CS205 Object Oriented Programming in Java

Module 3 - **More features of Java (Part 1)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Introduction:
  - ☑ **Packages** and Interfaces:
    - ☑Defining Package,
    - ☑CLASSPATH,
    - ☑AccessProtection
    - ☑ Importing Packages

Package

- **Packages** *are* ***containers for classes.***
- A package in Java is used to **group** related classes and interfaces.
- They are used to keep the class name space compartmentalized.
  - For example, a package allows us to create a class named **List,** which we can store in our own package and it will not collide with some other class named **List** stored elsewhere.
- Packages are stored in a *hierarchical manner.*
- The package is both a **naming** and a **visibility control mechanism.**

Packages(contd.)

- We can define classes inside a package
  - that are not accessible by code outside that package. (default)

  OR

  - that can be also accessed by subclasses outside the package. (protected)

  OR

  - That can be accessed by all classes in all packages(public)

Defining Package

- To **create a package**, simply include a **package command** as the **first statement** in a Java source file.
  - All classes declared in that file will belong to the specified package.
- The package statement **defines a name space** in which classes are stored.
- If we are **not** writing package statement, the class names are put into the ***default package, which has no name***.

Defining Package(contd.)

- General form for **creating a package :**

  package *packagename;*

  Example: If we write the following statement at the beginning of our java program then it will create a package named **Oop**.

package ***Oop;***

Defining Package(contd.)

- Java uses file system **directories** to store packages.
- Example: Any classes that we declare to be part of the package **Oop** must store their **.class** files in a directory called **Oop.**
- Any file can include the same package statement.
- The package statement simply specifies to which package the classes defined in a file belongs to.

Defining Package(contd.)

- We can create a **hierarchy of packages.**
  - Separate each package name from other using period(dot) symbol.
- General form of a multileveled package statement is :

package *pkg1.pkg2.pkg3;*

This specifies that package pkg3 is inside package pkg2 and pkg2 package is inside pkg1.

- E.g The package declared as

  package java.awt.image;
  - needs to be stored in the path **java\awt\image** in a Windows environment
- We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that we create?

1. By default, the Java run-time system uses the **current working directory** as its starting point.

   →if our package is in a subdirectory of the current directory, it will be found.

2. We can specify a directory path or set paths by setting the **CLASSPATH environmental variable.**

3. We can use the **-classpath** option **with java** and **javac** to specify the path to your classes.

CLASSPATH (contd.)

- Example

package MyPack;

- For a program to find MyPack, one of three things must be true.
  - Either the program can be executed from a directory immediately above **MyPack** **or**
  - the **CLASSPATH** must be set to include the path to **MyPack,** **or**
  - the -**classpath option** must specify the path to MyPack when the program is run via **java**
- To execute the program
  - java MyPack.programname

CLASSPATH9contd.)

- In the case of CLASSPATH and –classpath option , the class path *must not include* ***MyPack****, itself.* It must simply specify the *path to* ***MyPack.***
- Suppose the path of MyPack directory is **C:\MyPrograms\Java**\MyPack
  - Then the **class path** to MyPack is **C:\MyPrograms\Java**

Access Protection

- Addresses four categories of visibility for class members:
  - 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

Access Protection(contd.)

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection(contd.)

- A **non-nested class** has only two possible access levels:
  - default
  - public.
- When a class is declared as **public,** it is accessible by any other code.

```
public class A {////
}
```

- If a class has **default** access, then it can only be accessed by other code within its same package.

```
class B
{  }
```

- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the public class.

Importing Packages

- All of the standard classes are stored in some named package.
- If we want to use classes in some other packages, they must be *fully qualified with their package name or names*,. It is **difficult** to type in the long dot-separated package path name for every class we want to use.
  - **TO SOLVE THIS PROBLEM**, we can use **import** statement. The **import** statement helps to bring certain classes, or entire packages, into visibility.

- To use a class or a package from the library, we need to use the **import** keyword

- **import** statements is written **after** the package statement(if exists) and **before** all class definitions.

Importing Packages(contd.)

- 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 package pkg1 separated by a dot (**.**). Here square bracket denotes that it is optional.

- E.g.

```
import pack1;          // import the package pack1

import java.io.*;      // import all the classes from the package java.io

import java.util.Date; //import the Date class from the package java.util
```

Importing Packages(contd.)

- All of the standard Java classes included with Java are stored in a package called **java**
- The basic language functions are stored in a package inside of the java package called **java.lang**
  - it is implicitly imported by the compiler for all programs.

Importing Packages(contd.)

- Using an **import** statement:

```
import java.util.*;
class MyDate extends Date {
//statements , methods,variables
}
```

- Without the import statement looks like this:

```
class MyDate extends java.util.Date
{
}
```

Without Using import statement- we have to use class from other package as packagename.classname (fully quantified)


```
//Program A.java
package pack1;
public class A
 {
int a=100;
public int c=20;
protected int d=50;
    public void msg()
    {
System.out.println("Base class A Hello");
    }
}
```

```
//Program B.java
package pack2;
class B{
public static void main(String args[])
{
     pack1.A obj = new pack1.A();
    obj.msg();
    System.out.println("c="+obj.c);
//System.out.println("d="+obj.d);
// cannot access protected of
//different package i.e. pack1
    //System.out.println("a="+obj.a);
//cannot access private of other class
  }
}
```

Using **import package.*** statement to import all classes in pack1 to program file in pack2

//Program A.java

```
package pack1;
public class A
 {
int a=100;
public int c=20;
protected int d=50;
 public void msg()
    {
System.out.println("Base class A Hello");
    }
}
```

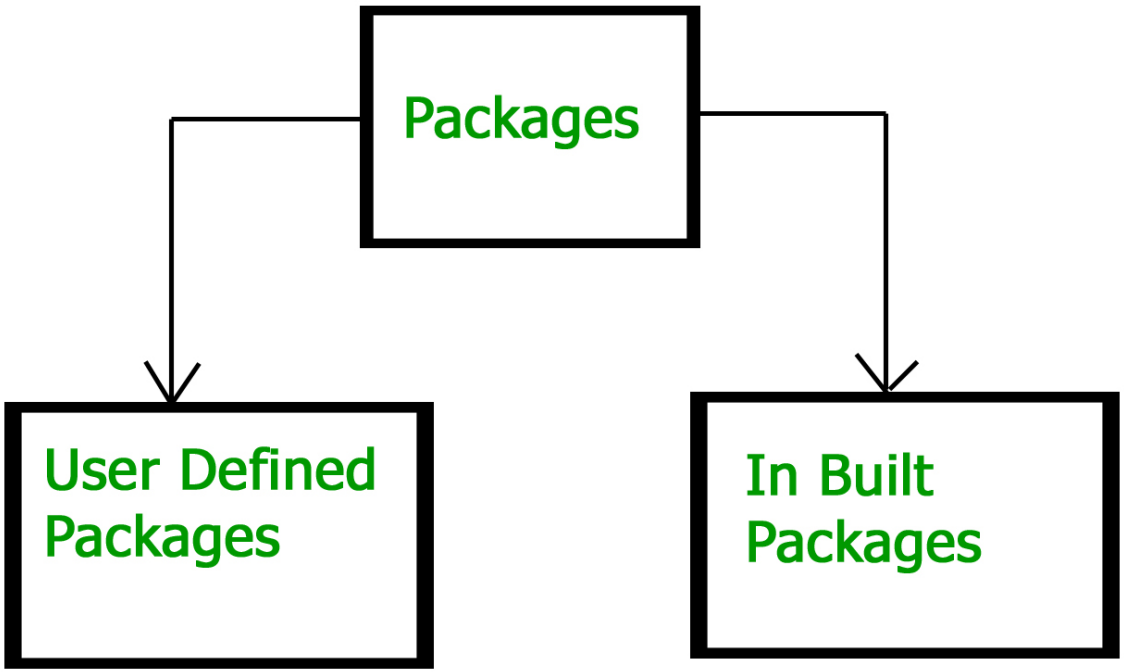
//Program B.java

```
package pack2;
import pack1.*;
class B{
public static void main(String args[])
{
      A obj = new A();
     obj.msg();
     System.out.println("c="+obj.c);
//System.out.println("d="+obj.d);
// cannot access protected of different package
     pack1
     //System.out.println("a="+obj.a);/
/ /cannot access private of other class
  }
}
```

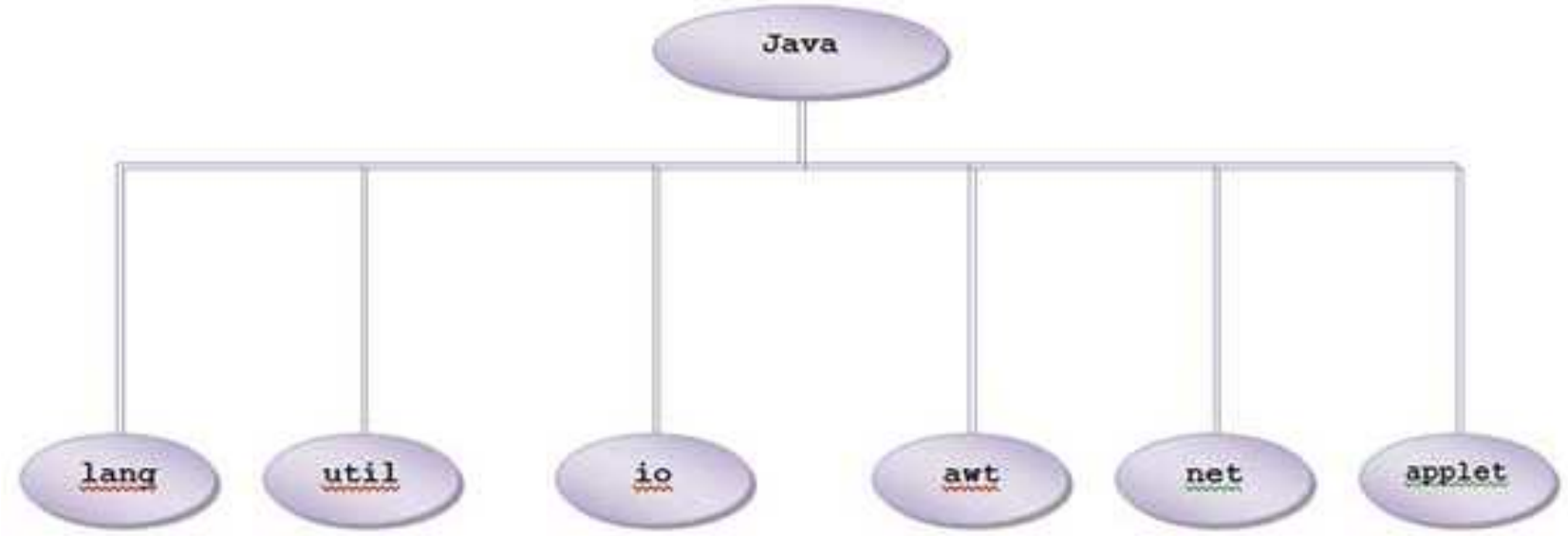
Using **import package.classname** statement to import class A in pack1 to program file in pack2


```
//Program A.java
package pack1;
public class A
 {
int a=100;
public int c=20;
protected int d=50;
 public void msg()
    {
System.out.println("Base class A Hello");
    }
}
```

```
//Program B.java
package pack2;
import pack1. A;
class B{
public static void main(String args[])
{
      A obj = new A();
     obj.msg();
     System.out.println("c="+obj.c);
//System.out.println("d="+obj.d);
// cannot access protected of different package
     pack1
     //System.out.println("a="+obj.a);/
/ /cannot access private of other class
  }
}
```


Packages
User Defined Packages
In Built Packages

Built-in Packages



Java Foundation Packages

- Java provides a large number of classes grouped into different packages based on their functionality.
- The six foundation Java packages are:
  - java.lang
    - Contains classes for primitive types, strings, math functions, threads, and exception
  - java.util
    - Contains classes such as vectors, hash tables, date etc.
  - java.io
    - Stream classes for I/O
  - java.awt
    - Classes for implementing GUI – windows, buttons, menus etc.
  - java.net
    - Classes for networking
  - java.applet
    - Classes for creating and implementing applets

- Steps and examples for creating and using packages

- Create a folder **pack1** inside D drive
- Create a file A.java

```
package pack1;
public class A
{
     public static void main(String args[] )
    {
    System.out.println("Hello");
    }
    public void show()
    {System.out.println(“show in A");
    }
}
```

Method 1

- Take ***path before pack1*** folder in command prompt here it s D drive.
- **Compile** using

  D:\>**javac pack1/A.java**

- **To run**

D:\>**java pack1/A**

Or

D:\>**java pack1.A**

Method 2

- Set classpath in command prompt to path to folder before the package pack1

C:\Users\USER>**set CLASSPATH=;D:**

To compile

C:\Users\USER>**javac -cp . D:\pack1\A.java**

To run

C:\Users\USER>**java pack1.A**

Hello

Method 3:

- Using –classpath option
- Compile using

C:\Users\USER>**javac D:\pack1\A.java**

Or

C:\Users\USER>**javac -classpath . D:\pack1\A.java**

- Run using

C:\Users\USER>**java** -classpath **D:\ pack1.A**

E.g using import statement

- Create a folder **pack2** inside D drive
- Create a file B.java in it

```
package pack2;
import pack1.*;
class B{
public static void main(String args[])
{
    A obj = new A();
    obj.show();
    System.out.println("main in class B");
}
}
```

```
D:\>javac pack1\A.java

D:\>javac pack2\B.java

D:\>java pack2.B
show in A
main in class B
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - **More features of Java (Part 2)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Introduction:
  - ☑ **Interfaces**.

Interface

- Interface can be created using the keyword **interface.**
- Interfaces are syntactically similar to classes.
- Interface **does not have instance variables**.
- The **methods** in interface are declared without any body.
  - **Interface** never implements methods.
- Any number of classes can implement an **interface.**
- One class can implement any number of interfaces.
  - This helps to achieve multiple inheritance.

Interface(contd.)

- To implement an interface,
  - a class must create and define the complete set of methods that are declared by the interface.
- Each class can have its own implementation of the methods.
- By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Interfaces support **dynamic method resolution at run time.**

Interface(contd.)

- General form of an interface:

```
accessspecifier interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

Interface(contd.)

- When **no access specifier** is included, then it has **default** access.
  - the interface is only available to other members of the package in which it is declared.
- The **methods** are declared have **no bodies**. They end with a semicolon after the parameter list.
- They are **abstract methods.**
- Each class that includes an interface must implement all of the methods.
- Variables re implicitly **final** and **static,** meaning they cannot be changed by the implementing class.
  - They must also be initialized.
- All methods and variables are implicitly **public**

Example


```
interface Callback {
void show(int param);
}
```

Implementing Interfaces

- After an interface has been defined, one or more classes can implement that interface.
  - For that include the **implements** clause in a class definition
- General form of a class that includes the **implements** clause

class classname [**extends** superclass] [**implements** interface [,interface...]]

{

// class-body

}

//square bracket denotes optional

- If a class implements more than one interface, the interfaces are separated with a comma
- When we implement an interface method, it must be declared as **public.**


```
interface Callback
{
void show(int param);
}
```

```
class Sample implements Callback
    {
public void show(int p)
    {
    System.out.println("show p= " + p);
    }
//other methods
}
```

Here **Callback** is an interface The class Sample implements that interface. So **Sample** class should define the method in **Callback** , show (int param)

Accessing Implementations Through Interface References

- We can declare variables as **object references** that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable

interfacename obj=object of implementing class;

```
interface Callback
{
void show(int param);
}
class Sample implements Callback
{
public void show(int p)
    {
    System.out.println("show p= " + p);
    }
//other methods
}
```

```
class Test{
public static void main(String args[])
    {
    Callback c = new Sample();
    c.show(42);
    }
}
```

Here c is an interface reference variable .It has only has knowledge of the methods declared by its **interface declaration.**

Partial Implementations

- If a class includes an interface but **does not fully implement the methods required by that interface**, then that **class** must be declared as **abstract**.

```
interface Callback {
void show(int param);
}
abstract class Incomplete implements Callback {
 int a, b;
void display()
{
System.out.println("display");
}}
```

>>Here the class Incomplete does not implement **show**() in **the interface Callback. So the class Incomplete is abstract class**

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a **member interface** or a **nested interface.**
- **A** nested interface can be declared as public, private, or protected.
  - The top level interface must either be declared as **public** or use the **default** access level.

Nested Interfaces

- If we want to use a ***nested interface outside of its enclosing scope***, the nested interface must be qualified by the name of the class or interface of which it is a member.

Nested Interfaces


```
class A {
        // this is a nested interface
         public interface NestedIF
         {
         boolean isNotNeg(int x);
         }
}
class B implements A.NestedIF {
 public boolean isNotNeg (int x)
{
 return x < 0 ? false: true;
 }
}
```

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

 A.NestedIF nif = new B();
 if(nif.isNotNeg(10))
 System.out.println("10 is not negative");
}

}
```

Applying Interfaces

Variables in Interfaces

- When we include an interface in a class (using "implement" the interface), all of those **variable** names in the interface will be in scope as **constants.**
  - That is they are imported to class name space as **final** variables.
-

Interfaces Can Be Extended


```
import java.util.Random;
interface Interf {
int NO = 0;
int YES = 1;
}
class Question implements Interf
{
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 50)
return NO; // 30%
else
return YES;
}
}
```

```
class AskMe implements Interf
{
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break; } }
public static void main(String args[])
    {
Question q = new Question();
answer(q.ask()); }
}
```

Interfaces Can Be Extended

- One *interface can inherit another* by use of the keyword **extends.**
- When a class *class1* implements an interface *interface1* that inherits another interface *interface2*, then *class1* must provide implementations for all methods defined within the interface inheritance chain(both *interface1* and *interface2).*

// One interface can extend another-E.g.


```
.
interface A {
void meth1();
void meth2();
}
interface B extends A {
void meth3();
}
class MyClass implements B
 {
public void meth1()
 {
System.out.println("Implement
    meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");}
}

class IFExtend {
public static void main(String arg[])
 {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - **More features of Java (Part 3)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **More features of Java :**
  - **Input / Output:**
    - I/O Basics
    - Reading Console Input
    - Writing Console Output
    - PrintWriter Class

I/O Basics

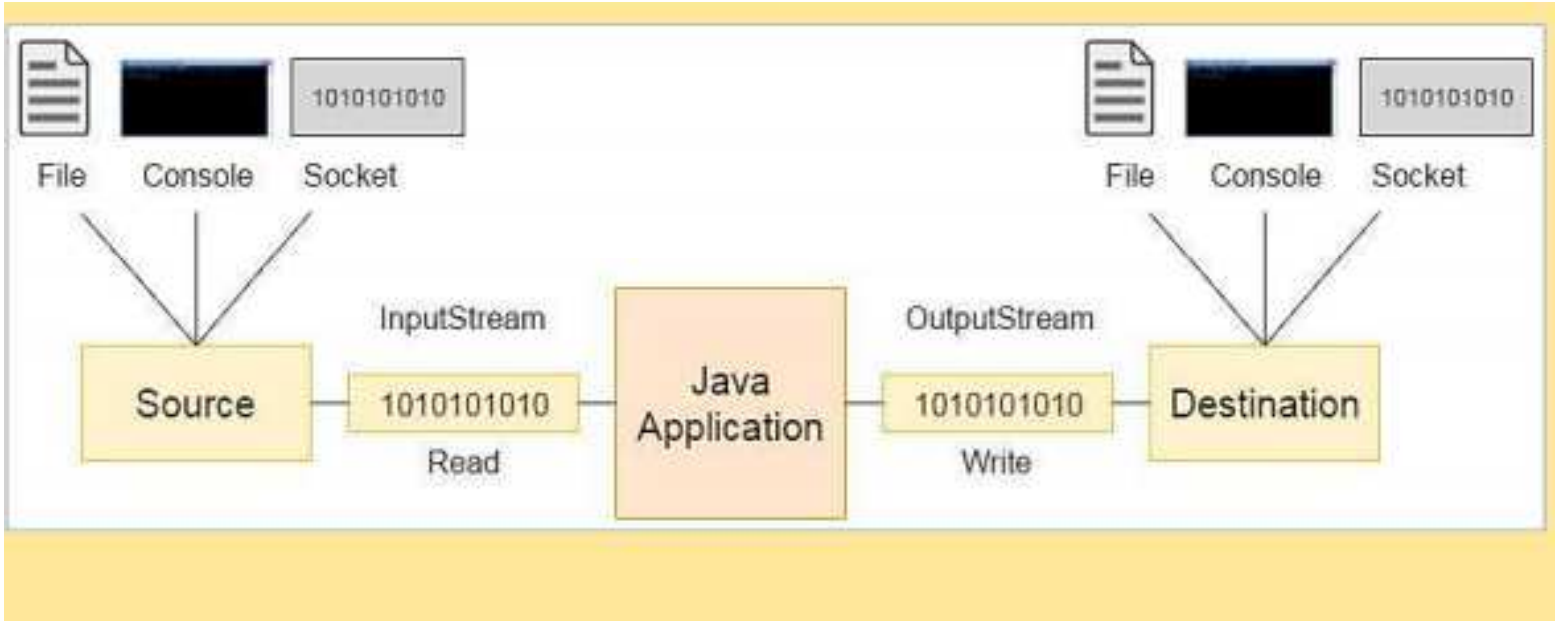
- Only **print( ) and println( )** are used frequently. All other I/O methods are not used significantly.
  - Because most real applications of Java are not text-based, console programs.
- Java's support for console I/O is limited.

Streams

- Java programs perform I/O through **streams**.
- A stream is an abstraction that either **produces or consumes** information.
- A stream is a sequence of objects that supports various methods.
- A stream is linked to a physical device by the Java I/O system.
  - **Input stream** may refer to different kinds of input:
    - from a disk file, a keyboard, or a network socket
  - **Output stream** may refer to
    - the console, a disk file, or a network connection.

Working of Javava I/O stream

- Stream is like a flow of data.



Streams

- The java.io package contains all the classes required for input and output operations.
- Java defines two types of streams: **byte** and **character**.
- ***Byte streams*** provides a means for handling input and output of bytes.
  - Byte streams are used when reading or writing binary data.
- ***Character streams*** provide a means for handling input and output of characters.
  - they use Unicode.
  - in some cases, character streams are more efficient than byte streams.

ByteStream classes

- Byte streams are defined by using two class hierarchies.

  At the top are two abstract classes:

  - **InputStream** and **OutputStream.**

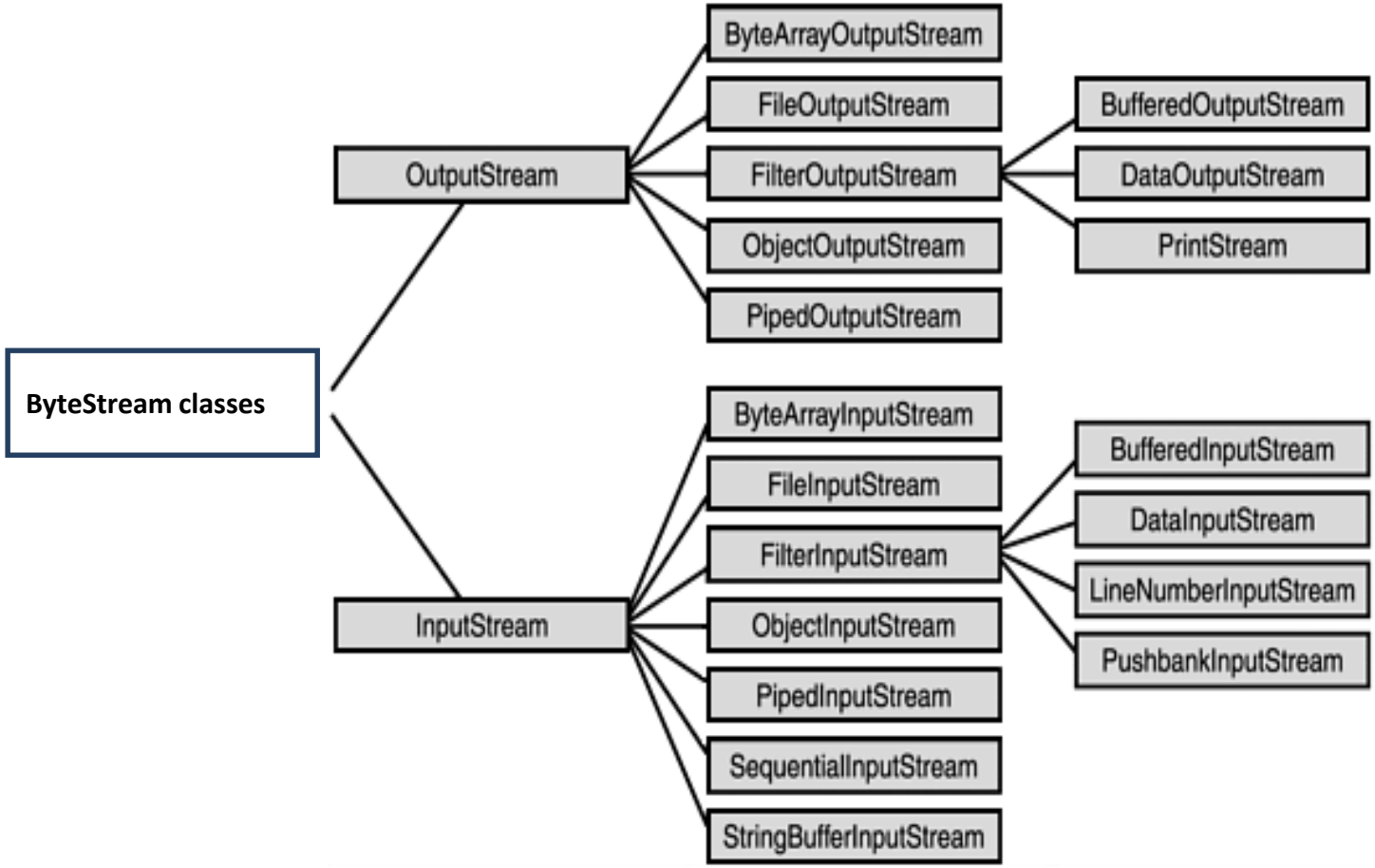
- Each of these abstract classes has several concrete subclasses

  - that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

- Two of the most important are **read( ) and write( ),**
  - These methods are overridden by derived stream classes.

ByteStream classes

- **ByteStream classes**



Byte Stream I/O classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

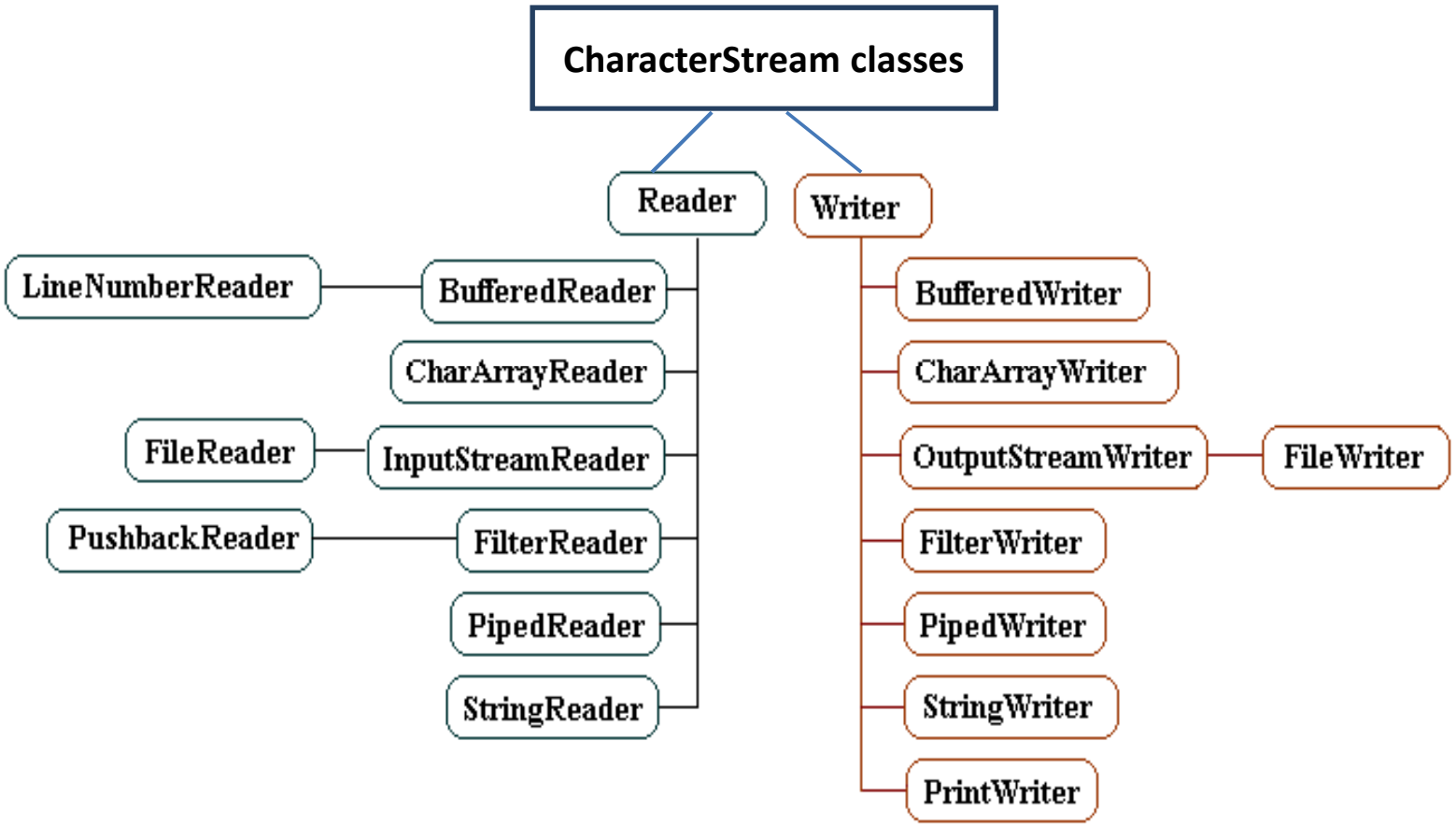
Character streams

- Character streams are defined by using two class hierarchies.

  At the top are two abstract classes,

  - **Reader** and **Writer.**

- Java has several concrete subclasses of each of these.
- Two of the most important methods are **read( )** and **write( ).**
  - These methods are overridden by derived stream classes.


CharacterStream classes
Reader
Writer
LineNumberReader
BufferedReader
CharArrayReader
FileReader
InputStreamReader
PushbackReader
FilterReader
PipedReader
StringReader
BufferedWriter
CharArrayWriter
OutputStreamWriter
FileWriter
FilterWriter
PipedWriter
StringWriter
PrintWriter

Character Stream I/O classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams

- All Java programs automatically import the **java.lang** package. This package defines a class called System.
- **System** contains three **predefined stream variables**:
  - **in, out,** and **err**.
  - These fields are declared as **public, static,** and **final** within System.
- **System.out** refers to the standard output stream.
- **System.in** refers to standard input, which is the keyboard by default.
- **System.err** refers to the standard error stream, which is the console by default.
- System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.

Reading Console Input

- The preferred method of reading console input is to use a **character-oriented stream.**
- In Java, console input is accomplished by reading from **System.in.**
  - To obtain a character based stream that is attached to the console, wrap System.in in a BufferedReader object.

Reading Console Input(contd.)

- BufferedReader supports a **buffered input stream.**
  - Its most commonly used constructor is:

  BufferedReader(Reader inputReader)

- Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class.

Reading Console Input(contd.)

- One of the concrete subclasses of Reader is **InputStreamReader.**
- **InputStreamReader** converts bytes to characters.
  - It reads bytes and decodes them into characters using a specified charset.
- To obtain an InputStreamReader object that is linked to System.in, the constructor that can be used is :

InputStreamReader(**InputStream** inputStream)

- Following line of code creates a **BufferedReader** that is connected to the keyboard:

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

- By wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedReader**, we can read input from the user in the command line.
- After this statement executes, **br** is a character-based stream that is linked to the console through System.in

Reading console-summary

To accept data from keyboard, we use **System.in. (bytestream)**

We need to **connect keyboard** to an **input stream object**.

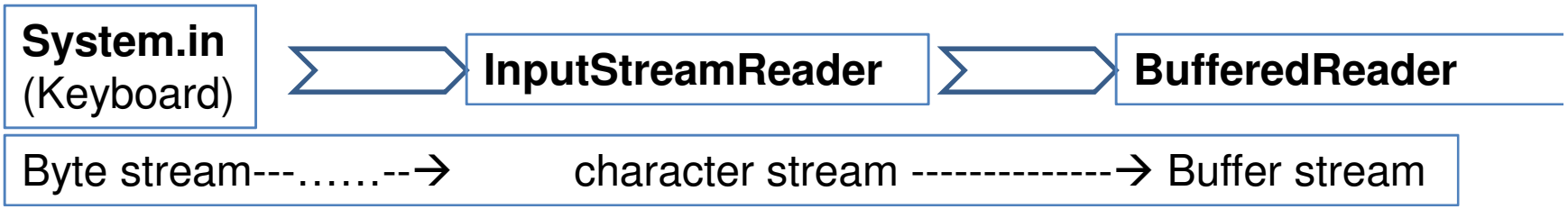
Here we can use InputStreamReader that can read data from the keyboard (**convert byte stream to character stream**)

Now our data reaches InputSreamReader..

BufferedReader class is used to read the text from a character-based input stream.

To make program run fast and to make reading efficient , ***buffering can be done using BufferedReader class.*** It can read data from stream.

Create BufferedReader object and **connect InputStreamReader object to it**




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

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

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

- **System.in** → keyboard(Byte stream)
- Convert **byte steam** to **character stream** using **InputStreamReader**.
- Then wrap that character stream in a **buffered stream BufferedReader**

Reading Characters

- To read a character from a **BufferedReader, use read( ).** The version of read( ) is

  int **read( )** throws **IOException**

- Each time that **read( )** is called, it **reads a character** from the input stream and **returns it as an integer value**. It returns –1 when the end of the stream is encountered

Reading Characters(E.g.)


```
import java.io.*;
class Readinp
{
public static void main(String a[]) throws IOException
{
char c;
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter a letter");
c=(char)br.read();
System.out.println("Letter="+c);


}
}
```

OUTPUT

```
Enter a letter
a
Letter=a
```

Enter characters one by one when you type q it will stop reading


```
class BRRead {
public static void main(String args[])
throws IOException
{
char c;
BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
    c = (char) br.read();
    System.out.println(c);
    } while(c != 'q');
}
}
```

OUTPUT

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

No input is actually passed to the program until you
press ENTER.

Reading Strings

- To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader class.**
- Its general form is

String readLine( ) throws IOException

This returns a **String** object.

Reading Strings(E.g.)


```
import java.io.*;
class Readinp
{
public static void main(String a[]) throws IOException
{
char c;
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter a line of text");
c=(char)br.read();
s=br.readLine();
System.out.println("Line is "+s);
}
}
```

OUTPUT

```
Enter a line of text
How are you
Line is How are you
```

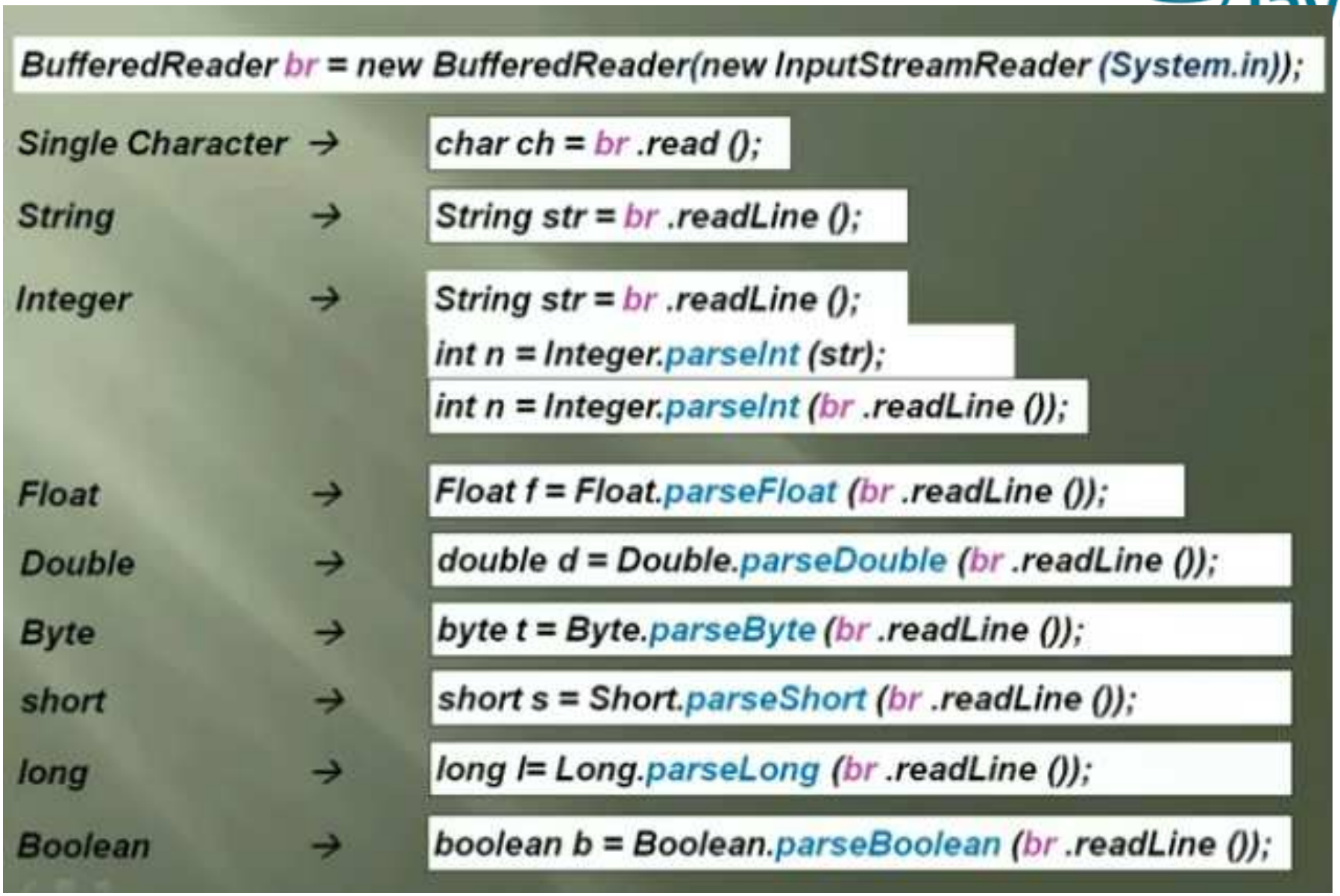
Enter lines of text one by one when you type stop it will stop reading


```
import java.io.*;
class Readlinetillstop
{
public static void main(String a[]) throws IOException
{
String s[] = new String[100];

BufferedReader br=new BufferedReader(new  InputStreamReader(System.in));
System.out.println("Enter a line of text");
System.out.println("Enter 'stop' to quit.");
for(int i=0; i<100; i++)
{
s[i] = br.readLine();
if(s[i].equals("stop")) break;
}
System.out.println("Lines are ");
for(int i=0; i<100; i++)
{

if(s[i].equals("stop")) break;
System.out.println(s[i]);
}  }}
```

OUTPUT

```
Enter a line of text
Enter 'stop' to quit.
what
how are you
ok
Stop
Lines are
what
how are you
ok
```

Read console inputs



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

Single Character →	`char ch = br .read ();`
String →	`String str = br .readLine ();`

Integer →	`String str = br .readLine ();`
	`int n = Integer.parseInt (str);`
	`int n = Integer.parseInt (br .readLine ());`

Float →	`Float f = Float.parseFloat (br .readLine ());`
Double →	`double d = Double.parseDouble (br .readLine ());`
Byte →	`byte t = Byte.parseByte (br .readLine ());`
short →	`short s = Short.parseShort (br .readLine ());`
long →	`long l= Long.parseLong (br .readLine ());`
Boolean →	`boolean b = Boolean.parseBoolean (br .readLine ());`

Writing Console Output

- Console output is usually done through **print( ) and println().**
- These methods are defined by the class **PrintStream.**
  - It is the type of object **referenced by System.out.**
  - **System.out** is a byte stream,
- **PrintStream** is an output stream derived from **OutputStream**,
  - **PrintStream** also implements the low-level method write( ).
  - write( ) can be used to write to the console.

Writing Console Output

- The simplest form of write( ) defined by PrintStream is

  void write(int *byteval)*

  - This method writes the byte specified by *byteval* to the stream
  - *byteval is declared* as an integer, only the low-order eight bits are written.

```
// Demonstrate System.out.write(). Write letter ‘A’ to console.
class WriteDemo {
public static void main(String args[]) {
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

OUTPUT
A

PrintWriter class

- For real-world programs, the recommended method of writing to the console using Java is through a **PrintWriter stream.**
- **PrintWriter** is one of the **character-based classes.**
- **PrintWriter** defines several constructors.

  PrintWriter(**OutputStream** *outputStream,* ***boolean*** *flushOnNewline)*

- Here, *outputStream* is an object of type OutputStream, and *flushOnNewline* controls whether Java flushes the output stream every time a println( ) method is called.
  - If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.

- **PrintWriter** supports the **prin)** and **println( ) methods**
- If an argument is not a simple type, the **PrintWriter methods** call the object's **toString( )** method and then print the result.
- To **write to the console** by using a **PrintWriter,** specify **System.out** for the output stream and flush the stream after each new line.

PrintWriter pw = new **PrintWriter**(System.out, **true**);

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[])
    {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
    }
}
```

OUTPUT

```
This is a string
-7
4.5E-7
```

System.out

- System.out is a byte stream.
- **System.out** refers to the standard output stream(monitor).
- **System:** It is a final class defined in the java.lang package.
- **out:** This is an instance of PrintStream type, which is a public and static member field of the System class.

PrintWritter

- **PrintWriter** should be used to write a stream of characters
- PrintWriter is a subclass of **Writer** (character stream class)
- It is used in real world programs to make it easier to internationalize the program

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 4)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

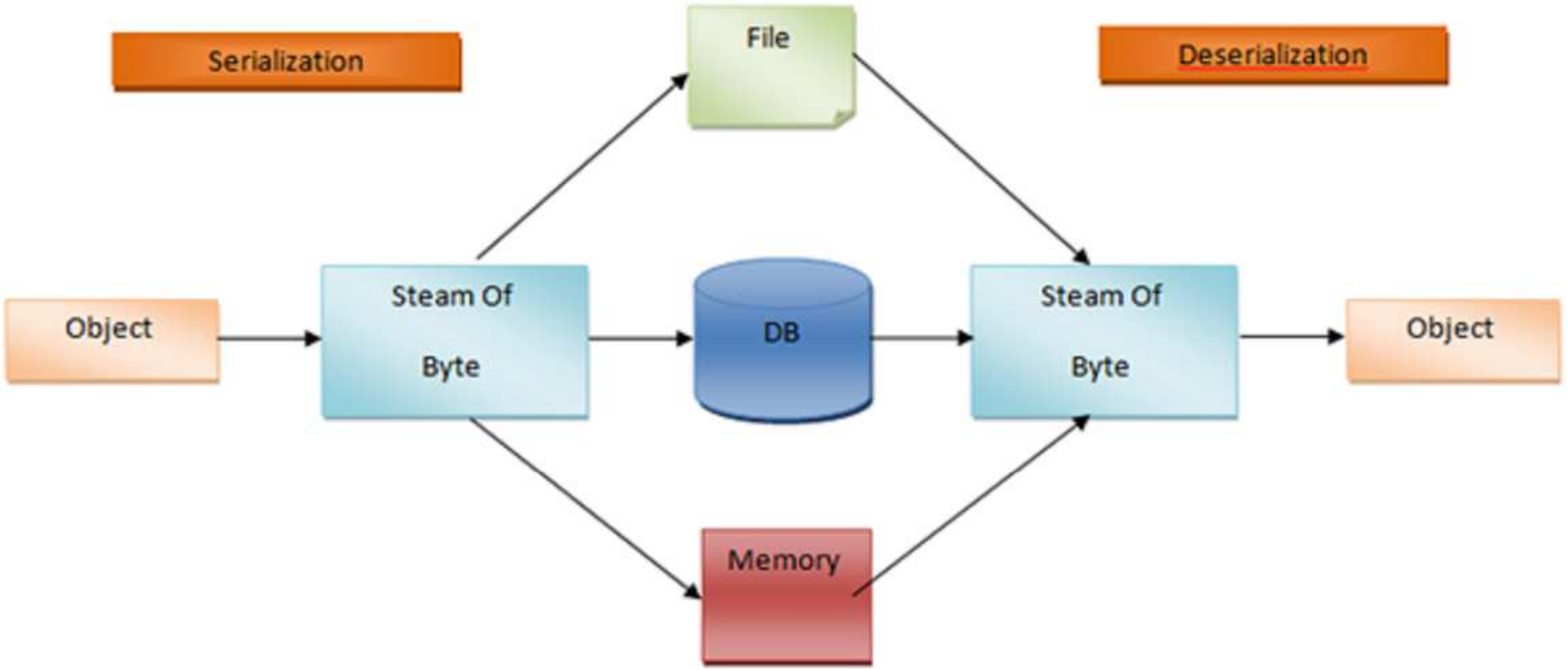
- **More features of Java :**
  - **Input / Output:**
    - ✓Object Streams and Serialization

Object Streams and Serialization

- **Object streams** support I/O(input-output) of objects
- The object stream classes are
  - **ObjectInputStream**
  - **ObjectOutputStream**

Serialization

- **Serialization** is the process of writing(converting) the state of an object to a byte stream.
  - This is useful when we want to
    - save(store) the state of the program to a ***persistent(permanent)*** storage area, such as a file or
    - when we want to send it over network.
- Later we can restore these objects by using the process of ***deserialization***.
  - **Deserialization c**onverts byte streams into object.


Serialization
File
Deserialization
Object
Steam Of
Byte
DB
Steam Of
Byte
Object
Memory

Serialization(contd.)

- Serialization is also needed to implement Remote Method Invocation (RMI).
  - RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
  - The ***sending machine*** serializes the object and transmits it. The ***receiving machine*** deserializes it.

Serialization(contd.)

- If we attempt to serialize an object at the top of an object graph,
  - all of the other referenced objects are recursively located and serialized.
- Similarly, during the process of deserialization, all of these objects and their references are correctly restored when deserialization is done at the top.

Serialization(contd.)

- Interfaces and classes that support serialization are:
  - **Serializable**
  - **Externalizable**

Serialization(contd.)

- **Serializable**
  - Only an object that implements the **Serializable interface** can be **saved and restored** by the serialization facilities.
  - The **Serializable interface** defines no members**.**
  - It is simply used to indicate that a class may be serialized.
  - If a class is serializable, all of its subclasses are also serializable.
  - Variables that are declared as **transient** and **static variables** are not saved by the serialization facilities.

Externalizable

- Much of the work to save and restore the state of an object occurs automatically.
  - The programmer may need to have control over these processes.
  - it may be desirable to use compression or encryption techniques.
- The **Externalizable interface** is designed for these situations.
- The **Externalizable interface defines** two methods:

void **readExternal**(ObjectInput *inStream)* throws IOException, ClassNotFoundException

void **writeExternal**(ObjectOutput *outStream)* throws IOException

- *inStream* is the byte stream from which the object is to be read
- *outStream* is the byte stream to which the object is to be written

ObjectOutput

- The **ObjectOutput interface** extends the **DataOutput interface** and supports object serialization**.**
- It defines the methods such as **writeObject( )**
- **writeObject( ) method** is called to **serialize** an object.
- All of these methods will throw an **IOException** on error conditions

ObjectOutput-methods

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

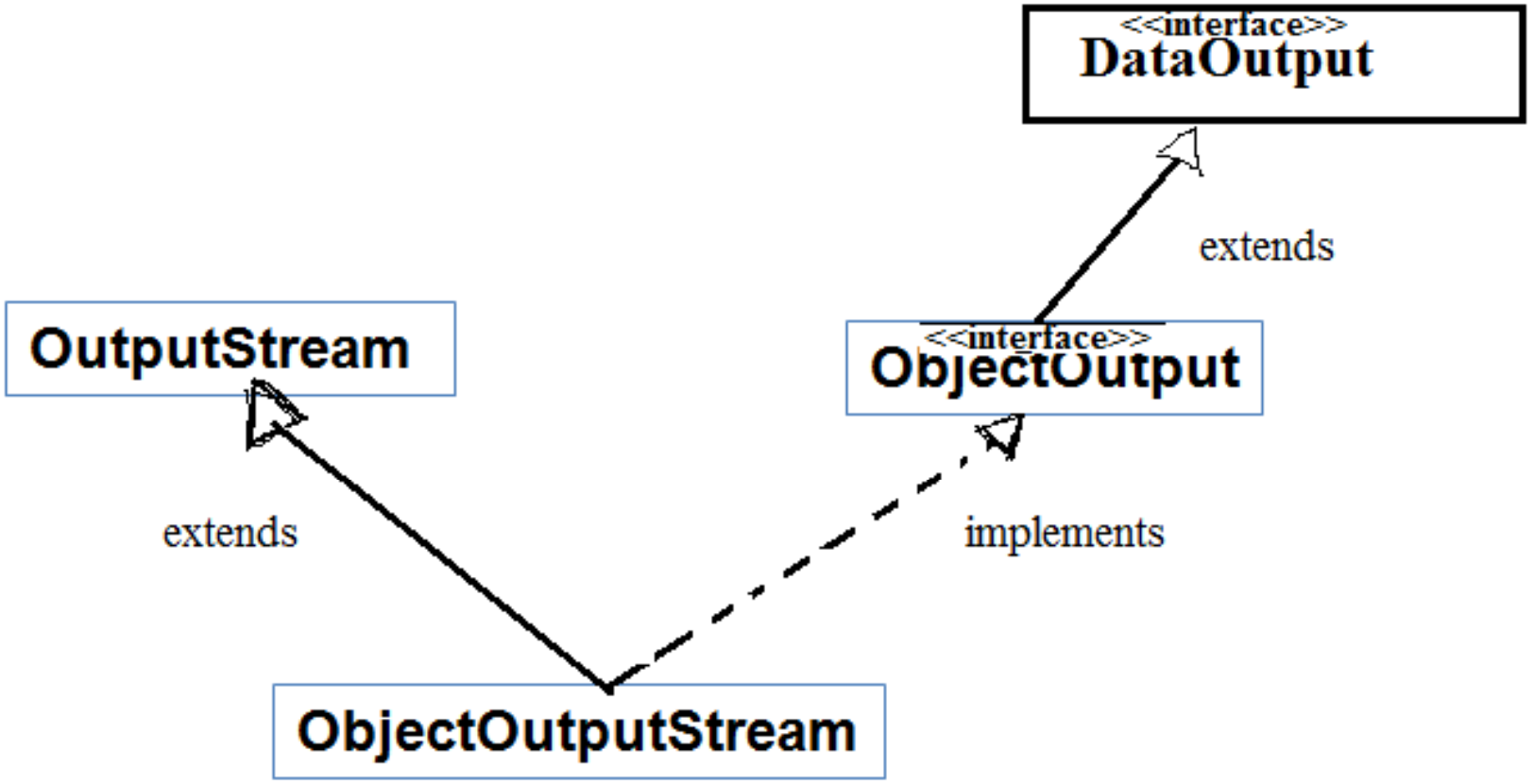
The Methods Defined by **ObjectOutput**

ObjectOutputStream

- The **ObjectOutputStream class extends** the **OutputStream class** and **implements** the **ObjectOutput interface.**
- It is responsible for **writing objects to a stream.**
- A constructor of this class is

ObjectOutputStream(OutputStream *outStream) throws IOException*

  - The argument outStream is the output stream to which serialized objects will be written.
- Methods in this class will throw an **IOException** on error conditions.
- There is also an inner class to **ObjectOuputStream** called **PutField.**
  - It facilitates the writing of persistent fields.


<<interface>>
DataOutput
extends
OutputStream
<<interface>>
ObjectOutput
extends
implements
ObjectOutputStream

ObjectOutputStream-methods

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

ObjectInput

- The **ObjectInput interface** extends the **DataInput** interface and defines the method such as **readObject( )** method.
- This is called to **deserialize** an object.
- All of these methods will throw an **IOException** on error conditions.
- The **readObject( )** method can also throw **ClassNotFoundException**

ObjectInput-methods

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

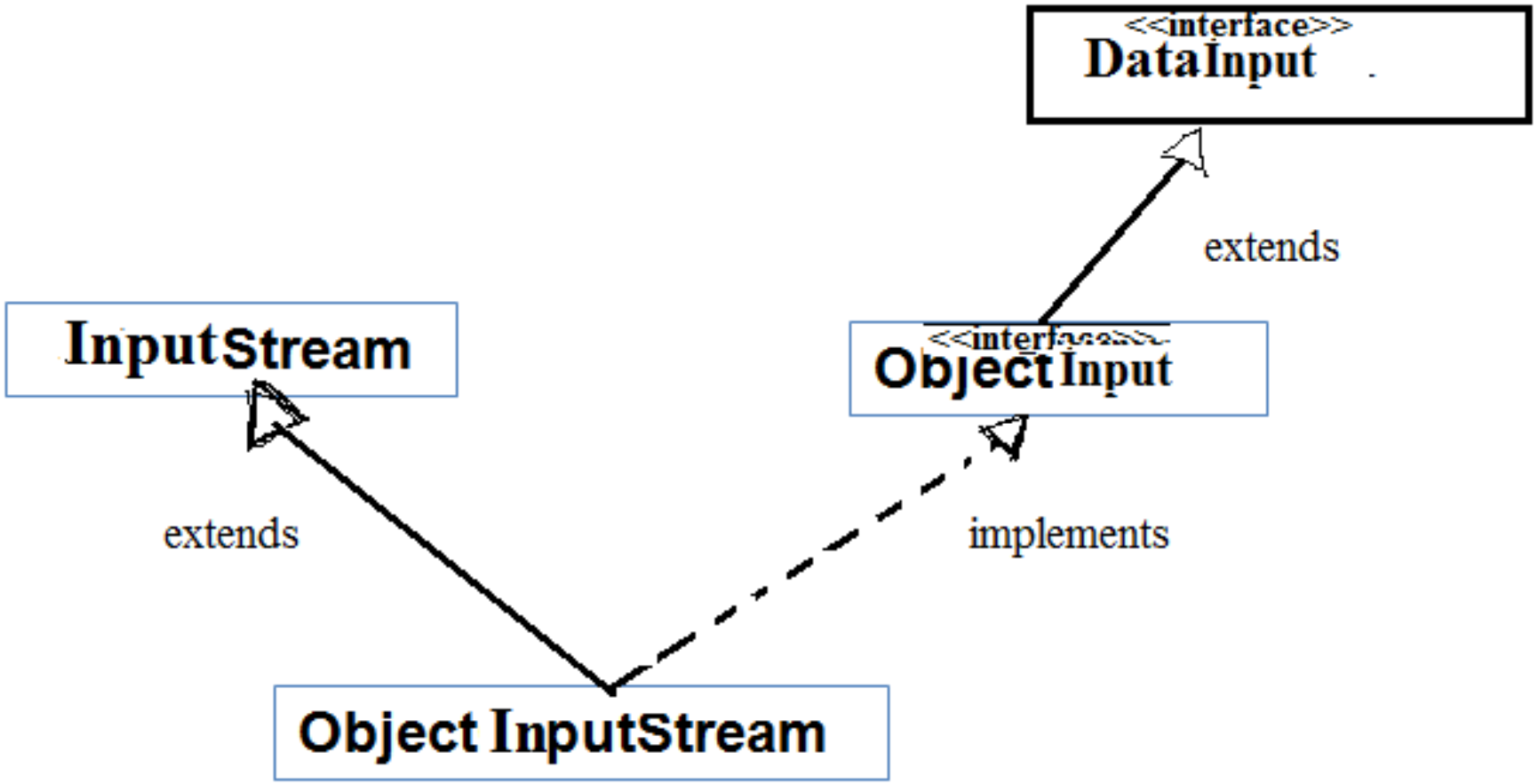
ObjectInputStream

- The **ObjectInputStream class extends the InputStream class** and **implements** the **ObjectInput** interface.
- **ObjectInputStream** is responsible for **reading objects from a stream.**
- A constructor of this class is

ObjectInputStream(InputStream *inStream)* throws IOException

  - The argument *inStream is the input stream from which serialized objects should be read.*

- The meThods will throw an **IOException** on error conditions.
- The **readObject() method** can also throw **ClassNotFoundException.**
- There is also an inner class to ObjectInputStream called **GetField.** It facilitates the reading of persistent fields


<<interface>>
DataInput
extends
InputStream
<<interface>>
ObjectInput
extends
implements
Object InputStream

ObjectInputStream-methods

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 5)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **More features of Java :**
  - Working with **Files**

Working with Files

- In Java, all files are **byte-oriented**.
- Java provides methods to
  - read bytes from a file and
  - write bytes to a file.

Working with Files(contd.)

- Two of the most often-used file stream classes are

  - **FileInputStream**

    - FileInputStream is an input stream to **read data** from a file in the form of sequence of bytes

  - **FileOutputStream**

    - **FileOutputStream** class is an output stream for **writing data** to a file

Working with Files- OPEN a file

- **To open a file,**
  - create an object of one of these classes
  - specify the name of the file as an argument to the constructor.
- If we want to open a file for **reading**
  - Create object of FileInputStream class
- If we want to open a file for **writing**
  - Create object of FileOutputStream class

Working with Files(OPEN a file contd.)

- Main constructors are

FileInputStream(String *fileName*) throws FileNotFoundException

FileOutputStream(String *fileName*) throws FileNotFoundException

- Here, *fileName* specifies the name of the file (as String i.e. enclose in double quotes) that we want to open.
- When we create an **input stream**, if the file does not exist, then **FileNotFoundException** is thrown**.**
- For **output streams**, if the file cannot be created, then **FileNotFoundException is thrown.**
  - When an ***output file*** is opened, any file that is already existing with the same name as output file is destroyed.

Working with Files(OPEN a file) contd.

- To open a file for **reading-**

  We have to create **FileInputStream** class object and pass *filename* as the parameter to the constructor.

E.g. to open the file Sample.txt for reading

```
FileInputStream fileobject;
fileobject = new FileInputStream("Sample.txt");
```

Working with Files(OPEN a file) contd.

- To open a file for **writing**

  We have to create **FileOutputStream** class object and pass *filename* as the parameter to the constructor.

E.g. to open the file Sample.txt for writing

```
FileOutputStream fileobject;
fileobject = new FileOutputStream("Sample.txt");
```

Working with Files(closing a file)

- After completing file read or write operations, we should **close** the file by calling **close( ).**
- It is defined by both **FileInputStream** and **FileOutputStream :**

void **close( )** throws IOException

Working with Files(closing a file) contd.

E.g. to close file Sample.txt opened for reading

```
FileInputStream fileobject;

fileobject = new FileInputStream(“Sample.txt");

//statements for reading the file

fileobject.close();
```

Working with Files(read a file)

- To **read data** from a file,

  1. First, we have to create **FileInputStream** class object and pass *filename* as the parameter to the constructor.

     E.g.
     ```
     FileInputStream fileobject;
     fileobject = new FileInputStream("Sample.txt");
     ```

  2. Next,we can use a version of **read( )** that is defined within **FileInputStream.**

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

     E.g.
     ```
     int c=fileobj.read();
     ```

- Each time read() called, it reads a single byte from the file and returns the byte as an integer value.
  - **read( )** returns –1 when the **end of the file** is encountered.
  - read() can throw an **IOException.**

Read contents from file(E.g)

Write a program to read &display the contents in the file Sample.txt

```
import java.io.*;
class Readfile
{
public static void main(String arg[]) throws IOException
{
FileInputStream f;
try
{
f= new FileInputStream(“Sample.txt");
int c;
    do    {
          c=f.read();
          if(c!=-1)
          {System.out.print((char)c);}
          } while(c!=-1);
}
```

```
catch(FileNotFoundException e)
{
System.out.println("File not found");
return;
}
f.close();

}
}
```

Working of file read program

- In the above program, to read the file, an object pf **FileInputStream** class is created.

FileInputStream f;

f= new FileInputStream(“Sample.txt");

- Here the argument of the constructor in FileInputStream is the name of the file to be read. Here “Sample.txt”
- The following statement **read one byte** from the file and store in integer variable c

  c=f.read();

Working of file read program(contd.)

- The following statement **converts the integer variable c into character** using char(c) and print that character on the output screen (console)

  System.out.print((char)c);

This continues until c is equal to -1(end of file)

Working of file read program(contd.)

- ***Exceptions***(run time error) like FileNtotFoundException (if the given filename is not in the system path) may occur during file operations.
  - So it is better to **enclose file operation statements within try block** for handling exceptions.
- If the file we try to read a file that does not exist, then that exception is caught by the following catch block and the corresponding action in it is done.

```
try{
   //File Operation statements
   }
   catch(FileNotFoundException e)
   {
   System.out.println("File not found");  //print this message if file is not found
   }
```

Working with Files(write to a file)

- To write to a file, we can use the **write( )** method defined by **FileOutputStream.**

```
void write(int byteval) throws IOException
```

  - This method **writes the byte** specified by *byteval to the file.*
  - *Although byteval is declared as* an integer, **only the low-order eight bits are written** to the file.
  - If an error occurs during writing, an **IOException** is thrown

Steps to write data to a file

- To **write data** from a file,
  1. First, we have to create **FileOutputStream** class object and pass *filename* as the parameter to the constructor.
  2. Using write function store the byte value in file

E.g. int c=65;

FileOutputStream fileobject;

fileobject = new **FileOutputStream**("Sample.txt");

fileobject.**write**(*c*);

- Here lower order will be stored.So this will store ASCII value of 65 that is letter A in file Sample.txt

FILE COPY –copy contents from *test.txt* to *cp.txt*


```
import java.io.*;
class Rdfcopy
{ public static void main(String a[]) throws IOException
{
FileInputStream f1=null;
FileOutputStream f2=null;
try
{
f1= new FileInputStream("test.txt");
f2= new FileOutputStream("cp.txt");
int c;
do
{
c=f1.read();
if(c!=-1)
{
f2.write((char)c);
System.out.print((char)c);
}
}while(c!=-1);
}
catch(FileNotFoundException e)
        {
System.out.println("File not found");
return;    }
f1.close();
f2.close();
}
}
```

Working of file copy program

- For reading a file, FileInputStream object need to created.
  - Here f1

```
FileInputStream f1=null;
f1= new FileInputStream("test.txt");
```

- For writing to a file, FileOutputStream object need to created
  - Here f2

```
FileOutputStream f2=null;
f2= new FileOutputStream("cp.txt");
```

Working of file copy program(contd)

```
c=f1.read();
if(c!=-1)
{
f2.write((char)c);
System.out.print((char)c);
}
```

This means that integer **f1.read()** reads a single byte from file pointed by f1(test.txt) and store in integer variable c.

If c is not -1 (end-of-file) then c is converted int character using(char) casting.

f2.write((char)c);

This statement writes the character equivalent of c into file pointed by f2(cp.txt)

This continues until c==-1

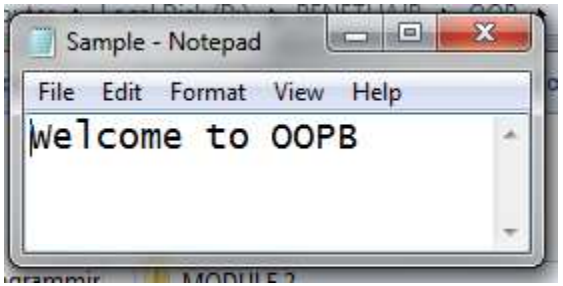
FILE READ - file name given as command line argument

Execution: java Readcommandline test.txt

```
import java.io.*;
class Readcommandline
{
public static void main(String arg[]) throws IOException
{
FileInputStream f;
        try
        {
        f= new FileInputStream(arg[0]);
        int c;
                do
                {
                c=f.read();
                if(c!=-1)
                    {System.out.print((char)c);}
                }while(c!=-1);
        }
        catch(FileNotFoundException e)
        {
        System.out.println("File not found");
        return;
}
f.close();}}
```

Example

```
import java.io.*;
 class Writesentencefile
{
public static void main(String arg[]) throws IOException
{
FileOutputStream f;
String s;
try
{
f= new FileOutputStream("Sample.txt");
      s="Welcome to OOP";
byte b[]=s.getBytes();          //converting string into byte array
f.write(b);

 f.write(66);        // write lower bytes. Here we will get ASCII vlue of 66 i.e. letter  B

}
catch(FileNotFoundException e)
{
System.out.println("File not found");
return;
}
f.close();
} }
```



FileReader

- The **FileReader** class creates a **Reader** that we can use to read the contents of a file.
- Its two most commonly used constructors are shown here:

  FileReader(String *filePath)*

  FileReader(File *fileObj)*

  – They can throw a **FileNotFoundException.** Here, ***filePath*** *is the full path name of a file, and* ***fileObj*** *i*s a File object that describes the file.
- The following example shows how to. It reads its own source file, which must be in the current directory.

Read lines from a file and print these to the standard output stream using FileReader


```
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws IOException
{
    FileReader fr = new FileReader("Sample.txt");
    BufferedReader br = new BufferedReader(fr);
    String s;
    while((s = br.readLine()) != null)
     {
          System.out.println(s);
    }
fr.close();
}
}
```

FileWriter

- **FileWriter** creates a Writer that you can use to write to a file.
- Its most commonly used constructors are:

  FileWriter(String *filePath)*

  FileWriter(String *filePath, boolean append)*

  FileWriter(File *fileObj)*

  FileWriter(File *fileObj, boolean append)*

  – They can throw an **IOException.** Here, ***filePath*** is the full path name of a file, and ***fileOb***j is a File object that describes the file. If append is true, then output is appended to the end of the file.

FileWriter(contd.)

- **FileWriter** will create the file before opening it for output when you create the object.
  - In the case where we attempt to open a read-only file, an **IOException will be thrown.**
- **getChars( )** method is used to extract the character array equivalent.

public void getChars(int srhStartIndex, int srhEndIndex, char[] destArray, int destStartIndex)

Parameters:

srhStartIndex : Index of the first character in the string to copy.

srhEndIndex : Index after the last character in the string to copy. **destArray** : Destination array where chars wil get copied.

destStartIndex : Index in the array starting from where the chars will be pushed into the array. **Return:** It does not return any value.

Write a string to file using FileWriter

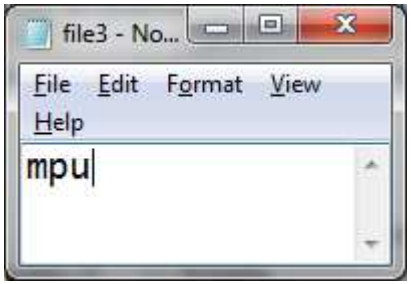

```
import java.io.*;
class FileWriterSimple
{
public static void main(String args[]) throws IOException
{
String source = "Welcome to OOP class\n" + " Study well";
char buffer[] = new char[source.length()]; // allocate space equal to length of string
source.getChars(0, source.length(), buffer, 0);
//copy the characters from position 0 to whole length(end) to buffer at position 0.
FileWriter f1 = new FileWriter("file1.txt");
f1.write(buffer);
f1.close();
}
}
```

Append a string to file using FileWriter


```
import java.io.*;
class FileWriterSimple
{
public static void main(String args[]) throws IOException
{
String source = "Welcome to OOP class\n" + " Study well";
char buffer[] = new char[source.length()]; // allocate space equal to length of string
source.getChars(0, source.length(), buffer, 0);
    //copy the characters from position 0 to whole length(end) from source
    //to buffer at /position 0.
FileWriter f1 = new FileWriter("file1.txt“,true); //append the contents
f1.write(buffer);
f1.close();
}
}
```

Write the alternate letter from string to file1.txt.
Write whole string to file2.txt,
Write the string starting from index 2 and upto 5 letters into file3.txt

```
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws IOException {
String source = "Computers";
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer, 0);
FileWriter f0 = new FileWriter("file1.txt");
for (int i=0; i < buffer.length; i += 2) {
f0.write(buffer[i]);      //Write letters at position 0,2,4,6…. ino file1.txt
}
f0.close();
FileWriter f1 = new FileWriter("file2.txt");
f1.write(buffer);         //write all contents in buffer in file2.txt
f1.close();
FileWriter f2 = new FileWriter("file3.txt");
f2.write(buffer,2,3);     //Write letters from 2nd position to three letters
f2.close();
}
}
```



Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 6)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **More features of Java :**
  - ☑ **Exception Handling:**
    - **Checked Exceptions**
    - **Unchecked Exceptions**
    - ***try*** Block and ***catch Clause***

Exception Handling

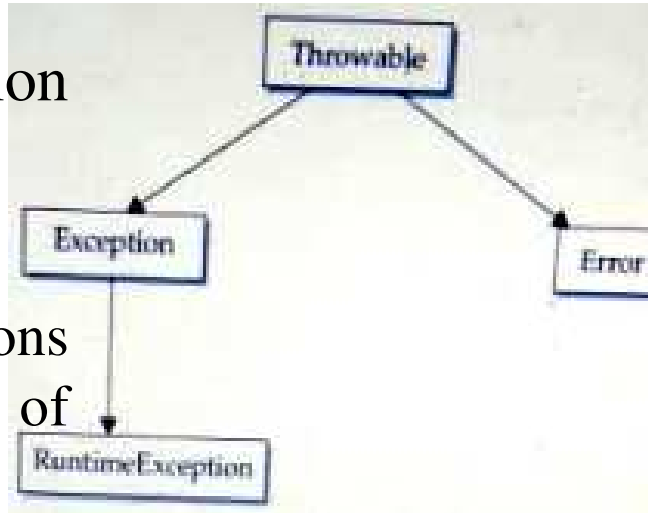
- An ***exception*** *is* an **abnormal condition** that occur in a code sequence at ***run time.***
  - Exception is a RUN TIME ERROR
- A Java exception is an **object** that describes an exceptional (that is, error) condition that occurred in a piece of code.
- When an exceptional condition arises,
  - an object representing that exception is created and
  - *It is thrown in the method that caused the error.*
    - That method may choose to handle the exception itself, or pass it on.
    - The exception is then *caught and processed*

Exception Handling(contd.)

- Exceptions can be **generated by**
  - the Java run-time system, or
  - they can be manually generated by your code.
- Exceptions thrown by Java are related to
  - **Fundamental errors** that **violate the rules** of
    - the Java language or
    - the constraints of the Java execution environment.

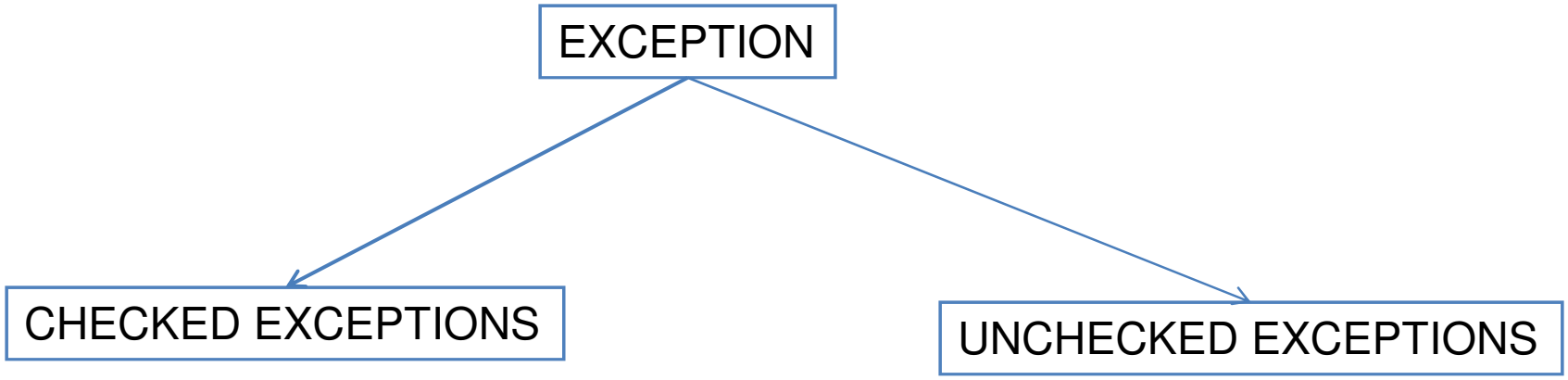
Exception Types

- All exception types are subclasses of the built-in class **Throwable.**
- **Throwable** has two subclasses that partition exceptions into two distinct branches.
  - ❑ One branch is headed by **Exception.**
    - This class is used for exceptional conditions that ***user programs should catch***. Subclass of this helps to create custom exception types.
    - **RuntimeException** is a subclass of **Exception.**



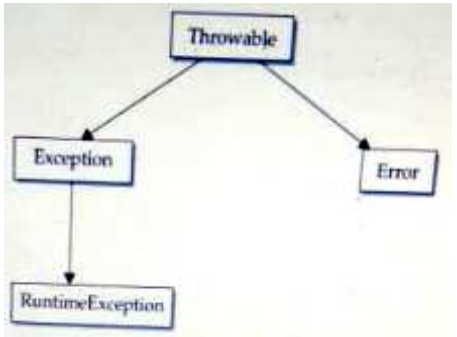
  - ❑ The other branch is headed by **Error**
    - This defines exceptions that are ***not expected to be caught*** under normal circumstances by our program.(***unchecked***)
    - Exceptions of type Error are used by the Java run-time system to indicate errors.

      E.g. Stack overflow,Out of Memory error


EXCEPTION
CHECKED EXCEPTIONS
UNCHECKED EXCEPTIONS

Unchecked exception

- Unchecked exception classes are defined inside **java.lang** package.
  - The **unchecked exceptions** are subclasses of the standard type **RuntimeException.**
  - In the Java language, these are called ***unchecked exceptions*** *because* ***the compiler does not check*** *to see whether there is a method that handles or* throws these exceptions.
  - If the program has unchecked exception then it will *compile without error* but exception occurs when program runs.



•E.g Exceptions under Error , ArrayIndexOutOfBoundException

Unchecked exception(contd.)

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Checked exception

- There are some exceptions that are defined by **java.lang** that **must be included in a method's throws list,** if a method generates such exceptions and that ***method does not handle it itself***. These are called **checked exceptions**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

- **IOException**
- **FileNotFoundException**
- **SQLException**

Checked exception(contd.)

- Checked exceptions are the exceptions (in java.lang) that are checked at compile time.
  - If some statement in a method **throws a checked exception**, then that method must
    - either handle the exception or
    - it must specify the exception using *throws* keyword.

Checked exceptions

- Checked at compile time.(COMPILE TIME EXCEPTIONS)
- Not sub class of RunTimeException
- The method must either handle the exception or it must specify the exception using *throws* keyword.
- Shows compile error if checked exception is not handled.
- E.g. *ClassNotFoundException, IOException*

Unchecked exceptions

- NOT checked at compile time.(RUN TIME EXCEPTINS)
- Sub class of RunTimeException
- It is NOT needed to handle or catch these exceptions
- DO NOT Show compile error if exception is not handled. But shows run-time error.
- Eg. *ArithmeticException, ArrayIndexOutOfBoundsException*

Exception handling fundamentals

Exception handling fundamentals(contd.)

- Program statements that we want to check for exceptions are written within a **try block.**
  - If an exception occurs within the try block, it is **thrown**.
  - The code inside **catch** can catch this exception and handle it in some manner.
- ***System-generated exceptions*** are automatically thrown by the Java run-time system.
- To **manually throw** an exception, use the keyword **throw.**
- Any exception that is thrown out of a **method** must be specified as such by a **throws** clause**.**
- Any code that absolutely must be executed after a **try block completes** is put in a **finally block.**

```
try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
// ...
finally
{
    // block of code to be executed after try block ends
}
```


Here, ExceptionType is the type of exception that has occurred.

Uncaught Exceptions

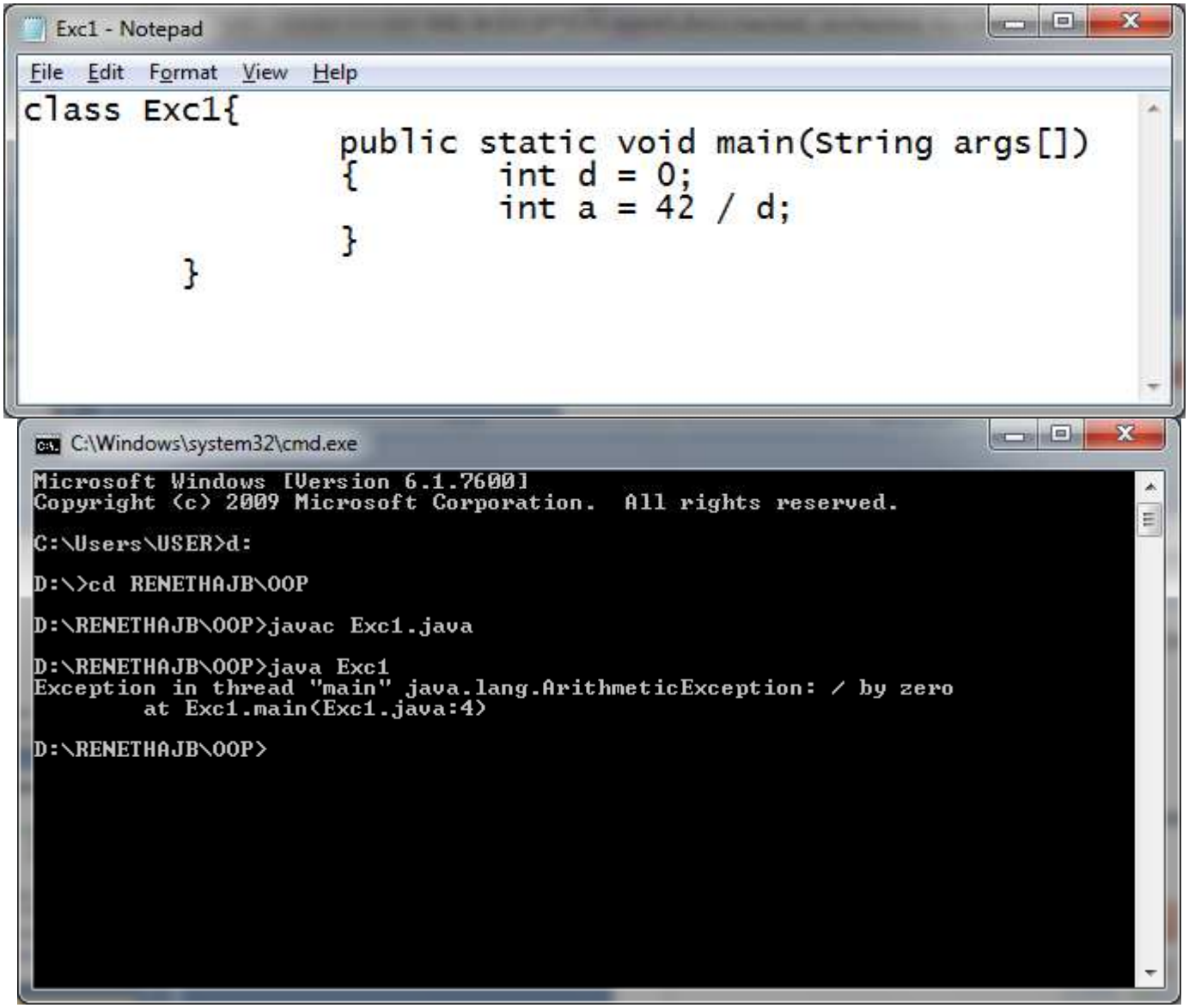
- Consider the program

```
Lineno.1        class Ex{
Lineno.2        public static void main(String args[])
Lineno.3                        {       int d = 0;
Lineno.4                                int a = 42 / d;
Lineno.5                        }
Lineno.6                }
```

- This small program causes a ***divide-by-zero error***((42/0)
- Java run time system constructs a new exception object and then *throws this exception.*
- *Th erogram stops by showing the following exception(run time erroe)*
- java.lang.ArithmeticException: / by zero at **Ex**.**main**(**Ex**.java:**4**)

- java.lang.**ArithmeticException**: / by zero at **Ex.main(Ex**.java:**4)**

- Here **Ex** is the class name , **main** is the method name,**;** **Ex.**java is the file name**;** and the exception is inline number **4.**
- These details are all included in the simple stack trace.
- The type of exception thrown is a subclass of Exception called **ArithmeticException** (describes what type of error happened.)


Exc1 - Notepad
File Edit Format View Help
class Exc1{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
}
}
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\USER>d:
D:\>cd RENETHAJB\OOP
D:\RENETHAJB\OOP>javac Exc1.java
D:\RENETHAJB\OOP>java Exc1
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Exc1.main(Exc1.java:4)
D:\RENETHAJB\OOP>

```
Lineno.1        class Exc1 {
Lineno.2        static void subroutine()
Lineno.3         { int d = 0;
Lineno.4          int a = 10 / d;
Lineno.5         }
Lineno.6        public static void main(String args[])
Lineno.7         { Exc1.subroutine();
Lineno.8         }
Lineno.9        }
```

- *java.lang.**ArithmeticException**: / by zero*

  *at Exc1.**subroutine**(Exc1.java:**4**)*

  *at Exc1.**main**(Exc1.java:**7**)*

try Block and *catch* *Clause*

- Benefits of exception handling
  - First, it allows us to **fix the error.**
  - Second, it **prevents** the program from **automatically terminating.**
- To **guard against** and **handle a run-time error**, simply enclose the code that we want to monitor inside a **try block.**
- Immediately ***after the try block***, there is a **catch clause** that **specifies the exception type** that we wish to catch . The catch block can process that exception..

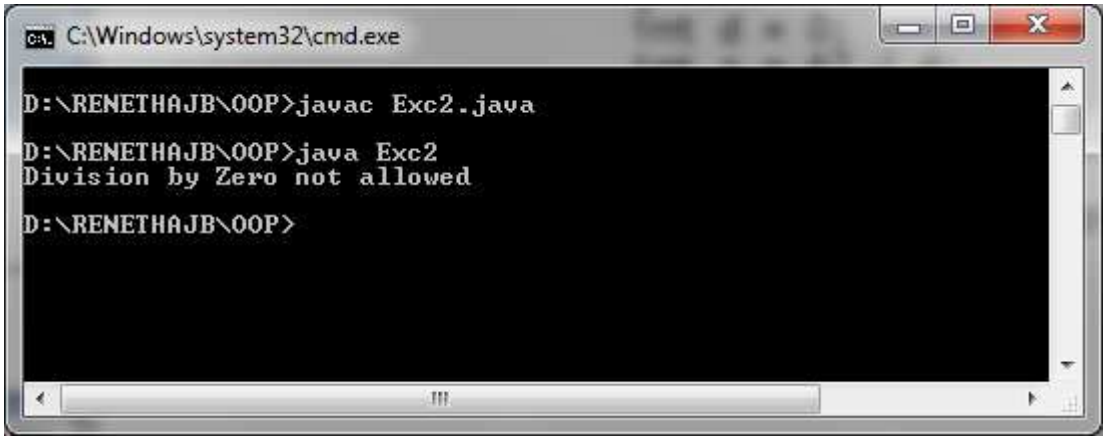
```
class Exc2{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("Division by Zero not allowed");
                }
        }
}
```

```
class Exc2{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("Division by Zero not allowed");
                }
        }
}
```

Exc2 - Notepad

File Edit Format View Help

```
class Exc2{
                public static void main(String args[])
                {       try
                        {
                        int d = 0;
                        int a = 42 / d;
                        }
                        catch(ArithmeticException ae)
                        {
                        System.out.println("Division by Zero not allowed");
                        }
                }
        }
```



try-catch block to handle division by zero exception


```
class Ex {
public static void main(String args[]) {
int d, a;
try {                                   // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
    }
    catch (ArithmeticException e)       // catch divide-by-zero error
    {
    System.out.println("Division by zero.");
    }
System.out.println("After catch statement.");
}
}
```

OUTPUT
Division by zero.
After catch statement.

Working of the program

- In this program the *System.out.println("This will not be printed.");* inside the try block is never executed because a = 42 / d;

- Once an exception is thrown, program control transfers out of the try block into the catch block.
  - i.e. catch is not "called" but controls goes out to catch when exception occurs, so execution never "returns" to the try block from a catch.
  - Thus, the line "This will not be printed." is not displayed.

try-catch (contd.)

- A try and its catch statement form a unit.
- The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
  - Each catch block can catch exceptions in statements inside immediately preceding try block.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- We cannot use try on a single statement

try-catch Example

```
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
      b = r.nextInt();
      c = r.nextInt();
      a = 12345 / (b/c);
    }
catch(ArithmeticException e)
{
      System.out.println("Division by zero.");
      a = 0;                    // set a to zero and continue
}
System.out.println("a: " + a);
} } }
```

Here **b** and **c** are random numbers .
If the value of b or c becomes zero then
a=12345./ (b/c) becomes
a=12345/0;
(Division by zero(ArithmeticException) will occur)
This statement is inside try block
So exception will be caught by catch and prints message ***Division by zero.***
and set the value of a to 0 and proceeds
NO RUNTIME ERROR!!

Displaying a Description of an Exception

- We can display this description in a **println( )** statement by simply passing the exception as an argument.

```
class Exc3{
        public static void main(String args[])
        {       try
                {
                int d = 0;
                int a = 42 / d;
                }
                catch(ArithmeticException ae)
                {
                System.out.println("The exception occurred is "+ae);
                }
        }
}
```

Exc3 - Notepad

File Edit Format View Help

```
class Exc3{
                public static void main(String args[])
                {       try
                        {
                        int d = 0;
                        int a = 42 / d;
                        }
                        catch(ArithmeticException ae)
                        {
                        System.out.println("The exception occurred is "+ae);
                        }
                }
        }
```

C:\Windows\system32\cmd.exe

```
D:\RENETHAJB\OOP>javac Exc3.java

D:\RENETHAJB\OOP>java Exc3
The exception occured is java.lang.ArithmeticException: / by zero

D:\RENETHAJB\OOP>
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 7)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **More features of Java :**
  - ☑ **Exception Handling:**
    - Multiple **catch** Clauses
    - Nested **try** Statements

Multiple catch Clauses

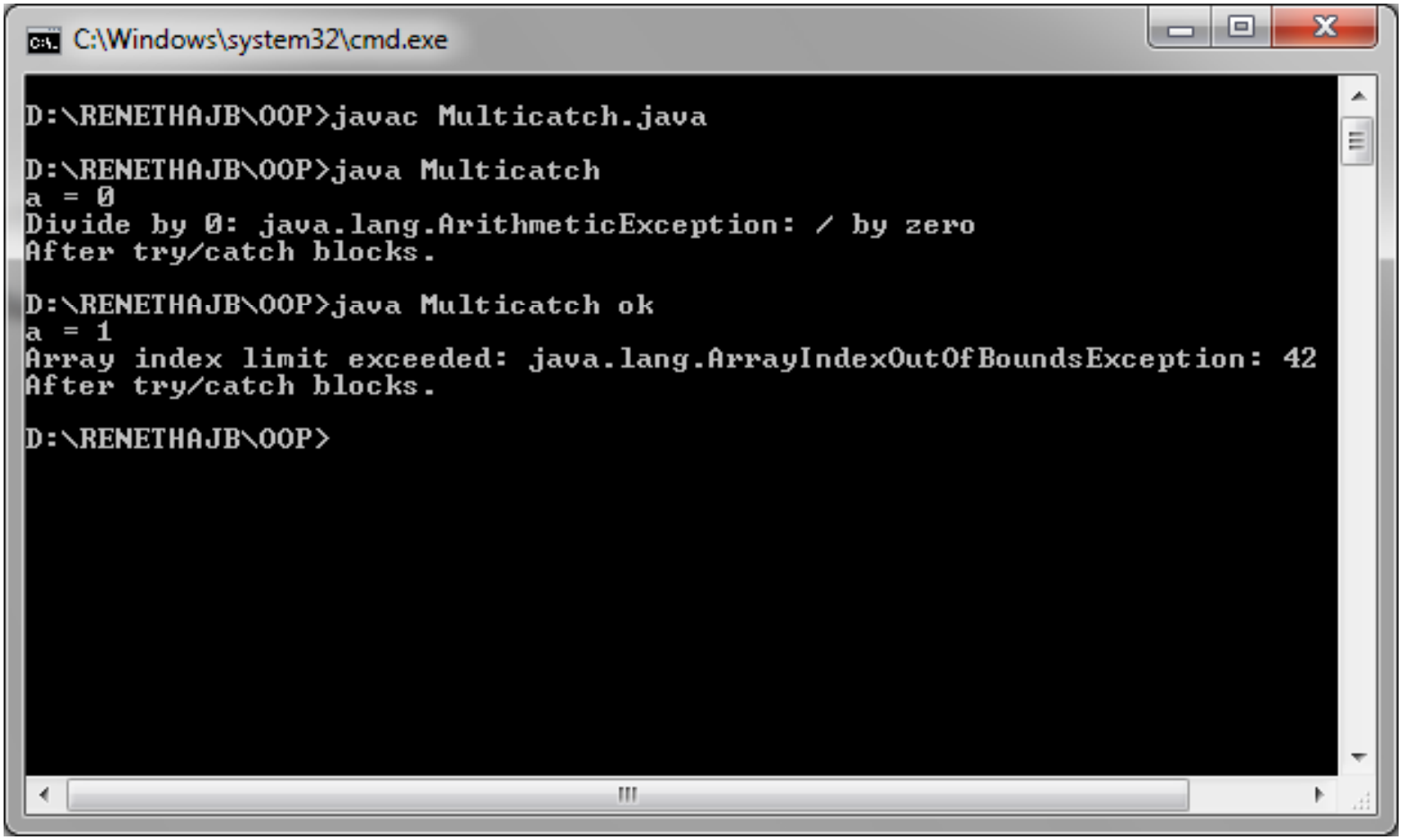
- There can be more than one exception in a single piece of code.
  - To handle this type of situation, we can specify two or more **catch** clauses, each catching a ***different type of exception***.
- When an exception is thrown,
  - each catch statement is inspected in order, and
  - the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the other catch statements are bypassed(ignored), and execution continues after the **try/catch** block.

Multi catch-Example

```
class Multicatch {
public static void main(String args[]) {
try {
      int a = args.length;                  //number of commandline arguments
      System.out.println("a = " + a);
      int b = 42 / a;              //when a is 0 this will raiseAthmeticException
      int c[] = { 1 };
      c[42] = 99;    //size of array is 1. So c[42] leds to ArrayIndexOutOfBoundsException
    }
     catch(ArithmeticException e)
      {
      System.out.println("Divide by 0: " + e);
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
      System.out.println("Array index oob: " + e);
      }
System.out.println("After try/catch blocks.");
}
}
```

Here the value of a is set as the number of command line arguments. If no command line arguments are there during execution
E.g. **java MultiCatch**
Here a is 0
So int b = 42 / a; will cause ArithmeticException. and is caught by **catch(ArithmeticException e)**.

If command line arguments are there ,then a is not zero. E.g. **java MultiCatch ok**
(Here a=1. So no exception occurs in int b = 42 / a **)**
Size of array c is1 (only one element).
So c[42] = 99; will cause ArrayIndexOutofBoundsException occurs(because position 42 is not there in this array)



- Output

C:\Windows\system32\cmd.exe
D:\RENETHAJB\OOP>javac Multicatch.java
D:\RENETHAJB\OOP>java Multicatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
D:\RENETHAJB\OOP>java Multicatch ok
a = 1
Array index limit exceeded: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.
D:\RENETHAJB\OOP>

Multi-catch (contd.)

- When we use multiple **catch statements,** it is important that exception subclasses must come before any of their superclasses.
- If we are using catch with superclass exception before the catch with subclass exception then catch with subclass exception will be ignored.
  - Such codes are unreachable. Unreachable code is an ERROR.

- E.g. Exception class is the superclass of all other exception classes like ArithmeticException, FileNotFoundException etc.

```
try
{
//statements
}
catch(Exception e)          //ALL EXCEPTIONS WIL BE CAUGHT HERE
{   //statements
}
catch(ArithmeticException ae)        //This catch is never used for catching
{//statements
}
```

Any exception that occurs in try block will be caught by the first suitable catch. Here all exceptions will match with Exception object. So even though ArithmeticExcepton occurs inside try block, it will be caught by catch(Exception e) block. So catch(ArithmeticException ae) will never catch it.

Multi catch(ERROR) // superclassexception should not be caught before catching subclass

```
class SuperSubCatch {
public static void main(String args[])
    {
     try {
     int a = 0;
     int b = 42 / a;
     }
     catch(Exception e)        //All exceptions are caught here
     {System.out.println("Generic Exception catch.");
     }
     /* The next catch is never reached because
     ArithmeticException is a subclass of Exception. */
     catch(ArithmeticException e)
     {      // ERROR - unreachable
     System.out.println(" Arithmetic Exception occurred ");
     }
    }
}
```

COMPILE ERROR- the **second catch statement** is **unreachable** because the exception has already been caught by Exception

A subclass must come before its superclass in a series of catch statements.


```
class SuperSubCatch {
public static void main(String args[])
{
    try {
    int a = 0;
    int b = 42 / a;
    }
    catch(ArithmeticException e)
    {
    System.out.println(" Arithmetic Exception occurred ");
    }
    catch(Exception e)
    {System.out.println("Generic Exception catch.");
    }
}
}
```

This is the correct usage of catch. The catch with subclass exception(AritnmeticException) should appear before catch with super class exception(Exception)

Nested *try* Statements

- The **try** statement can be nested.
  - A **try** statement can be inside the block of another **try.**
- Each time a try statement is entered, the context of that exception is pushed on the stack.
  - If an **inner try** statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
  - This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
  - If **no catch statement matches**, then the Java run-time system will handle the exception.

```
class NestTry {
public static void main(String args[]) {
try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        try {
            if(a==1) a = a/(a-a);          // division by zero
            if(a==2)
             { int c[] = { 1 };
               c[42] = 99;                 // generate an out-of-bounds exception
             }
        } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
        }
}
catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block**.**

Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block.

Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled.

If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

Nested try(contd.)

- We can enclose a call to a method within a **try** block.
  - Inside that method we can have another try statement.
- In this case, the try within the method is still nested inside the outer try block, which calls that method.

```
class MethNestTry {
static void show(int a) {
try {          // nested try block

if(a==1) a = a/(a-a);     // division by zero
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[]) {
try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        show(a);// show contains a try – catch . So nested try.
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

Here try in main function act as outer try block. Inside that try show() function is called . So try catch inside show() function is **inner** to the try in main function.

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block and is caught by outer catch clause(matching is there)**.**
Execution of the program with one command-line argument generates a divide-by-zero exception from within the try block in show().
Since the inner catch(no matching) block does not catch this exception, it is passed on to the outer try block in main function(matching is there) , and it is handled.

If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block and is caught by innercatch inside show

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 3 - More features of Java (Part 8)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **More features of Java :**
  - ☑ **Exception Handling:**
    - ***throw***
    - ***throws***
    - ***finally***

throw statement

- Our program can throw an exception explicitly, using the **throw** statement.
- The general form of **throw is shown here:**

```
throw ThrowableInstance;
```

  - *ThrowableInstance* must be an **object** of type ***Throwable*** or a ***subclass of Throwable.***
  - Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

throw(contd.)

- Two ways to obtain a **Throwable object:**

1. using a parameter in a **catch** clause, or
2. creating one with the **new** operator

1) Using a parameter in a catch clause

```
catch(ArrayIndexOutOfBoundsException ar)
{
    throw ar;
}
```

2) Creating one with the new operator

```
throw new ArrayIndexOutOfBoundsException();
```

throw statement(contd..)

- The *flow of execution* **stops** immediately after the **throw** statement.
  - Any statements after throw statement will not be executed.
- When exception is thrown using **throw** statement :-
  - the nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception thrown.
    - If that catch statement has a **matching exception type as the thrown exception**, control is transferred to that statement.
    - If **not matching**, then the *next enclosing try statement is inspected, and so on.*
    - If **no matching catch is found**, then the *default exception handler halts the program and prints the stack trace.*

throw Example 1

```
class ThrowDemo
{
static void show()
{
     try
     {throw new NullPointerException("demoexception");
     }
     catch(NullPointerException e)
     {
      System.out.println("Caught inside show");
      throw e;        // rethrow the exception
      }
}
public static void main(String args[])
{
      try {
      show();
           }
      catch(NullPointerException e)
      { System.out.println("Recaught in main: " + e);
      }
} }
```

Here first **throw** in show is caught by matching catch is in show function.

Next throw has no immediate catch So since the exception matches with catch in the **main** function(that calls show), the exception is caught by that matching catch in main.

java ThrowDemo
Caught inside show
Recaught in main: java.lang.NullPointerException: **demoexception**

throw with matching catch in calling function


```
class ThrowDemo2
{
   static void show()
   {
         throw new NullPointerException("demoxception");
   }
    public static void main(String args[])
    {
         try
         {
         show();
         } catch(NullPointerException e)
         { System.out.println("Caught in main: " + e);
         }
     }
}
```

Here no matching catch for throw is in show functior
So since the exception matches with **catch** in the main function(that calls show), the exception is caught by that matching catch

OUTPUT

Caught in main: java.lang.NullPointerException: demoexception

throw with NO matching catch


```
class ThrowDemo2
{
   static void show()
   {
         throw new NullPointerException("demoxception");
   }
    public static void main(String args[])
    {
         try
         {
         show();
         }
          catch(ArithmeticException e)
         { System.out.println("Caught in main: " + e);
         }
     }
}
```

Here no matching catch is in show function.
So since the exception does not matches
with catch in the main function(that calls show)
also, the exception is not caught in the program
the ***default exception handler halts the program and prints the stack trace***

OUTPUT

```
Exception in thread "main" java.lang.ArithmeticException: demoxception
        at ThrowDemo2.show(ThrowDemo2.java:3)
        at ThrowDemo2.main(ThrowDemo2.java:9)
```

throw(contd.)

- Many of Java's built-in run-time exceptions have at least two constructors:
  - one with no parameter and
  - one that takes a string parameter.
- When constructor with string parameter is used, the argument specifies a **string <u>that describes the exception</u>**.
  - This string is displayed when the object is printed using **print( ) or println( ).**
  - It can also be obtained by a call to getMessage( ), which is defined by Throwable.

  throw new ***NullPointerException***("**demoxception**");
- Here the string **demoxception** inside the constructor of ***NullPointerException*** is the name of the exception.

throws

- A **throws** clause lists the types of exceptions that a **method(function) might throw**.
- **throws** keyword is used with the method signature(header)
- If a method has an exception and it does not handle that exception, it must specify this using **throws** , so that callers of the method can guard themselves against that exception.
- **throws** is necessary for all exceptions, except those of type **Error** or **RuntimeException** or any of their subclasses

throws (contd.)

- All other **exceptions** that a **method can throw** must be declared in the **throws** clause.
  - If they are not, a compile-time error will result.
- General form of a method declaration that includes a **throws clause:**

type method-name(parameter-list) ***throws*** *exception-list*

```
{
   // body of method
}
```

throw statement but no throws in method-ERROR

```
public class ThrowsEg {
  static void vote(int age) {
    if (age < 18) {
      throw new IllegalAccessException("You must be at least 18 years old.");
                    } else {
                      System.out.println(" You can vote!");
                            }
  }
   public static void main(String[] args)
  {
            vote(15);
  }
}
```

```
D:\RENETHAJB\OOP>javac ThrowsEg.java
ThrowsEg.java:4: unreported exception java.lang.IllegalAccessException; must be
caught or declared to be thrown
      throw new IllegalAccessException("You must be at least 18 years old.");
      ^
1 error
```

COMPILE ERROR

This program tries to throw an exception that it does not catch.

Because the program does not specify a **throws clause to declare this exception to be thrown,** the program will not compile.

Include **throws** in method and **try catch** in calling function.

Using throws


```
public class ThrowsEg {
 static void vote(int age) throws IllegalAccessException{
  if (age < 18) {
    throw new IllegalAccessException("You must be at least 18 years old.");
                } else {
                        System.out.println(" You can vote!");
                        }
 }
 public static void main(String[] args)
 {
  try{
            vote(15);
     }
     catch(Exception e)
     {
    System.out.println("Exception: "+e);
     }
 }
}
```

OUTPUT

Exception: java.lang.ArithmeticException: You must be at least 18 years.

```
import java.io.*;
class Sample{
 void show() throws IOException{
  throw new IOException("Thrown IO error");
 }
}
public class Testthrows{
   public static void main(String args[]){
    try{
     Sample s=new Sample();
     s.show();
    }
catch(Exception e){System.out.println("Exception handled. "+e);}

    System.out.println("Normal program flow");
  }
}
```

Output

```
Exception handledjava.io.IOException: Thrown IO error
Normal program flow
```

finally

- **finally** creates a block of code that will be executed **after** a try/catch block has completed and **before** the control goes out from the try/catch block.

```
try {
// block of code to monitor for errors
      }
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{   // exception handler for ExceptionType2
}
// ...
finally
{   // block of code to be executed after try block ends
}
```

Why finally is needed?

- When exceptions are thrown, execution in a method takes a nonlinear path and changes the normal flow through the method.
  - Sometime exception causes the method to return prematurely.
  - This may cause problems in some cases.
  - E.g a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism.
    - In such situations the code for closing that file and other codes that should not be bypassed should be written inside **finally** block
    - This will ensure that necessary codes are not skipped because of exception handling.

finally(contd.)

- The **finally** block **will execute** whether or not an exception is thrown.
  - If an **exception is thrown**, the **finally** block will execute *even if no catch statement matches the exception*.
  - Any time a method is about to return to the caller from inside a **try/catch block**, (via an uncaught exception or an explicit return statement). the **finally clause is also executed just before the method returns.**
- *If* a **finally** block is associated with a **try**, the finally block will be executed upon conclusion of the try.

finally(contd.)

- The finally clause is optional. However, each try statement requires **at least one catch or a finally clause**

```
try
{
//monitor exception
}
finally
{
}
```

```
try
{
//monitor exception
}
catch(ExceptionType1 ob)
{
}
```

```
try
{
//monitor exception
}
catch(ExceptionType1 ob)
{
}
catch(ExceptionType2 ob)
{
}
//
finally
{
}
```

finally example

```
class FinallyTry
{
public static void main(String[] args)
    {
        try
        {
        int a=5/0;
        }
        catch(ArithmeticException ae)
        {
        System.out.println("Exception is "+ae);
        }
        finally
        {
        System.out.println("Inside finally");
        }
    System.out.println("After try - catch -finally");
    }
}
```

OUTPUT

Exception is java.lang.ArithmeticException: / by zero
Inside finally
After try - catch -finally

Here int a=5/0; inside try causes ArithmeticException
And it is caught by **catch(ArithmeticException ae)**
And prints the message
Exception is ***details about exception***
Then it enters finally block and prints ***Inside finally***
Then it comes out from try catch finally block
and prints the message
After try - catch -finally

finally example

```
class FinallyTry
{
public static void main(String[] args)
    {
        try
        {
        int a=5/2;
        }
        catch(ArithmeticException ae)
        {
        System.out.println("Exception is "+ae);
        }
        finally
        {
        System.out.println("Inside finally");
        }
    System.out.println("After try - catch -finally");
    }
}
```

OUTPUT

Inside finally

After try - catch -finally

Here int a=5/2; inside try does not cause exception (So it is not caught by catch(ArithmeticException ae))

Then it enters finally block and prints ***Inside finally***

Then it comes out from try catch finally block and prints the message

After try - catch -finally

finally Example

```
class Sample{
  void show(int n)
  {  int c=10;
     try
     {
     System.out.println("inside try");
     c=10/n;
     }
     catch(Exception e)
     {
     System.out.println("Exception  caught"+e);
     }

     finally
     {
     System.out.println("Finally done");
     }
  }


}
```

```
class Finally1
{
public static void main(String[] args)
     {
          Sample ob=new Sample();
          ob.show(1);
          ob.show(0);

          System.out.println("Finished");
     }
 }
```

```
inside try
Finally done
inside try
Exception caughtjava.lang.ArithmeticException: / by zero
Finally done
Finished
```

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
        }
}
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
        }
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {

System.out.println("procC's finally");
        }
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```


Java

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.