Basics of the Object-Oriented Programming

Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Exceptions in Java

- An exception is a problem that arises during the execution of a program
- When an **Exception** occurs the normal flow of the program is disrupted, and the program terminates abnormally
- It is recommended to handle the exceptions
- An exception can occur for different reasons:
  - A user has entered an invalid data in the application
  - A file that needs to be opened cannot be found
  - A file can not be read

Exceptions in Java

- There are three categories of Exceptions
  - **Unchecked exceptions**
    - » Are exceptions that occurs at the time of execution
    - » Are also called as **Runtime Exceptions**
    - » Include programming bugs, such as logic errors
    - » Are ignored at the time of compilation
    - » Example of **RuntimeException**
      - ArithmeticException
      - NullPointerException
      - ArrayIndexOutOfBoundsException
    - » Are exceptions that can be prevented programmatically
      - e.g. NullPointerException can be prevented if you check for null before calling any method
      - Similarly, ArrayIndexOutOfBoundException would never occur if you check the index first

Exceptions in Java

- There are three categories of Exceptions
  - **Checked exceptions**
    - » Are exceptions that are checked by the compiler at compilation-time
    - » Are also called compile time exceptions
    - » The programmer should handle these exceptions
  - **Errors**
    - » Are problems that arise beyond the control of the user or the programmer
    - » Are ignored in your code because you can rarely do anything about an error
      - For example, if a stack overflow occurs, an error will arise
    - » They are also ignored at the time of compilation

Exceptions in Java – Example of **Unchecked exceptions**

```
public class Exemplu{
  public static void main(String args[]){
     int v[] = new int[10];
      v[10] = 0; //Exception !
     System.out.println("You can't get here anymore..."); } }
```

- If the program is compiled and executed, the following exception is throwed:

  "Exception in thread "main“ java.lang.ArrayIndexOutOfBoundsException: 10 at Exemplu.main (Exemplu.java:4)“

Java exceptions - Example of **Checked exceptions**

- If you use **FileReader** class in your program to read data from a file, the following type of exceptions can generate
  - FileNotFoundException if the file specified in its constructor doesn't exist (in the constructor of **FileReader** class)
  - IOException if the file could not be read or closed(**read** and **close** methods from **FileReader)**

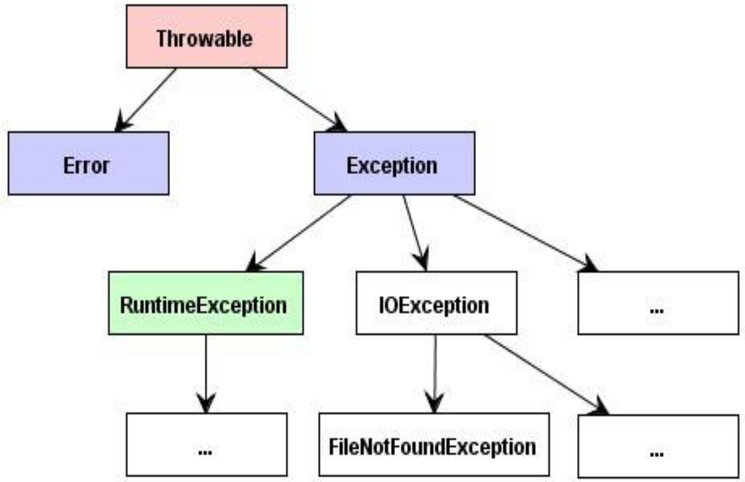
Java exceptions - Example of **Checked exceptions**

```
import java.io.File;
import java.io.FileReader;
public class FilenotFoundDemo {
   public static void main(String args[]) {
      File file = new File(“D://file.txt");
      FileReader f = new FileReader(file);
     //read and display the file, character by character
      int c;
      while((c=f.read())!= -1)
         System.out.print((char)c);
      //close the file
      f.close();
}
   }
```

The compiler notifies to handle IOException, along with FileNotFoundException

Java Exceptions - Exception Hierarchy

- All exception classes are subtypes of the java.lang.**Exception** class
- **Exception and Error** classes are subclasses of the **Throwable** class
  - Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs
  - Errors are generated to indicate errors generated by the runtime environment
    - » Example: JVM is out of memory
- **Exception** class has two main subclasses: **IOException** class and **RuntimeException** Class



Java Exceptions - Exception Hierarchy

- **The most important** methods from **Throwable** class:
  - **public String getMessage()**
    - »Returns a detailed message about the exception that has occurred
    - »This message is initialized in the **Throwable** constructor
  - **public String toString()**
    - »Returns the name of the class concatenated with the result of **getMessage**()
    - »Inherited from **Object** class
  - **public void printStackTrace()**
    - »Prints the result of **toString**() along with other details like the line number and class name where the exception occurred

Java exceptions - Catching Exceptions

- 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
- The syntax is:

```
try
  {//Instructions that can generate exceptions}
catch (ExceptionType1 variable)
  {// handling exceptions 1}
catch (ExceptionType2 variable)
  {// handling exceptions 2}
 finally{}
   . . .
```

Java exceptions - Catching Exceptions

- The code which is prone to exceptions is placed in the **try** block
- When an exception occurs, that exception occurred is handled by **catch** block associated with it
- Every **try** block should be immediately followed either by a **catch** block or **finally** block.
- A **catch** statement involves declaring the type of exception you are trying to catch

```
try
  {//Instructions that can generate exceptions}
catch (ExceptionType1 variable)
  {// handling exceptions 1}
```

Java exceptions - Catching Exceptions - Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFoundDemo {
   public static void main(String args[]) {
   try
       {
          File file = new File(“D://file.txt");
          FileReader fr = new FileReader(file);
       }
    catch(FileNotFoundException e)
     {
        System.err.println(“File not found");
     }
  }
}
```

Java exceptions - Catching Exceptions

- Multiple **catch** blocks
  - A try block can be followed by multiple **catch** blocks
    - » If an exception occurs in the protected code, the exception is thrown to the first **catch** block in the list
    - » If the data type of the exception thrown matches ExceptionType1, it gets caught there
    - » If not, the exception passes down to the second **catch** statement
    - » This continues until the exception either is **caught** or falls through all catches, in which case the current method stops execution, and the exception is thrown down to the previous method on the call stack

```
try
 {
   // Protected code
 }
catch (ExceptionType1 e1)
 {
   // Catch block
 }
 catch (ExceptionType2 e2)
 {
   // Catch block
 }
catch (ExceptionType3 e3)
 {
   // Catch block
 }
```

Java exceptions - Catching Exceptions - Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFoundDemo {
  public static void main(String args[]) {
  FileReader f = null;
  try {//open the file
        f = new FileReader(“D://file.txt");
       //read and display the file, character by character
       int c;
       while((c=f.read())!=-1)
          System.out.print((char)c);}
   catch(FileNotFoundException e) {
      System.err.println(“File not found");}
```

```
catch(IOException e)
    {   // Another type of exception is treated
        System.out.println(“Error to read”);
     }
finally{
      if (f != null)
       {  //close the file
          try
           { f.close();  }
          catch(IOException e)
           { System.err.println(“The file cannot be closed!"); }
        }
      }
}
```

Java exceptions - Catching Exceptions - **Throws** Keywords

- If a method does not handle a checked exception, the method must declare it using the **throws** keyword
- **Throw** an exception is made by specifiy a **throws** clause in the method declaration

[modifiers] **returnedType** method ([*arguments*]) **throws** $TExceptie_1$, $Exceptie_2$, ...
{…}

Java exceptions - Catching Exceptions - Throws Keywords

```
public class ReadFile{
  public static void readFile(String fis) throws
    FileNotFoundException, IOException
    {
        FileReader f =  new FileReader(fis);
        int c;
        while((c=f.read())!= -1)
              System.out.print((char)c);
       f.close();
    }
```

```
public static void main(String args[]) {
   if(args.length>0){
      try{readFile(args[0]);}
      catch(FileNotFoundException e) {
         System.err.println(“file not found");}
      catch(IOException e) {
         System.out.println("Error to read");}}
      else
        System.out.println(“The file name is missing");
  }
}
```

Java exceptions - Catching Exceptions - Throws Keywords

- For considered example:
  - Exception handling doesn’t make in **read** method
  - Called method (i.e., **main** method) handle the exceptions
    - » Not distinguish between exceptions caused by reading from the file, or the exception caused by closing the file
    - » Both exceptions are of type **IOException**
    - » Closing the file will not be made if an exception occurs while reading from file

Java exceptions - Catching Exceptions – **Finally** block

- The **finally** block follows a **try** block or a **catch** block
- A **finally** block of code always executes, irrespective of occurrence of an **Exception**
- A **finally** block appears at the end of the **catch** blocks
- We can use **finally** block without **catch block**

```
public static void readFile(String fis) throws
FileNotFoundException, IOException
  { FileReader f = null;
    try
        {
            f = new FileReader(fis);
            int c;
            while((c=f.read()) != -1)
               System.out.print((char)c);
          }
   finally{
        if(f!=null)
          f.close(); }}
```

Java exceptions - Catching Exceptions – **Throw** Keyword

- Is used to **throw** an exception

```
try {
    if (index >= vector.length) {
        throw new ArrayIndexOutOfBoundsException();
    }
    // Code that uses the vector array
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index out of bounds!");
}
```

Java exceptions - Catching Exceptions – **Throw** Keyword

```
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);}}
public class Example {
    public void divide(int a, int b) throws CustomException {
        if (b == 0) {
            throw new CustomException("Cannot divide by zero!");
        }
        int result = a / b;
        System.out.println("Result: " + result);
    }
    public static void main(String[] args) {
        Example example = new Example();
        try {  example.divide(10, 0);
        } catch (CustomException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }}}
```

- In this example, the divide method checks if the value of b is zero and if it is, it throws a CustomException with the message "Cannot divide by zero!".
- The main method creates an instance of the Example class and calls the divide method with arguments 10 and 0, which will result in a CustomException being thrown. The catch block catches the CustomException and prints the exception message to the console.
- Note that the divide method declares that it throws a CustomException using the throws keyword. This is necessary because CustomException is a checked exception, which means that it must be declared in the method signature or handled in a try-catch block.

Java exceptions - Advantages of catching exceptions

- Cod separation
- Errors propagation
- Groups errors by type

Java exceptions - Advantages of catching exceptions - **Code separation**

- Let’s consider that we want to write a program that read from a file
- The main steps are specified bellow:

```
readFile {
    open the file;
    determine the size of the file;
    allocate memory;
    read the file in the memory;
    close the file;
}
```

Java exceptions - Advantages of catching exceptions - **Code separation**

- The traditional code looks like this:

```
int readFile() {
    int errorCod = 0;
    open the file;
    if(the file opened)
       {determine the size of the file;
         if(the size was determined)
          {allocate memory;
             if(the memory is allocated)
                {read the file in the memory;
                   if(the file couldn’t read)
                     {errorCod = -1; }
                  }
         else { ...}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
   return errorCod; }
```

Java exceptions - Advantages of catching exceptions - **Code separation**

- Code using java exception mechanism looks like this:

```
int readFile() {
   try { open the file;
         determine the size of the file;
         allocate memory;
         read the file in the memory;
         close the file;}
   catch(the file couldn't open){Treatment of Error;}
   catch(the size could not be determined) {Treatment of Error;}
   catch(the memory could not be allocated) {Treatment of Error;}
   catch(the file could not be read) {Treatment of Error;}
   catch(the file could not be closed) {Treatment of Error; }}
```

Java exceptions - Advantages of catching exceptions - **Errors Propagation**

- When exception is occurred at the top of the stack and no exception handler is provided then exception is propagated:

```
int metoda1() {
  try{ metoda2(); }
  catch(TipExceptie e) {//proceseazaEroare; }
  ... }
int metoda2() throws TipExceptie{  metoda3(); … }
int metoda3() throws TipExceptie{ citesteFisier();
… }
```

- We can see that:
  - Exception is occurred in the method3() and in method3() we don’t have any exception handler
  - Uncaught exception will be propagated downward in stack (i.e., it will check appropriate exception handler in the method2())
  - Again, in method2 we don’t have any exception handler then again exception is propagated downward to method1() where it finds exception handler

Java exceptions - Advantages of catching exceptions - **Groups errors by type**

- Each exception is described by a class
- Classes describing exception are hierarchical organized

```
try{
   FileReader f = new FileReader("input.dat");
  //Exceptie posibila: FileNotFoundException }
catch(FileNotFoundException e) {
   //Exceptie specifica provocata de absenta
   //fisierului “input.dat” }
catch(IOException e) {
 //Exceptie generica provocata de o operatie IO}
catch(Exception e) {
  //Cea mai generica exceptie soft}
catch(Throwable e) {
//Superclasa exceptiilor}
```

Java exceptions - User-defined Exceptions

- You can create your own exceptions in Java
  - All exceptions must be a child of **Throwable**
  - If you want to write a checked exception that is automatically enforced by the Handle, you need to extend the **Exception** class

Java exceptions - User-defined Exceptions

```
class ExceptieStiva extends Exception {
  public ExceptieStiva(String mesaj)
    {super(mesaj);}}

class Stiva {
   int elemente[] = new int[100];
   int n=0; //numarul de elemente din stiva
   public void add(int x) throws ExceptieStiva {
      if(n==100)
           throw new ExceptieStiva("Stiva este plina!");
      elemente[n++] = x;}
public int remove() throws ExceptieStiva{
   if(n==0)
      throw new ExceptieStiva("Stiva este goala!");
   return elemente[n--];}}
```

Input/Output Streams in Java

- Most applications require:
  - Reading information from an external source
  - Writing information to an external destination
- Information
  - It can be of any type:
    - Primitive data, objects, images, sounds, etc.
  - Can be found anywhere
    - In a disk, network, memory

Input/Output Streams in Java

- Streams
  - Are communication cannel on 8 or 16 bits between two processes
  - Are unidirectional from producer to consumer
  - Each stream has a producer process and a consumer process
  - Between two processes there are any number of streams
  - Any process can be both producer and consumer at the same time, but on different streams
- We can have:
  - Input stream - is the stream that reads data
  - Output stream - is the stream that writes data
- Producer process
  - Process that describe an external data source
- Consumer process
  - Process that describe an external data destination

Input/Output Streams in Java

- A stream can represent different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays
- Streams support many kinds of data, including:
  - Simple bytes
  - Primitive data types
  - Localized characters
  - Objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways
- We ca see a stream as a sequence of data
  - A program uses an *input stream* to read data from a source, and an *output stream* to write data to a destination

Inputs and outputs

- **java.io**
  - It is the package that provides support for input/output operations
- General scheme for stream usage:

```
Open communication channel
while(there is more information to be processed)
    {read/write information;}
 close the communication channel;
```

Inputs and outputs

- Streams classification
  - The direction of the communication channel:
    - » Input streams (for reading)
    - » Output streams (for writing)
  - Data type:
    - » **Byte streams** (8 bits)
    - » **Character streams** (16 bits)
  - Their actions
    - » Primary streams
      - Implements **read/write** methods defined in superclasses
    - » Processing streams
      - Are responsible for processing data from a primitive stream

Input/Output Streams in Java - Primitive streams

- Are divided by the type of the data source in:
  - **File** - are used for read/ write from/in a file
    - »FileReader, FileWriter
    - »FileInputStream, FileOutputStream
  - **Memory** - are use for read/write information from /in memory; They are created on an already existing vector
    - »Character streams: CharArrayReader, CharArrayWriter
    - »Byte streams: ByteArrayInputStream, ByteArrayOutputStream
    - »StringReader, StringWriter
  - **Pipe** - implements input/output components of a data pipeline
    - »PipedReader, PipedWriter
    - »PipedInputStream, PipedOutputStream

Input/Output Streams in Java - Processing streams

- Based of the processing type that performed the processing streams are classified in:
  - ”Buffering” – clases used for introducing a buffer in the read/write process of the data
    - »BufferedReader, BufferedWriter
    - »BufferedInputStream, BufferedOutputStream
  - Byte-character conversion – abstract classes that define a common interface for streams that automatically filters read/written data
    - »InputStreamReader
    - »OutputStreamWriter
  - Concatenation - concatenates multiple input streams into one
    - » SequenceInputStream

Input/Output Streams in Java - Processing streams

- Based of the type of processing that performed the processing streams are classified in:
  - Serialization – classes used for objects serializations
    - **ObjectInputStream**, **ObjectOutputStream**
  - Converts data types **-** classes used to write read primitive data in a binary format, independent of the machine on which it is working
    - **DataInputStream**, **DataOutputStream**
  - Counting - automatically count the lines read from an input stream
    - **LineNumberReader**, **LineNumberInputStream**
  - Display - provides methods for displaying information
    - **PrintWriter, PrintStream**

Input/ Output Streams in Java - Character streams

- All character stream classes are descended from Reader and Writer classes
  - **Reader** – root class for input streams
    - Is extend by the following classes: **BufferedReader**, **InputStreamReader**, **StringReader**….
    - In turn the class **InputStreamReader** is extended by the class **FileReader**
  - **Writer** – root class for output streams
    - Is extended by the following classes: **BufferedWriter**, **OutputStreamWriter**, **StringWriter**
    - In turn the class **OutputStreamWriter** is extended by the class **FileWriter**

Input/Output Streams in Java– Byte streams

- All byte streams classes are descended from **InputStream** and **OutputStream** classes
  - **InputStream** – root class for input streams
    - Is extend by the following classes: **FileInputStream**, **FilterInputStream**, **ObjectInputStream**
    - In turn the class **ObjectInputStream** is extended by the class **BufferedInputStream**
  - **OutputStream** - root class for output streams
    - Is extend by the classes: **FileOutputStream, FilterOutputStream**, **ObjectOutputStream**
    - In turn the class **FilterOutputStream** is extended by the class **BufferedOutputStream**

Input/Output Streams in Java - Common Methods for Reader, InputStream

Reader	**InputStream**
int read() – read a character	**int read**() – reads the next byte of data from the input stream
int read(**char** buf[]) – and reads some number of characters from the character stream into the buffer array buf	**int read**(**byte** b[]) – reads some number of bytes from the input stream and stores them into the buffer array *b*
…	….

Input/Output Streams in Java - Common Methods for **Writer**, **OutputStream**

Writer	**OutputStream**
void **write**(**int** c) – write a character	void **write**(**int** c) – writes the specified byte to the output stream
void **write**(**char** buf[]) – write an array of characters	void **write**(**byte** buf[]) – writes all the bytes in the array to the output stream
void **write**(**String** str) – write a string	**-**
….	….

- Close a stream: with the **close** method
- Exceptions: **IOException** or classes that extends I**OException**

Input/ Output Streams in Java - Steps in defining a primitive stream

FluxPrimitiv numeFlux = **new** FluxPrimitiv(dispozitivExtern);

Example:

```
//crearea unui flux de intrare pe caractere
FileReader in = new FileReader("fisier.txt");

//crearea unui flux de iesire pe caractere
FileWriter out = new FileWriter("fisier.txt");

//crearea unui flux de intrare pe octeti
FileInputStream in = new FileInputStream("fisier.dat");

//crearea unui flux de iesire pe octeti
FileOutputStream out = new FileOutputStream("fisier.dat");
```

Input/ Output Streams in Java - Steps in defining a processing stream

FluxProcesare numeFlux = **new FluxProcesare**(fluxPrimitiv);

Example:

```
//crearea unui flux de intrare printr-un buffer
BufferedReader in = new BufferedReader(new FileReader("fisier.txt"));
//echivalent cu
FileReader fr = new FileReader("fisier.txt");
BufferedReader in = new BufferedReader(fr);

//crearea unui flux de iesire printr-un buffer
BufferedWriter out = new BufferedWriter(new FileWriter("fisier.txt")));
//echivalent cu
FileWriter fo = new FileWriter("fisier.txt");
BufferedWriter out = new BufferedWriter(fo);
```

Input/ Output Streams in Java - Streams for working with files

- Classes for working with files:
  - **FileReader**
    - » It possible to read the contents of a file as a stream of characters
  - **FileWriter**
    - » Makes it possible to write characters to a file
  - **FileInputStream**
    - » Makes it possible to read the contents of a file as a stream of bytes
  - **FileOutputStream**
    - » Makes it possible to write a file as a stream of bytes

Input/ Output Streams in Java - Streams for working with files

- Classes for working with files:

```
//copy  a file
 import java.io .*;
  public class Copiere{
    public static void main(String [] args){
      try{
        FileReader in = new FileReader("in.txt");
        FileWriter out = new FileWriter("out.txt");
        int c;
        while ((c = in.read())!=-1)
            out.write(c);
        in.close();
        out.close();
          }
     catch (IOException e){
        System.err.println(“Eroare la operatiile cu fisiere !");
        e. printStackTrace();}}}
```

Input/ Output Streams in Java - Read/Write with buffer

- Classes for read/write with buffer:
  - Introduce a buffer (a memory area) in the read/write process
  - *BufferedReader*
    - » Is used to read the text from a character-based input stream
    - » It provides buffering functionality, which can make reading from the input stream more efficient by reducing the number of I/O operations that are needed.
    - » It inherits *Reader* class
  - *BufferedWriter*
    - » Is used to provide buffering for *Writer* instances
    - » Uses an internal buffer to provide efficient writing of data to the output stream.
    - » It inherits from the Writer class and can be used to write single characters, arrays of characters, and strings to a character stream.

Input/ Output Streams in Java - Read/Write with buffer

- Classes for read/write with buffer (cont'):
  - **BufferedInputStream class**
    - » Provides buffering functionality for reading data from an input stream
    - » Like BufferedReader and BufferedWriter, BufferedInputStream uses an internal buffer to provide efficient reading of data from the input stream
  - **BufferedOutputStream class**
    - » Is used for buffering an output stream
    - » It internally uses buffer to store data
    - » It adds more efficiency than to write data directly into a stream

Input/ Output Streams in Java - Read/Write with buffer

- Classes for read/write with buffer

```
BufferedOutputStream out = new BufferedOutputStream(new
FileOutputStream("out.dat"), 1024)
  //1024 is buffer size
```

- Goals:
  - Efficiency – decrese the accesing number of the device
  - Extern – increase the execution speed

Read/Write with buffer

- Methods define in **BufferedReader**, **BufferedWriter**
  - **read** – read a character
  - **write** – write a character
  - **flush** – empty the buffer, even it is not full

Read/Write with buffer - Example

```
BufferedWriter out = new BufferedWriter(new FileWriter("out.dat"), 1024)
//we create a buffer writer of 1024 octets
for(int i=0; i<100; i++)
   out.write(i);
   // the buffer is not empty, nothing was written in the file
    out.flush();
   //the buffer is empty, the data is written to the file
  //we call readLine method
   BufferedReader br = new BufferedReader(new FileReader("in.txt"));
   String linie;
   while ((linie = br.readLine()) != null) {
         ...       //process the line form the file}
    br.close(); }
```

DataInputStream and **DataOutputStream classes**

- Provides methods for read/write data at a primitive level, and not a byte level

DataInputStream	**DataOutputStream**
readBoolean	writeBoolean
readByte	writeByte
readChar	writeChar
readDouble	writeDouble
readFloat	writeFloat
readInt	writeInt
readLong	writeLong

DataInputStream and **DataOutputStream classes (Example)**

```
public  class Test{
 public static void main(String args[]) throws IOException{
          //Scriere
          FileOutputStream fos = new FileOutputStream("test.dat");
          DataOutputStream out = new DataOutputStream(fos);

          out.writeInt(12345);
          out.writeDouble(12.345);
          out.writeBoolean(true);
          out.writeUTF("Sir de caractere");
          out.flush();
          fos.close();
          //Citire
          FileInputStream fis = new FileInputStream("test.dat");
          DataInputStream in = new DataInputStream(fis);
```

```
   int i = in.readInt();
   double d = in.readDouble();
   boolean b = in.readBoolean();
   String s = in.readUTF();
   System.out.println(d);
   System.out.println(b);
   System.out.println(s);
   fis.close();
   }
}
```

```
>>output
 12.345
 true
 Sir de caractere
```

Formatted Inputs and Outputs

- **java.util.Scanner** class
  - Formatted inputs
  - Is a class in java.util package used for obtaining the input of the primitive types like int, double, and strings.
  - It is the easiest way to read input in a Java program, though not very efficient
  - To create an object of **Scanner** class we may pass
    - The predefined object **System.in**, which represents the standard input stream
    - An object of class **File** if we want to read input from a file
  - To read numerical values of a certain data type, the function to use is **nextDataType**()
    - E.g., to read a value of type int, **nextInt**() can be used, to read strings, the **nextLine**() method can be used

Formatted Inputs and Outputs –Scanner class(Example)

```
Scanner s = Scanner(System.in);
String nume = s.next();
int varsta = s.nextInt();
double salariu = s.nextDouble();
s.close();

File file = new File("file/dep.txt");
    try {
        Scanner sc = new Scanner(file);
        while (sc.hasNextLine()) {
           String s = sc.nextLine();
             }
        sc.close();
    }
    catch (FileNotFoundException e) {
        e.printStackTrace();   }}
```

Formatted Inputs and Outputs

- PrintStream, PrintWriter classes
  - Formatted outputs
  - Have
    - » Methods for display of an array of characters: **print**, **println**
    - » Methods for formatted display of some variables: **format**, **printf**
- Formatting of the character arrays is based on the **java.util.Formatter** class

Standard input/output streams

- **System.in**
  - Standard input stream of **InputStream** type
- Syste**m.out**
  - Standard output stream of **PrintStream** type
- **System.err**
  - Standard error stream of **PrintStream** type

Standard input/output streams

- Display information on the screen
  - System.out.print(argument);
  - System.out.println(argument);
  - System.out.printf(format, argumente...);
  - System.out.format(format, argumente...);

```
//Example of displaying information on the screen

int i = 461012;
System.out.format("The value of i is: %d %n", i);
//The %d specifies that the single variable is a decimal integer.
//The %n is a platform-independent newline character.
//The output is: The value of i is: 461012
```

```
//Display errors

.....
catch(Exception e) {System.err.println("Exceptie:" + e);}
```

Read the date from the keyboard – BufferedReader class

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Introduceti o linie:");
String linie = stdin.readLine()
System.out.println(linie);
```

Read the date from the keyboard – BufferedReader class

```
//Citeste siruri de la tastatura si verifica daca reprezinta numere sau nu
import java.io.*;
public class EsteNumar {
  public static void main(String[] args) {
  BufferedReader stdin = new BufferedReader(new
                                        InputStreamReader(System.in));
  try {
        while(true){
          String s = stdin.readLine();
          if(s.equals("exit") || s.length()= =0)
              break;
         System.out.print(s);
         try{
               Double.parseDouble(s);
               System.out.println(":DA");
              }
 catch(NumberFormatException e)
              {  System.out.println(":NU");  }
      }
}
catch(IOException e) {
   System.err.println("Eroare la intrarea standard!")
                                    e.printStackTrace();
   }
  }
}
```

Redirecting standard streams

- Implies
  - The establishment of other source than keyboard for reading the data
  - The establishment of other source than the screen for data output
- In **System** classes there are the following static methods:
  - **setIn**(InputStream) – redirects the input
  - **setOut**(PrintStream) - redirects the output
  - **setErr**(PrintStream) - redirects the error

Redirecting standard streams

- Redirects the output is useful when a lot of data is displayed on the screen
  - We can redirect the display to a file
- Example of redirecting the outputs

```
import java.io.FileNotFoundException;
import java.io.PrintStream;
public class TestRedirect {
 public static void main(String args[]) throws FileNotFoundException {
   System.out.println("January");
   System.out.println("February");
   PrintStream ps = new PrintStream("D:/exempleCurs/streams/sample.txt");
   System.setOut(ps);
   System.out.println("March");
   System.out.println("April");
  ps.close();
 }
}
```

>>output:
January
February