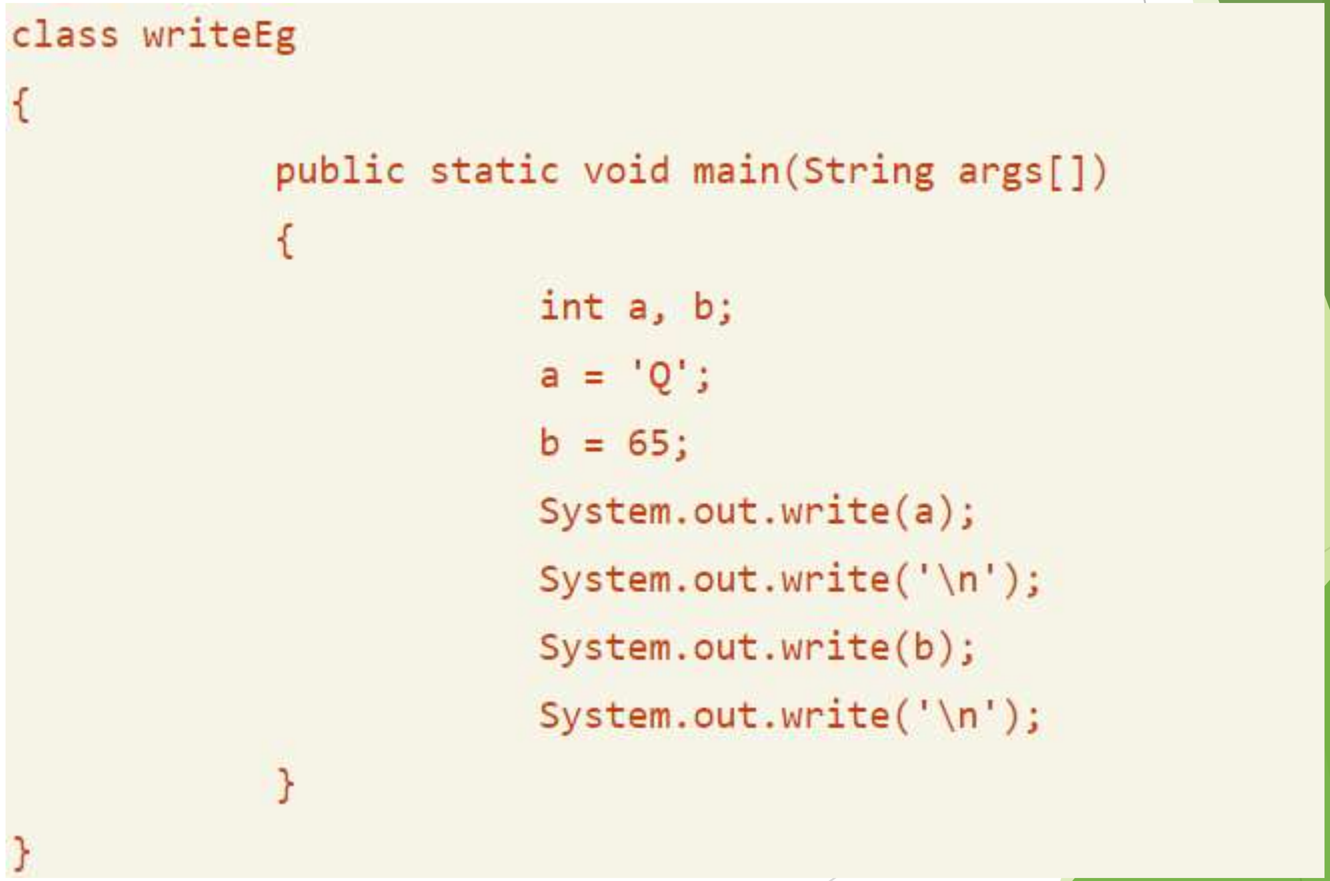
```
1
2
3
4
5
6
7
8
9
10
```

The write() method

- Alternatively, you can make use of the write() method for directing the output of your program to the console. The easiest ***syntax*** of the write() method is:

  void write(int b);

- Where b is an integer of low order eight bits.



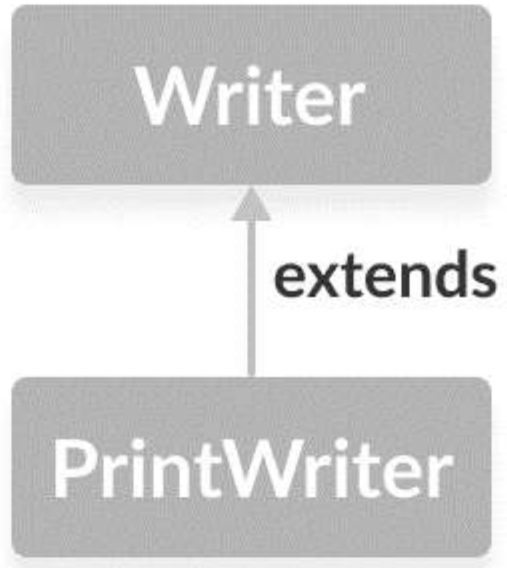
```
class writeEg
{
            public static void main(String args[])
            {
                        int a, b;
                        a = 'Q';
                        b = 65;
                        System.out.write(a);
                        System.out.write('\n');
                        System.out.write(b);
                        System.out.write('\n');
            }
}
```

- Output

```
     Q
     65
```

PrintWriter Class

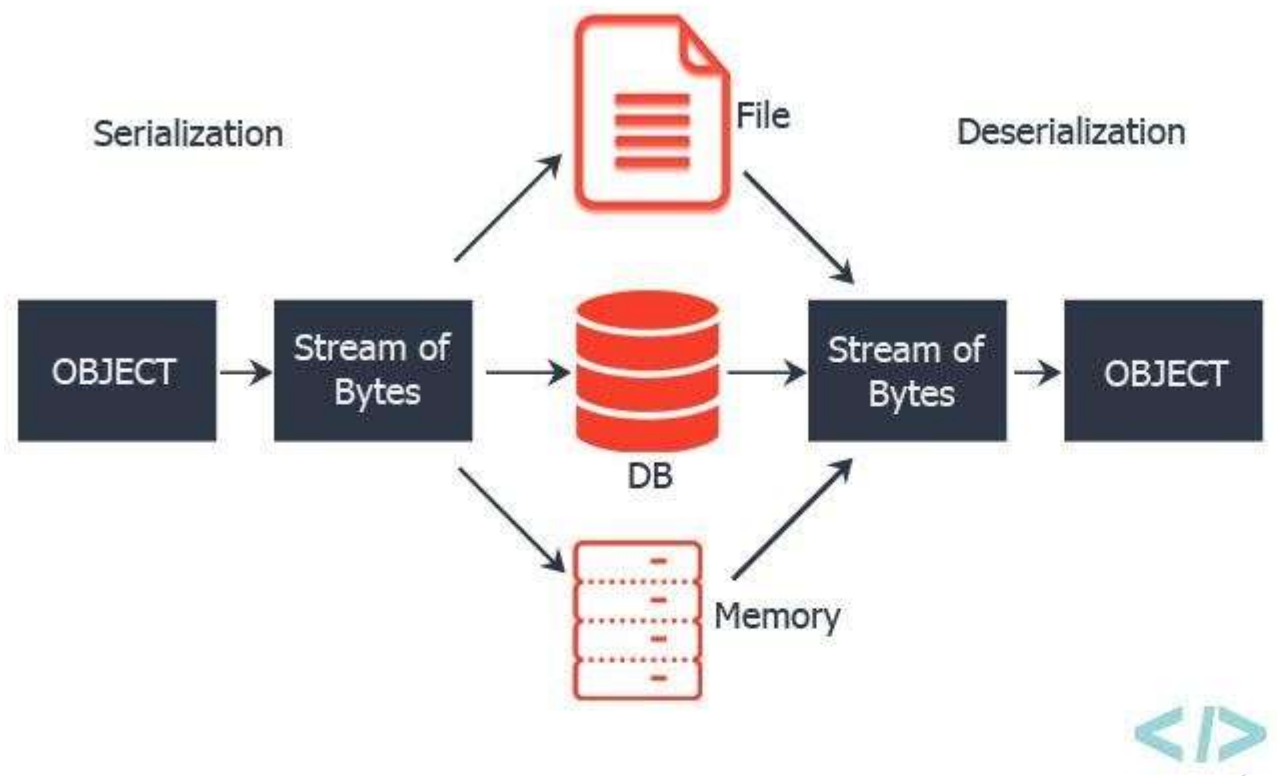
- The PrintWriter class of the java.io package can be used to write output data in a commonly readable form (text).
- PrintWriter class does not throw any input/output exception.
  - Instead, we need to use the checkError() method to find any error in it.
- Methods of PrintWriter
  - The PrintWriter class provides various methods that allow us to print data to the output.
  - print() - prints the specified data to the writer
  - println() - prints the data to the writer along with a new line character at the end
-



- Create a PrintWriter
  - In order to create a print writer, we must import the java.io.PrintWriter package first.
  - PrintWriter writerObj = new PrintWriter(System.out);
  - The printf() method can be used to print the formatted string. It includes 2 parameters: formatted string and arguments.
    - printf("I am %d years old", 25);

Object Streams and Serialization

- Serialization is the process of *converting the java code object into a byte stream*, to transfer the object code from one java virtual machine to another and recreate it using the process of Deserialization.



- **Why do we need Java Serialization?**
- Serialization allows us to transfer objects through a network by converting it into a byte stream.
- It also helps in preserving the state of the object.
- Deserialization requires less time to create an object than an actual object created from a class. hence serialization saves time.
- One can easily clone the object by serializing it into byte streams and then deserializing it.

- Serialization helps to implement persistence in the program.
  - It helps in storing the object directly in a database in the form of byte streams. This is useful as the data is easily retrievable whenever needed.
-

  Cross JVM Synchronization – It works across different JVM that might be running on different architectures(OS)

- **Object Streams and Serialization**

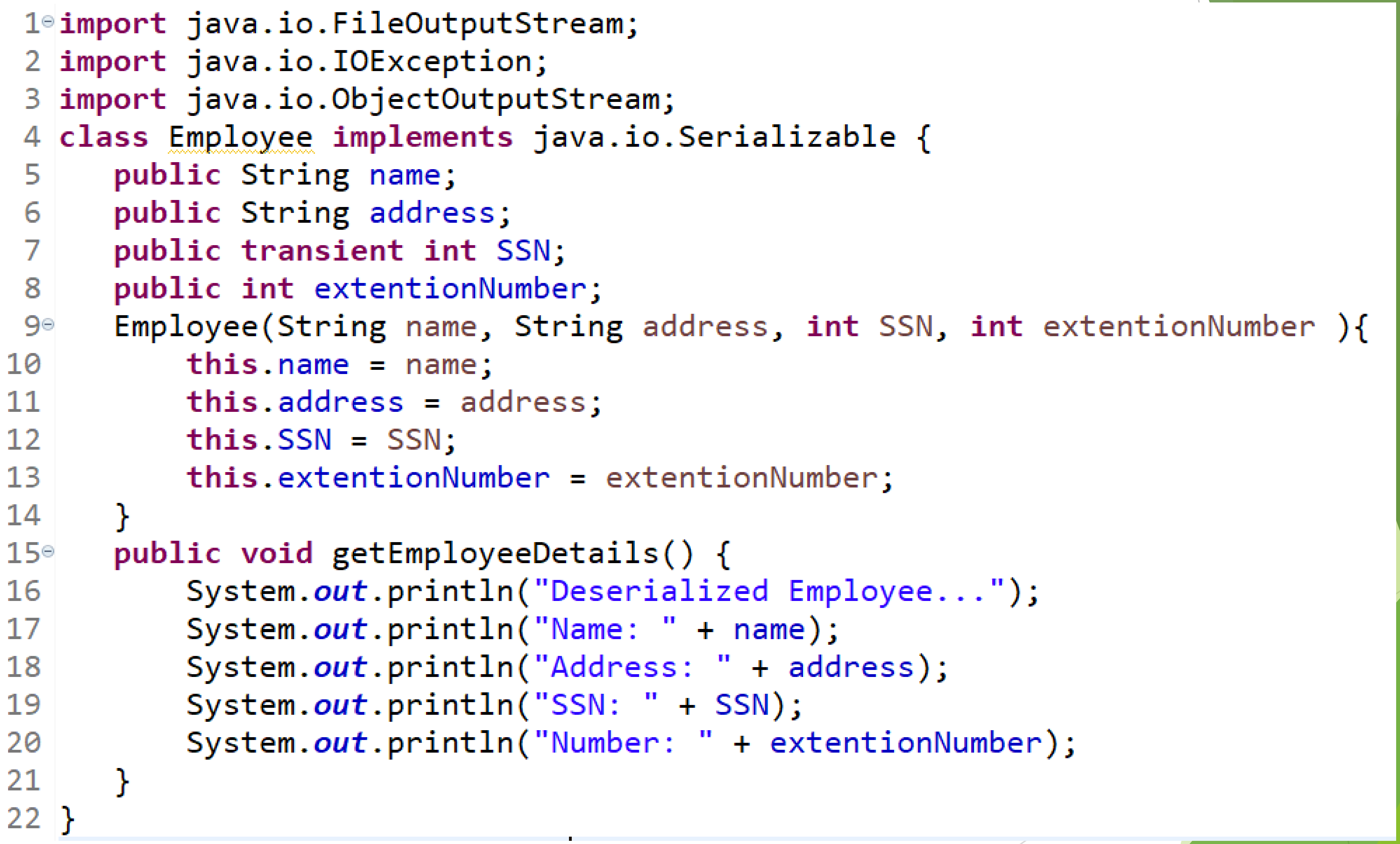
- Classes ObjectInputStream and ObjectOutputStream, which respectively implement the ObjectInput and ObjectOutput interfaces, enable entire objects to be read from or written to a stream.

- To use serialization with files, initialize ObjectInputStream and ObjectOutputStream objects with FileInputStream and FileOutputStream objects.

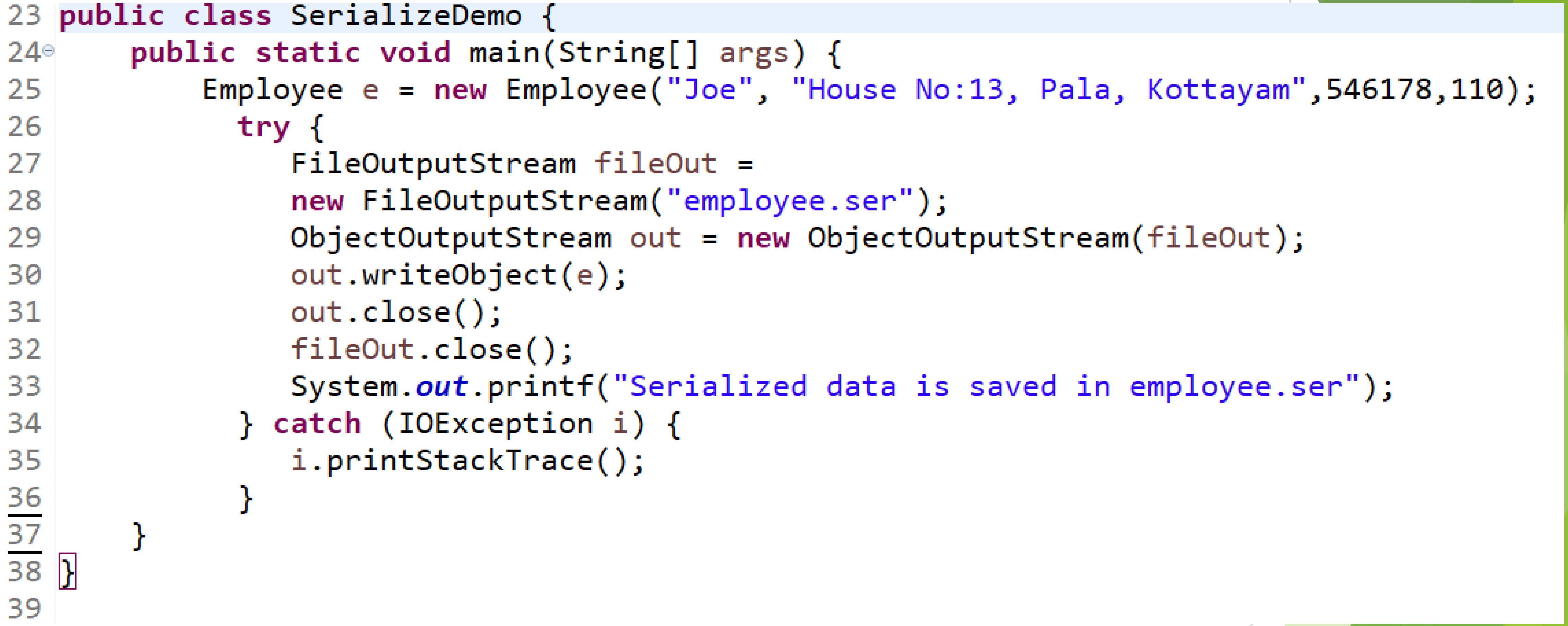
- ObjectOutput interface method writeObject takes an Object as an argument and writes its information to an OutputStream.
- A class that implements ObjectOuput (such as ObjectOutputStream) declares this method and ensures that the object being output implements Serializable.
- ObjectInput interface method readObject reads and returns a reference to an Object from an InputStream.
  - After an object has been read, its reference can be cast to the object's actual type.
- public final void writeObject(Object x) throws IOException
- public final Object readObject() throws IOException, ClassNotFoundException
- A class must implement java.io.Serializable interface to be eligible for serialization.

- Byte stream – read() and write()
- Character stream - read() and write() – reader and writer
- Object stream – read() and write() – readObject() and writeObject()

Serialization Demo

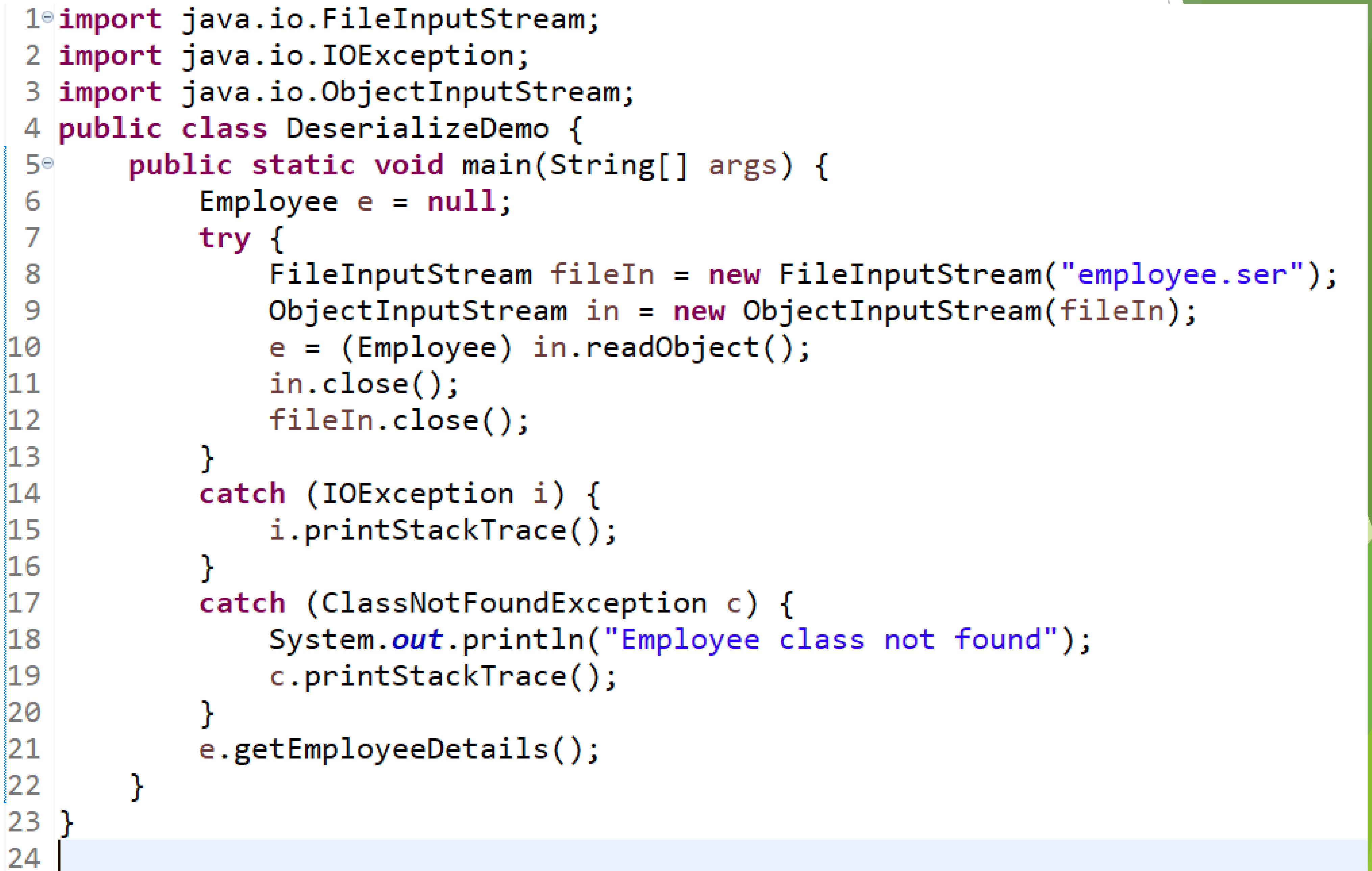


```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
class Employee implements java.io.Serializable {
    public String name;
    public String address;
    public transient int SSN;
    public int extentionNumber;
    Employee(String name, String address, int SSN, int extentionNumber ){
        this.name = name;
        this.address = address;
        this.SSN = SSN;
        this.extentionNumber = extentionNumber;
    }
    public void getEmployeeDetails() {
        System.out.println("Deserialized Employee...");
        System.out.println("Name: " + name);
        System.out.println("Address: " + address);
        System.out.println("SSN: " + SSN);
        System.out.println("Number: " + extentionNumber);
    }
}
```



```java
public class SerializeDemo {
    public static void main(String[] args) {
        Employee e = new Employee("Joe", "House No:13, Pala, Kottayam",546178,110);
          try {
             FileOutputStream fileOut =
             new FileOutputStream("employee.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut);
             out.writeObject(e);
             out.close();
             fileOut.close();
             System.out.printf("Serialized data is saved in employee.ser");
          } catch (IOException i) {
             i.printStackTrace();
          }
    }
}
```

Deserialization Demo



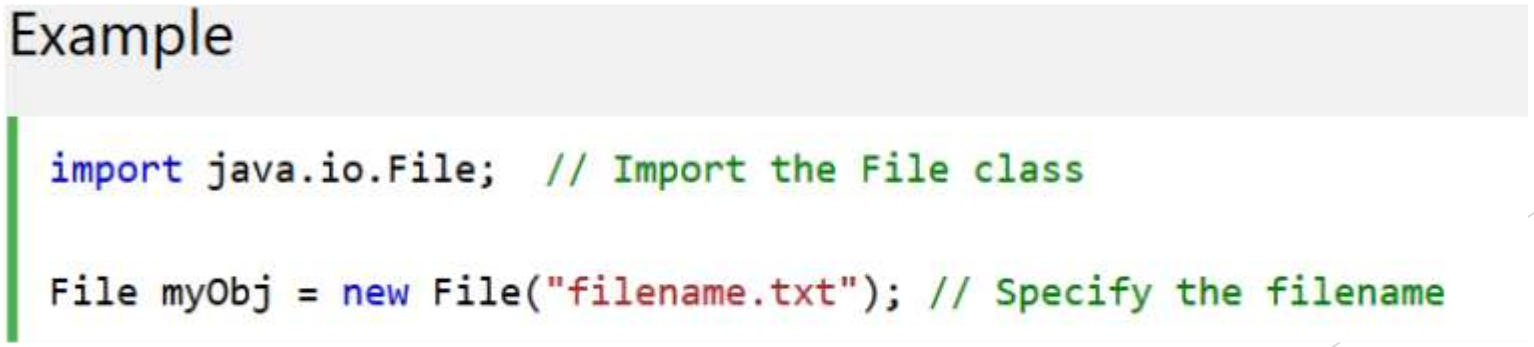
```java
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
public class DeserializeDemo {
    public static void main(String[] args) {
        Employee e = null;
        try {
            FileInputStream fileIn = new FileInputStream("employee.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            e = (Employee) in.readObject();
            in.close();
            fileIn.close();
        }
        catch (IOException i) {
            i.printStackTrace();
        }
        catch (ClassNotFoundException c) {
            System.out.println("Employee class not found");
            c.printStackTrace();
        }
        e.getEmployeeDetails();
    }
}
```

Working with Files

- Data stored in variables and arrays is temporary
- It’s lost when a local variable goes out of scope or when the program terminates
- For long-term retention of data, computers use files.
- Computers store files on secondary storage devices
  - hard disks, optical disks, flash drives and magnetic tapes.
- Data maintained in files is persistent data because it exists beyond the duration of program execution.

Java Files

- The File class from the java.io package, allows us to work with files.
- To use the File class, create an object of the class, and specify the filename or directory name:



Example

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

- **File Class**
- The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

- **Create a File**
- Use the createNewFile() method to create a file.
- This method returns a boolean value:
  - True if the file was successfully created, and False if the file already exists.
- Note that the method is enclosed in a try...catch block.
- This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):
-

Example

```java
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

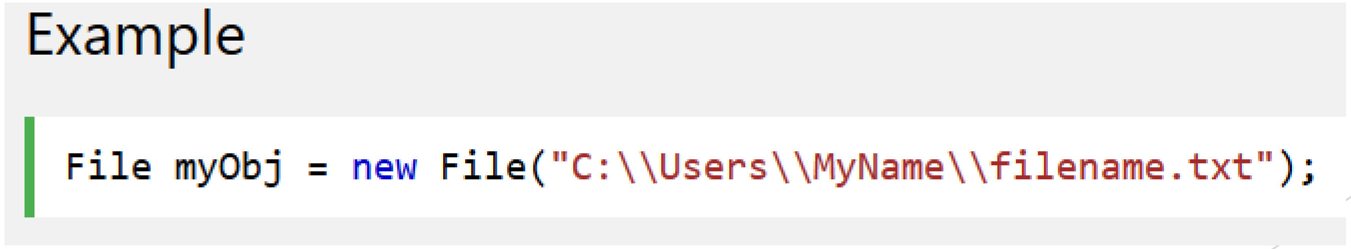
Creating File Objects

- Class File provides four constructors:

1. File(File parent, String child) : Creates a new File instance from a parent abstract pathname and a child pathname string.
2. File(String pathname) :Creates a new File instance by converting the given pathname string into an abstract pathname.
3. File(String parent, String child) : Creates a new File instance from a parent pathname string and a child pathname string.
4. File(URI uri) : Creates a new File instance by converting the given file: URI into an abstract pathname.

Create a File

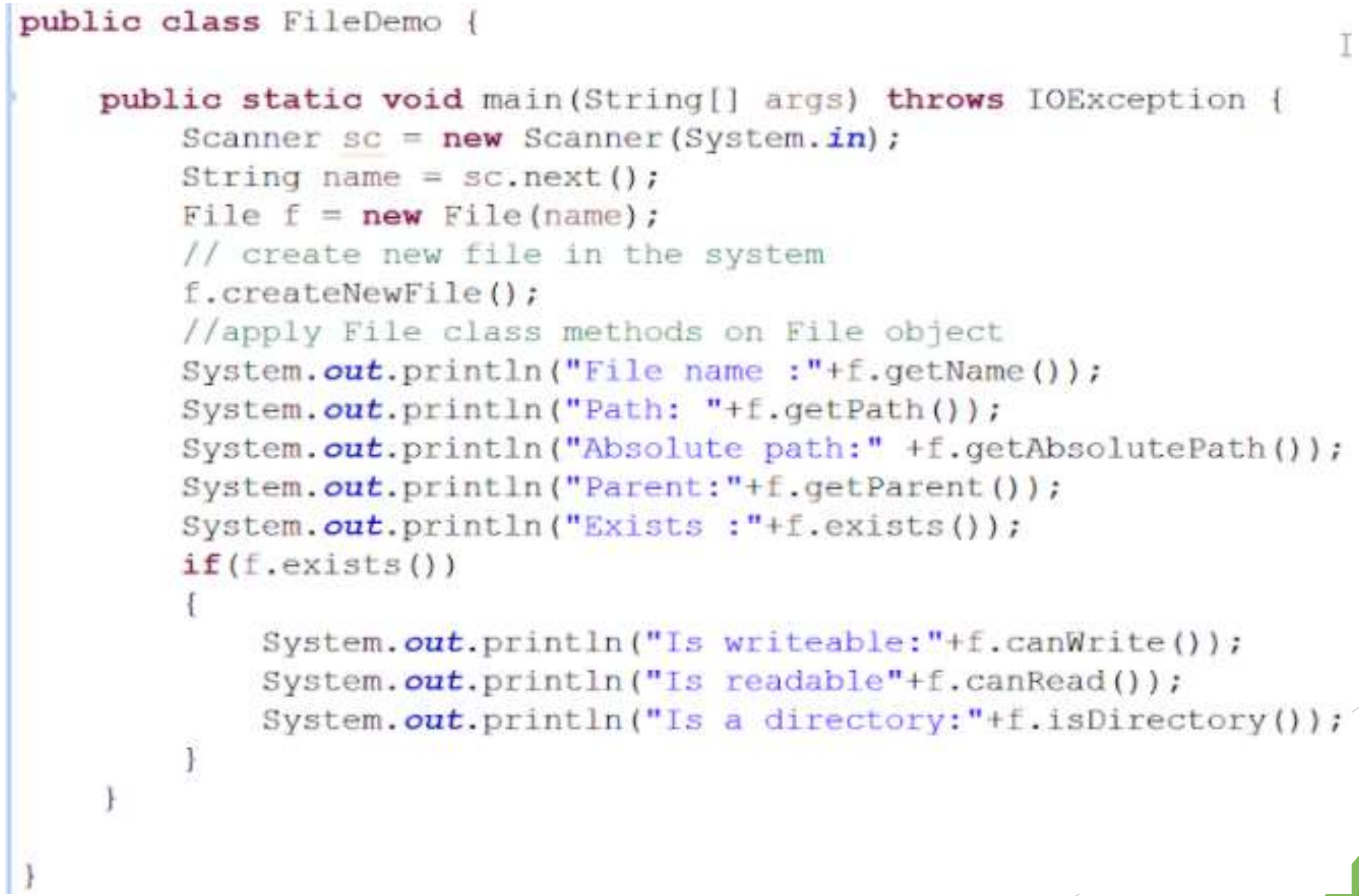
- To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows).
- On Mac and Linux you can just write the path, like: /users/name/filename.txt



Example

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

- **Get File Information**
- Now that we have created a file, we can use other File methods to get information about that file:
-



```
public class FileDemo {

    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        String name = sc.next();
        File f = new File(name);
        // create new file in the system
        f.createNewFile();
        //apply File class methods on File object
        System.out.println("File name :"+f.getName());
        System.out.println("Path: "+f.getPath());
        System.out.println("Absolute path:" +f.getAbsolutePath());
        System.out.println("Parent:"+f.getParent());
        System.out.println("Exists :"+f.exists());
        if(f.exists())
        {
            System.out.println("Is writeable:"+f.canWrite());
            System.out.println("Is readable"+f.canRead());
            System.out.println("Is a directory:"+f.isDirectory());
        }
    }

}
```

- **Java FileInputStream**
- The Java FileInputStream class, java.io.FileInputStream, makes it possible to read the contents of a file as a stream of bytes.
- The Java FileInputStream class is a subclass of Java InputStream.
-

```
InputStream input = new
FileInputStream("c:\\data\\input-
text.txt");
int data = input.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);
  data = input.read();
}
input.close();
```

- IOException

FileInputStream Constructors

- The `FileInputStream` class has a three different constructors you can use to create a `FileInputStream` instance.
- Two commonly used constructors are
  - `FileInputStream(String filePath)`

```
String path = "C:\\user\\data\\thefile.txt";
FileInputStream fileInputStream = new FileInputStream(path);
```

- `FileInputStream (File FileIObj)`

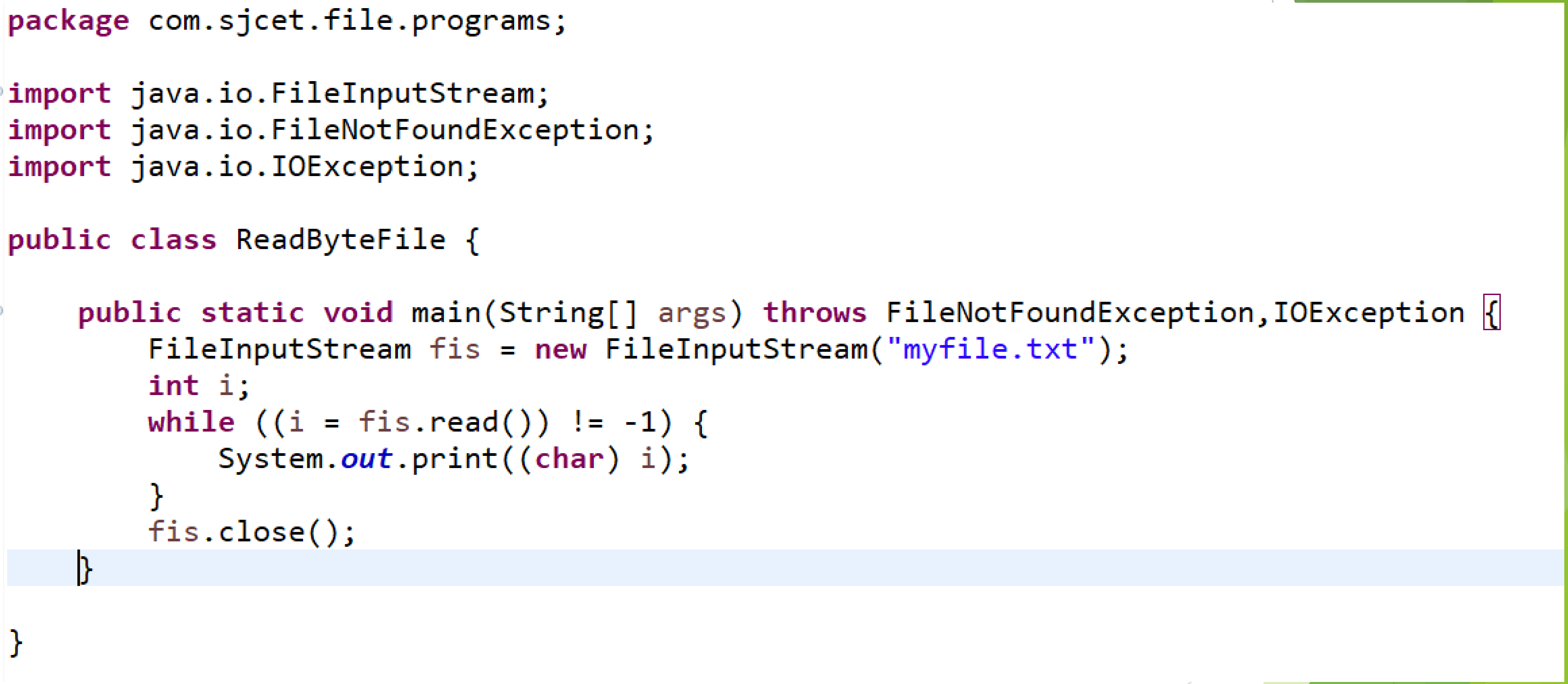
```
String path = "C:\\user\\data\\thefile.txt";
File   file = new File(path);
FileInputStream fileInputStream = new FileInputStream(file);
```

`FileNotFoundException`

- **FileInputStream - Important Methods:**
-

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to b.length bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to len bytes of data from the input stream. **b −** The destination byte array, **off −** The start offset in array b at which the data is written, **len −** The number of bytes to read.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
void close()	It is used to closes the stream.

FileInputStream - Example

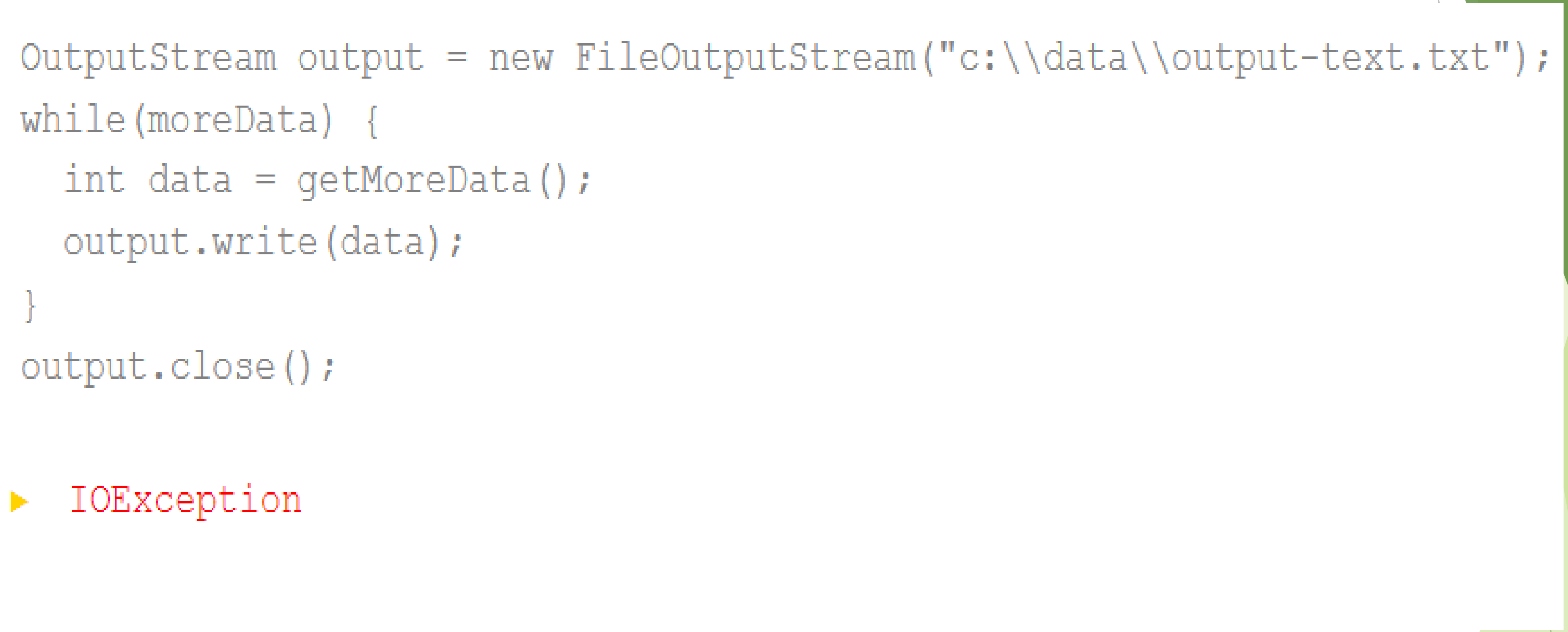


```java
package com.sjcet.file.programs;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ReadByteFile {

    public static void main(String[] args) throws FileNotFoundException,IOException {
        FileInputStream fis = new FileInputStream("myfile.txt");
        int i;
        while ((i = fis.read()) != -1) {
            System.out.print((char) i);
        }
        fis.close();
    }

}
```

- **Java FileOutputStream**

- FileOutputStream is an output stream for writing data to a File or to a FileDescriptor.
- FileOutputStream is a subclass of OutputStream, which accepts output bytes and sends them to some sink.
- In case of FileOutputStream, the sink is a file object.
- Sink controls the flow of data from the writer object into a single file

Java FileOutputStream Example



```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
while(moreData) {
  int data = getMoreData();
  output.write(data);
}
output.close();
```

- IOException

- **Java FileOutputStream constructors**

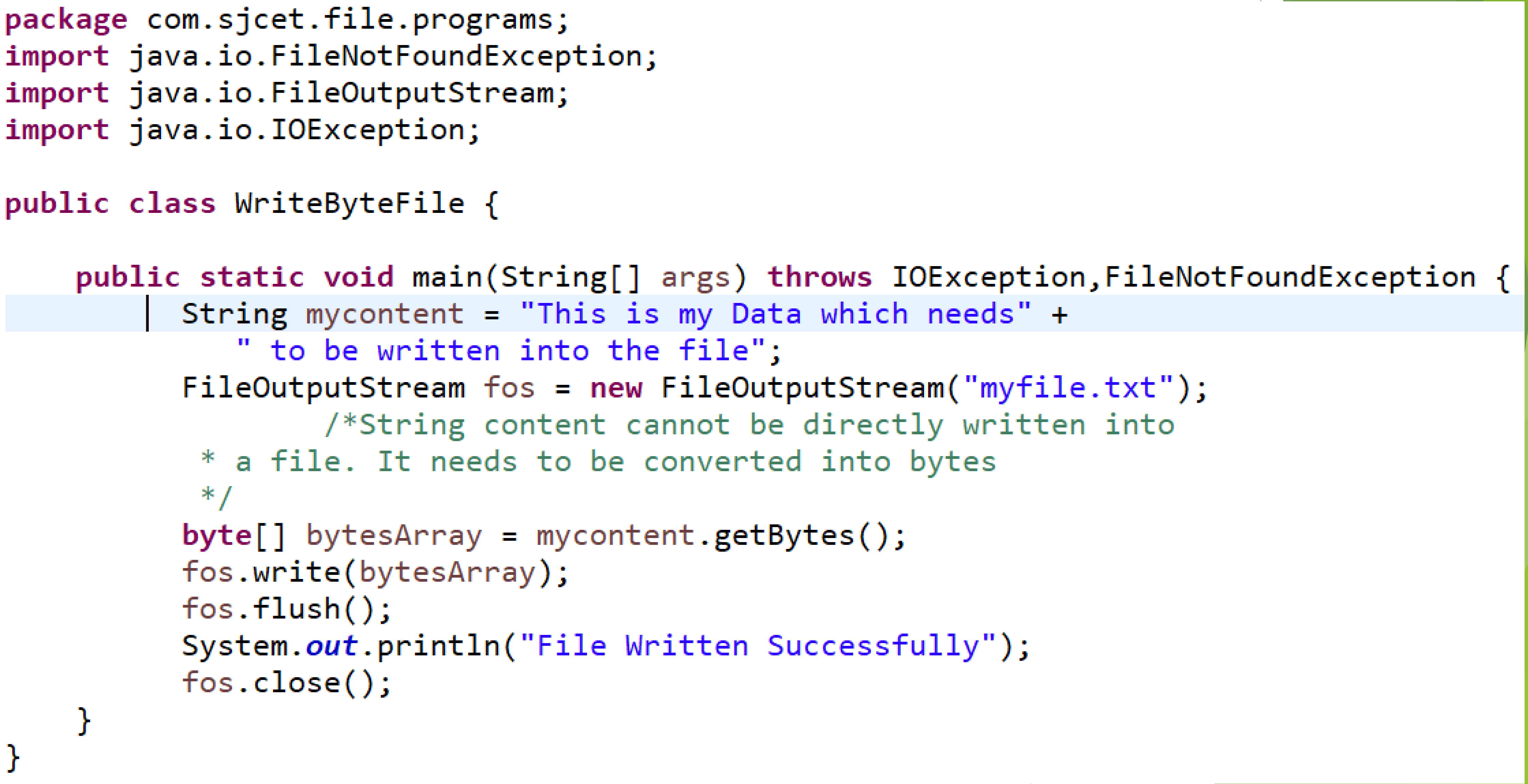
- FileOutputStream(File file)
  - creates a file output stream to write to a File object.
- FileOutputStream(File file, boolean append)
  - creates a file output stream to write to a File object; allows appending mode.
- FileOutputStream(String name)
  - creates a file output stream to write to the file with the specified name.
- FileOutputStream(String name, boolean append)
  - creates a file output stream to write to the file with the specified name;
  - allows appending mode.

- FileOutputStream(String name)
  - String path = "C:\\users\\jakobjenkov\\data\\datafile.txt";
  - FileOutputStream output = new FileOutputStream(path);
- FileOutputStream(File file)
  - String path = "C:\\users\\jakobjenkov\\data\\datafile.txt";
  - File   file = new File(path);
  - FileOutputStream output = new FileOutputStream(file);
- FileOutputStream(String name, boolean append)
  - OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");
  - OutputStream output = new FileOutputStream("c:\\data\\output-text.txt", true); //append
  - OutputStream output = new FileOutputStream("c:\\data\\output-text.txt", false); //overwrite

FileOutputStream - Important Methods:

Method	Description
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
void close()	It is used to closes the file output stream.

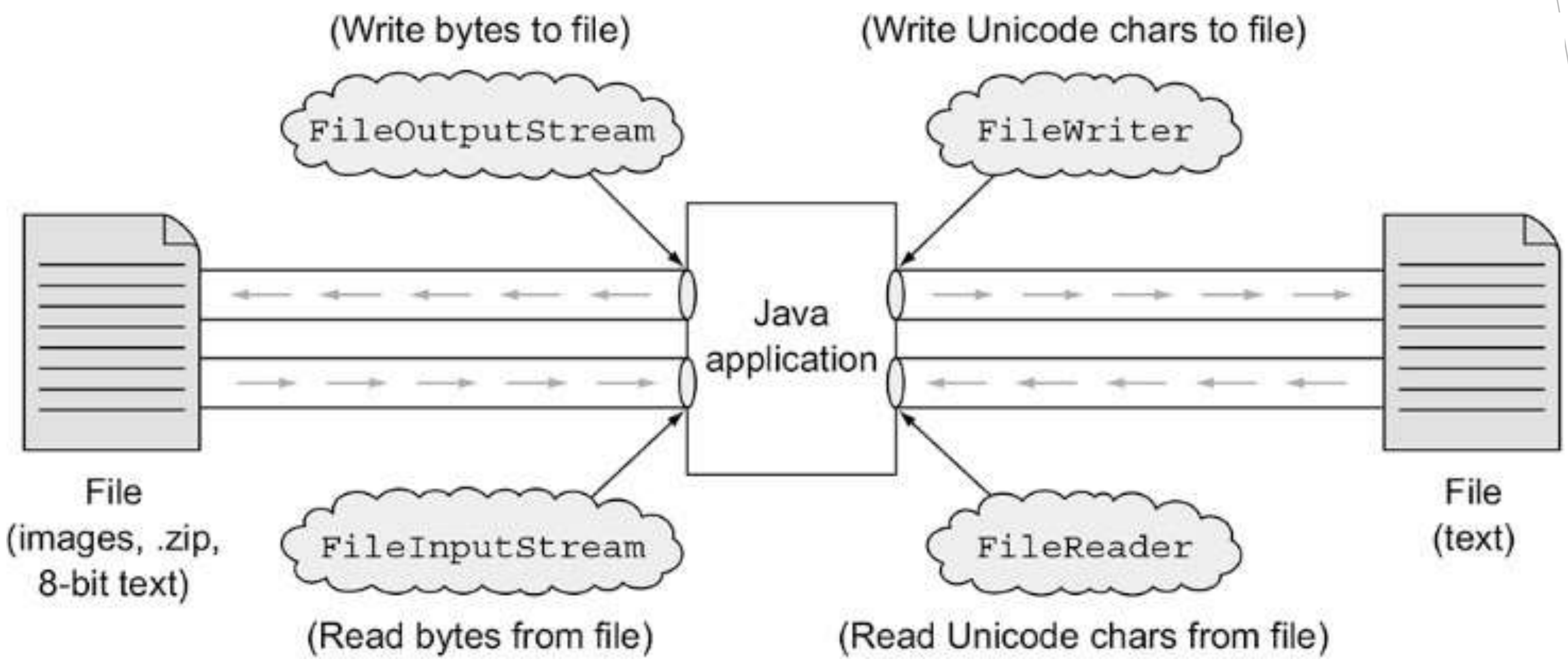
- FileOutputStream - Example
-



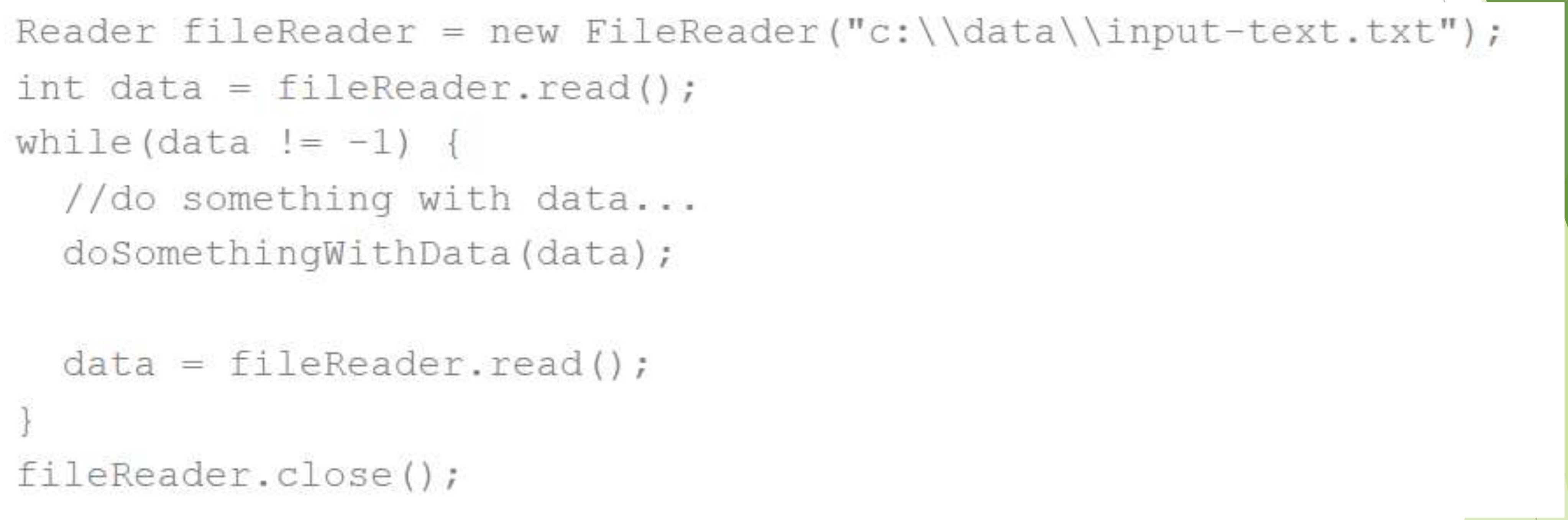
```java
package com.sjcet.file.programs;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteByteFile {

    public static void main(String[] args) throws IOException,FileNotFoundException {
          String mycontent = "This is my Data which needs" +
             " to be written into the file";
          FileOutputStream fos = new FileOutputStream("myfile.txt");
                  /*String content cannot be directly written into
           * a file. It needs to be converted into bytes
           */
          byte[] bytesArray = mycontent.getBytes();
          fos.write(bytesArray);
          fos.flush();
          System.out.println("File Written Successfully");
          fos.close();
    }
}
```

Java FileReader

- The Java FileReader class, java.io.FileReader makes it possible to read the contents of a file as a stream of characters.
- It works much like the FileInputStream except the FileInputStream reads bytes, whereas the FileReader reads characters.
- The FileReader is intended to read text, in other words.
- One character may correspond to one or more bytes depending on the character encoding scheme.

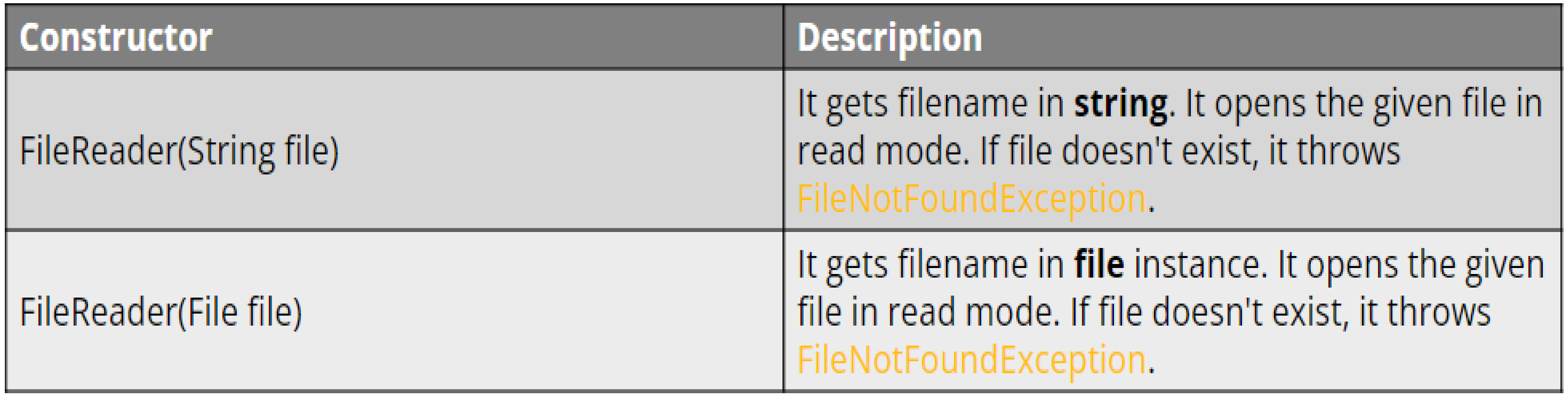

(Write bytes to file)
FileOutputStream
(Write Unicode chars to file)
FileWriter
Java application
File (images, .zip, 8-bit text)
FileInputStream
(Read bytes from file)
FileReader
(Read Unicode chars from file)
File (text)

Java FileReader Example



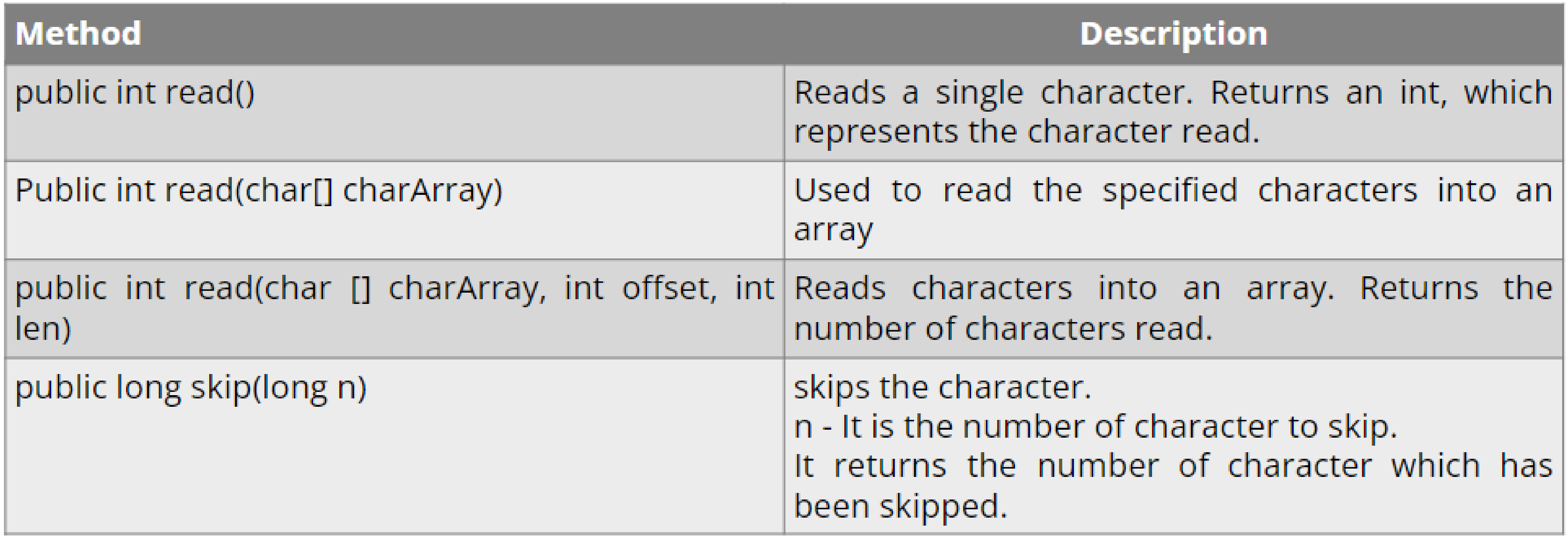
```
Reader fileReader = new FileReader("c:\\data\\input-text.txt");
int data = fileReader.read();
while(data != -1) {
  //do something with data...
  doSomethingWithData(data);

  data = fileReader.read();
}
fileReader.close();
```

Constructors of FileReader class



Constructor	Description
FileReader(String file)	It gets filename in **string**. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in **file** instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

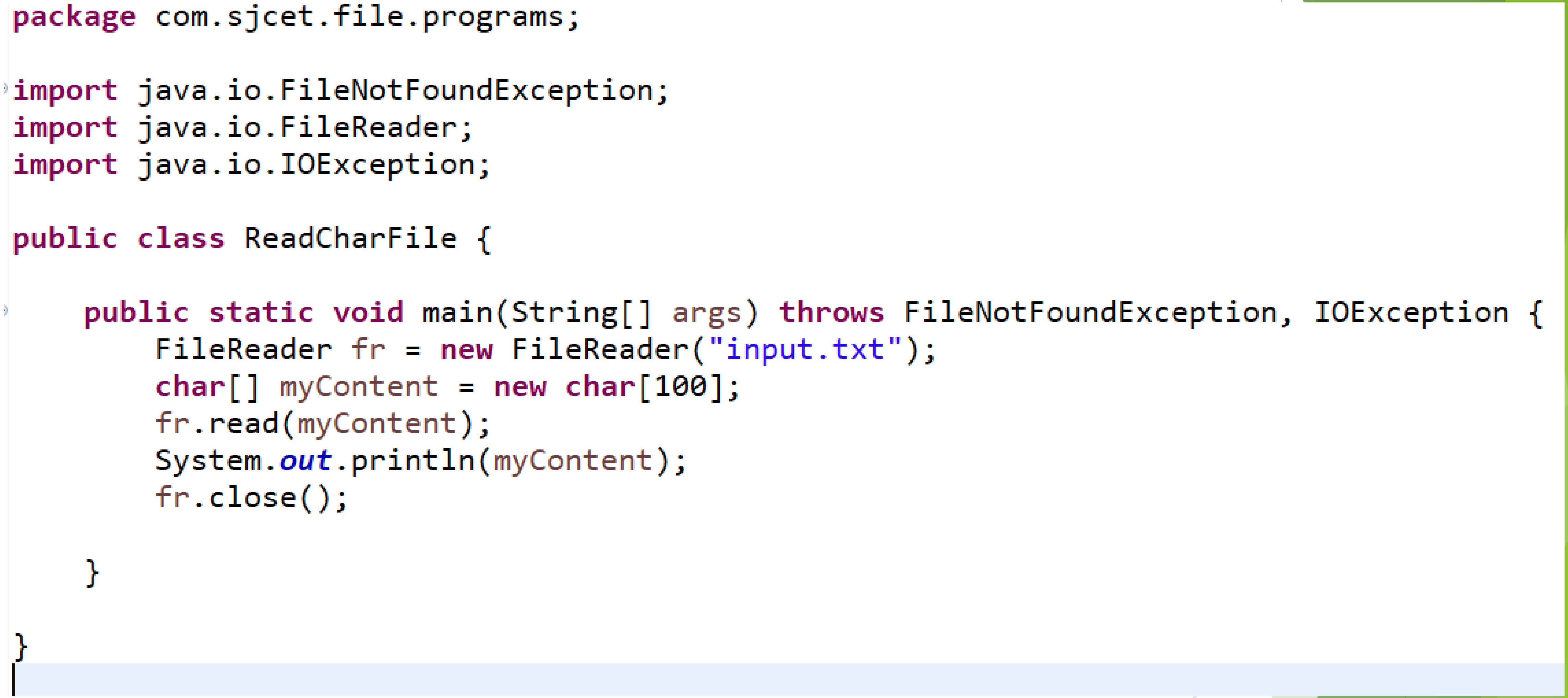
- **Methods of FileReader class**
-



Method	Description
public int read()	Reads a single character. Returns an int, which represents the character read.
Public int read(char[] charArray)	Used to read the specified characters into an array
public int read(char [] charArray, int offset, int len)	Reads characters into an array. Returns the number of characters read.
public long skip(long n)	skips the character. n - It is the number of character to skip. It returns the number of character which has been skipped.

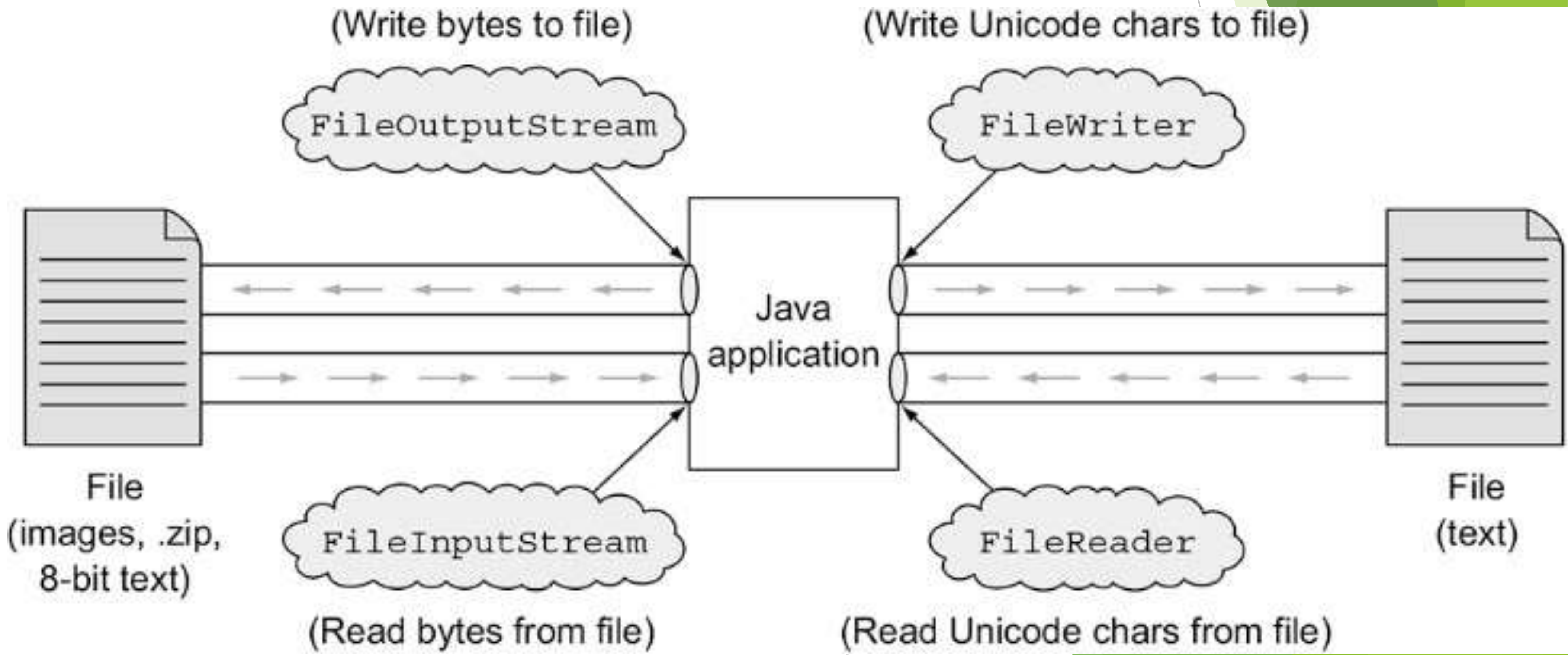
All Methods throws IOException

- FileReader Example
-



```
package com.sjcet.file.programs;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ReadCharFile {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        FileReader fr = new FileReader("input.txt");
        char[] myContent = new char[100];
        fr.read(myContent);
        System.out.println(myContent);
        fr.close();

    }

}
```

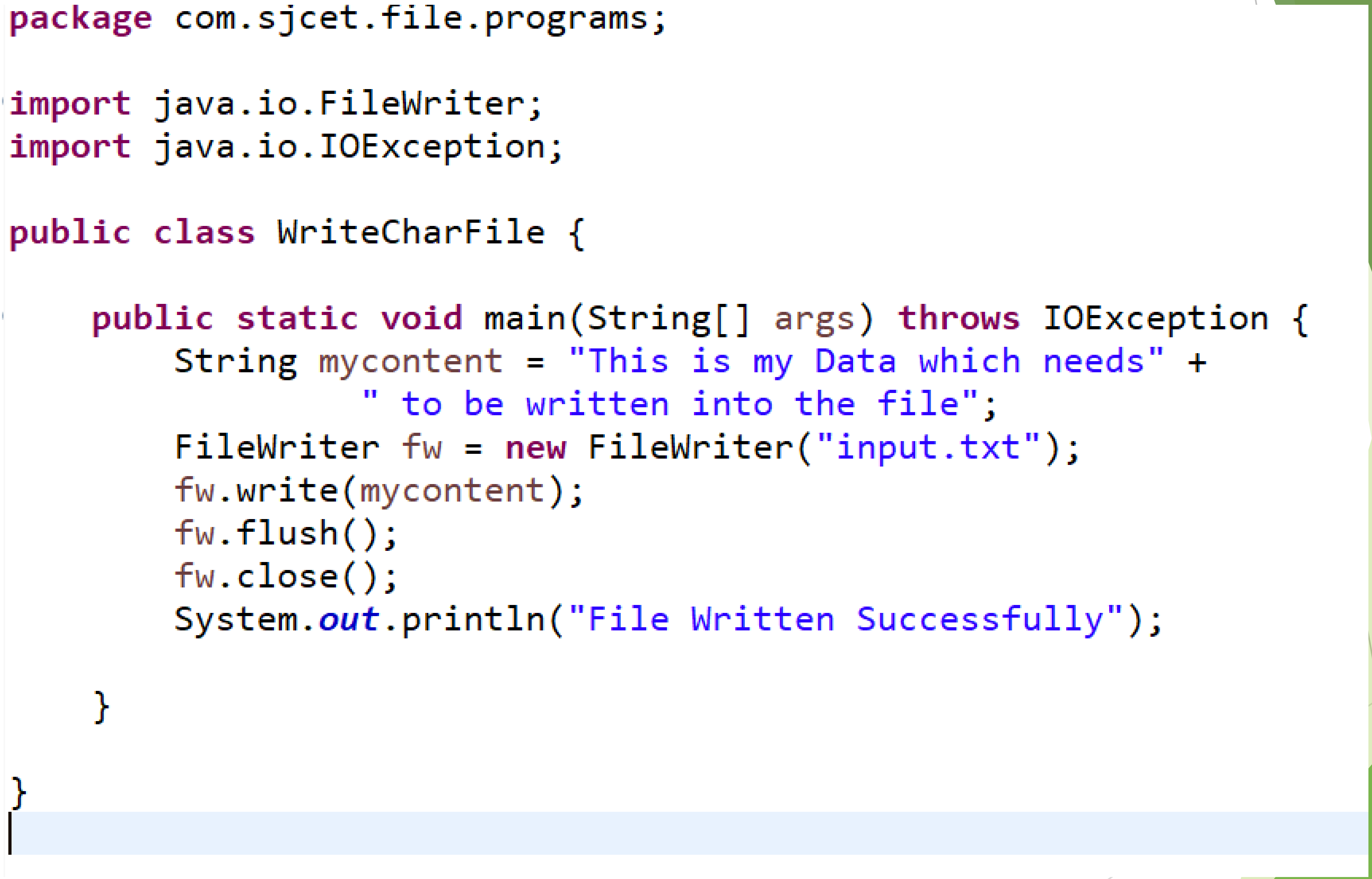
- **Java FileWriter**
- Java FileWriter class is a part of java.io package.
- FileWriter is meant for writing streams of characters.
- FileWriter is used to write to character files. Its write() methods allow you to write character(s) or strings to a file.
-



- **FileWriter Constructors**

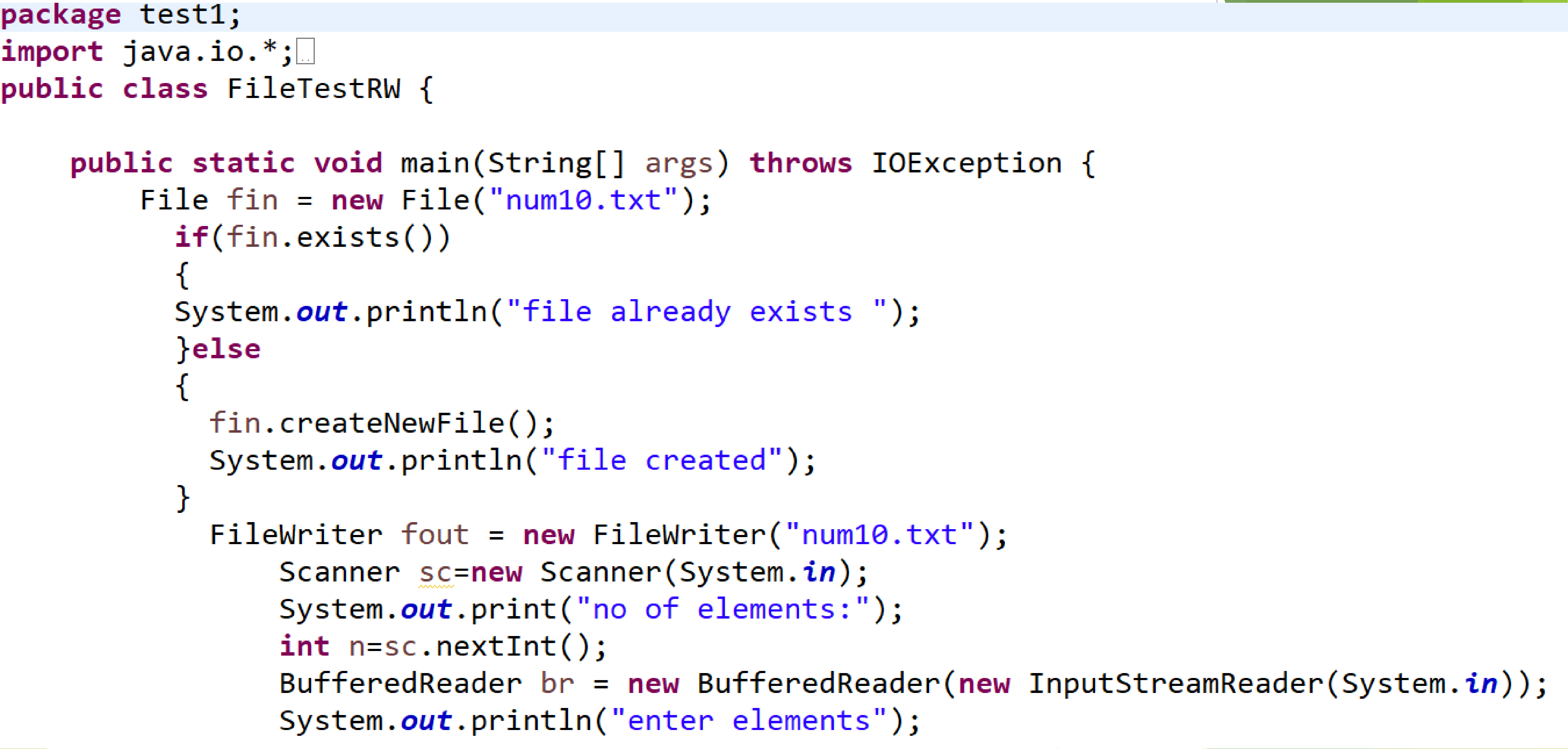
- FileWriter(File file)
- FileWriter(File file, boolean append)
- FileWriter(String fileName)
- FileWriter(String fileName, boolean append)

FileWriter Example



```
package com.sjcet.file.programs;

import java.io.FileWriter;
import java.io.IOException;

public class WriteCharFile {

    public static void main(String[] args) throws IOException {
        String mycontent = "This is my Data which needs" +
                  " to be written into the file";
        FileWriter fw = new FileWriter("input.txt");
        fw.write(mycontent);
        fw.flush();
        fw.close();
        System.out.println("File Written Successfully");

    }

}
```

Thank You!!!!!!!!!



```
package test1;
import java.io.*;
public class FileTestRW {

  public static void main(String[] args) throws IOException {
    File fin = new File("num10.txt");
     if(fin.exists())
     {
     System.out.println("file already exists ");
     }else
     {
      fin.createNewFile();
      System.out.println("file created");
     }
      FileWriter fout = new FileWriter("num10.txt");
        Scanner sc=new Scanner(System.in);
        System.out.print("no of elements:");
        int n=sc.nextInt();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("enter elements");
```

```
                for(int i=1;i<=n;i++)
                {
                    System.out.print("Enter the number "+i+":");
                    String num=br.readLine();
                    fout.write(num+" ");

                }
                System.out.println("numbers added");
                fout.close();
                FileReader fr = new FileReader("num10.txt");
                BufferedReader br1=new BufferedReader(fr);
                String s;
                while((s=br1.readLine()) != null)
                {
                System.out.println(s);
                }
                fr.close();

    }
```