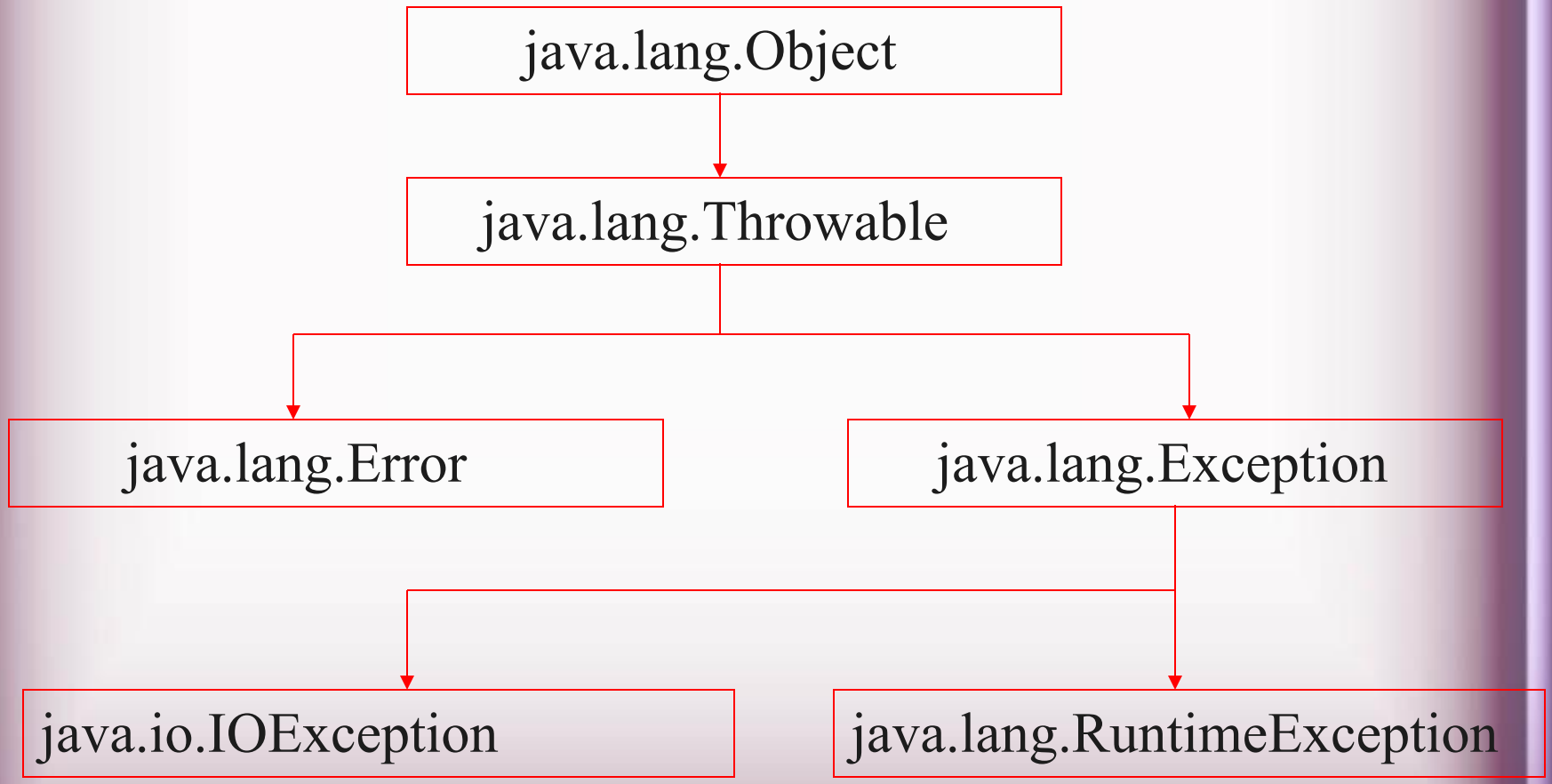
Exception handling

Exception Handling

- An exception is a problem that arises during the execution of a program.

- When an Exception occurs the normal flow of the program is disrupted and the program terminates abnormally, which is not recommended
  - Exceptions are to be handled.

Exception Handling



Exceptions

- ***checked exceptions***
  - Conditions that can readily occur in a correct program
  - **Includes** all subtypes of **Exception**
  - ***Excluding*** classes that extend **Runtime Exception**.
  - Subject to the *handle or declare* rule

- ***unchecked exceptions***
  - Problems that reflect program bugs
  - Probable bugs are represented by the RuntimeException class.
  - Compiler **doesn't enforce the handle or declare rule**.

- **Error**
  - Fatal situations are represented by the Error class.

Handle or Declare Rule

- Handle the exception by using the try-catch-finally block.

```
try{
    // some code
    } catch(Exception e){
}
```

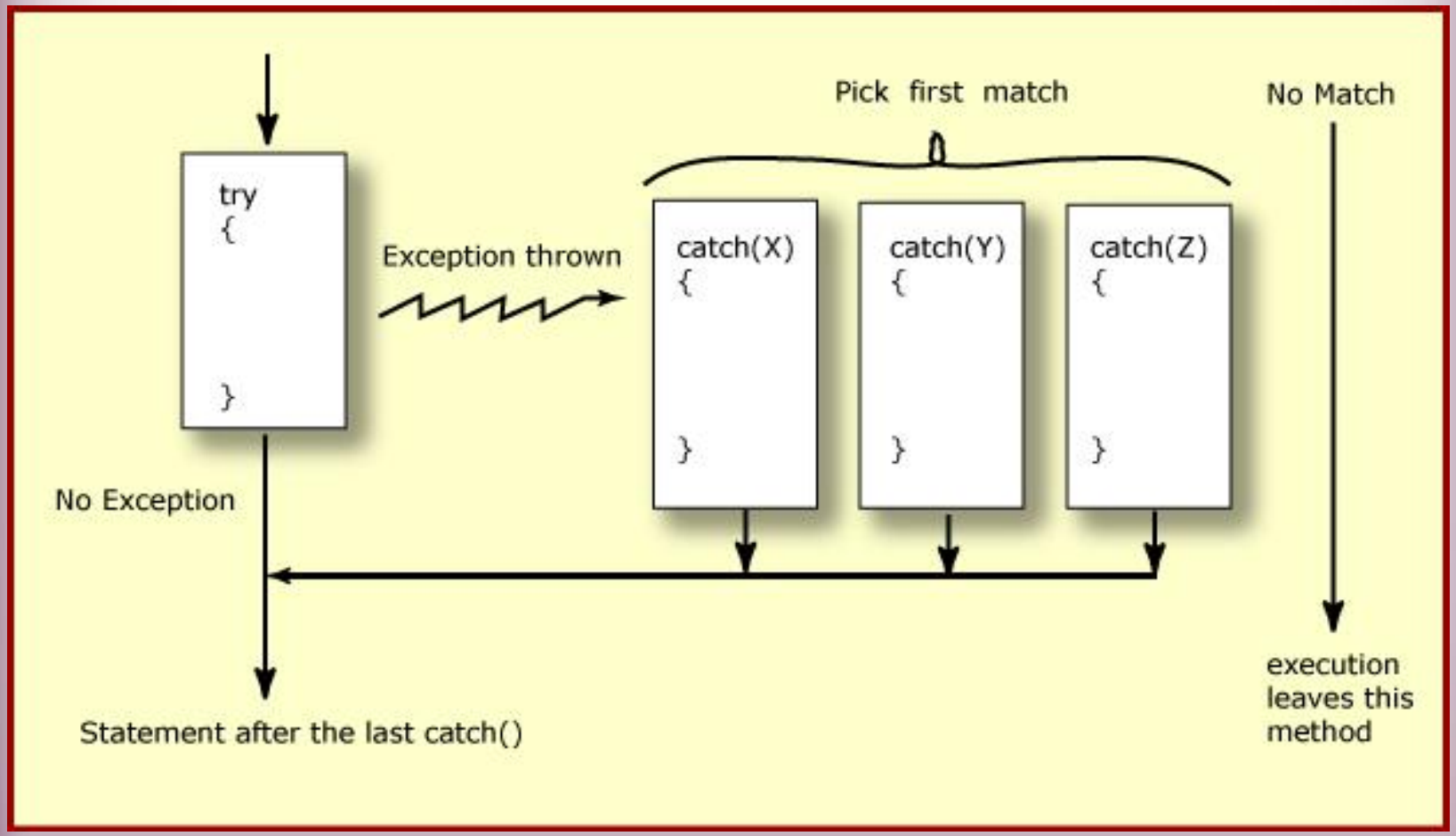
- Declare that the code causes an exception by using the throws clause.

  void trouble() **throws IOException { ... }**

  void trouble() **throws IOException, MyException { ... }**

- Other Principles
  - **You do not need to declare runtime exceptions or errors.**
  - **You can choose to handle runtime exceptions**

How try catch Work



Exception Handling

```
try {
   Object value = "twenty";
   Integer number = (Integer) value;
}
catch ( ClassCastException ex ) {
            ex.printStackTrace();
}
System.out.println("Good-bye" );

}
```

Nested Try Catch

```
try {
Object value = "twenty";
Integer number = (Integer) value;

    try {
    String mark ="50";
     Integer.parseInt(mark);
    } catch (NumberFormatException e) {
       e.printStackTrace();
    }
}
catch ( ClassCastException ex ) {
            ex.printStackTrace();
}
```

Catching Exception –Multiple Catch

```
try {
Object value = "twenty";
Integer number = (Integer) value;

String mark ="50";

 Integer.parseInt(mark);

}
catch ( ClassCastException ex ) {
      ex.printStackTrace();
}
catch (NumberFormatException e) {
    e.printStackTrace();
    }
```

Effective Exception Hierarchy

- The exception classes are arranged in a hierarchy.
- Catches for specific exceptions should always be written prior to the generic handler.
- A generic handler for Exception would cover any missing situation.

```
try
{
}
catch
(NumberFormatException e) {
}
catch (Exception ex2) {

 }
```

```
try
{

}
catch (Exception e1)    {
}
catch(NumberFormatException
e2) {

 }
```

Multiple Catch

```
try {
Object value = "twenty";
Integer number = (Integer) value;

String mark ="50";

 Integer.parseInt(mark);

}
catch ( ClassCastException| NumberFormatException ex )
{
            ex.printStackTrace();
}
```

Multiple Catch

```
try {

} catch (RangeCheckException | RuntimeException e) {
   }


try {

 } catch (RangeCheckException | Exception e) {

}
```

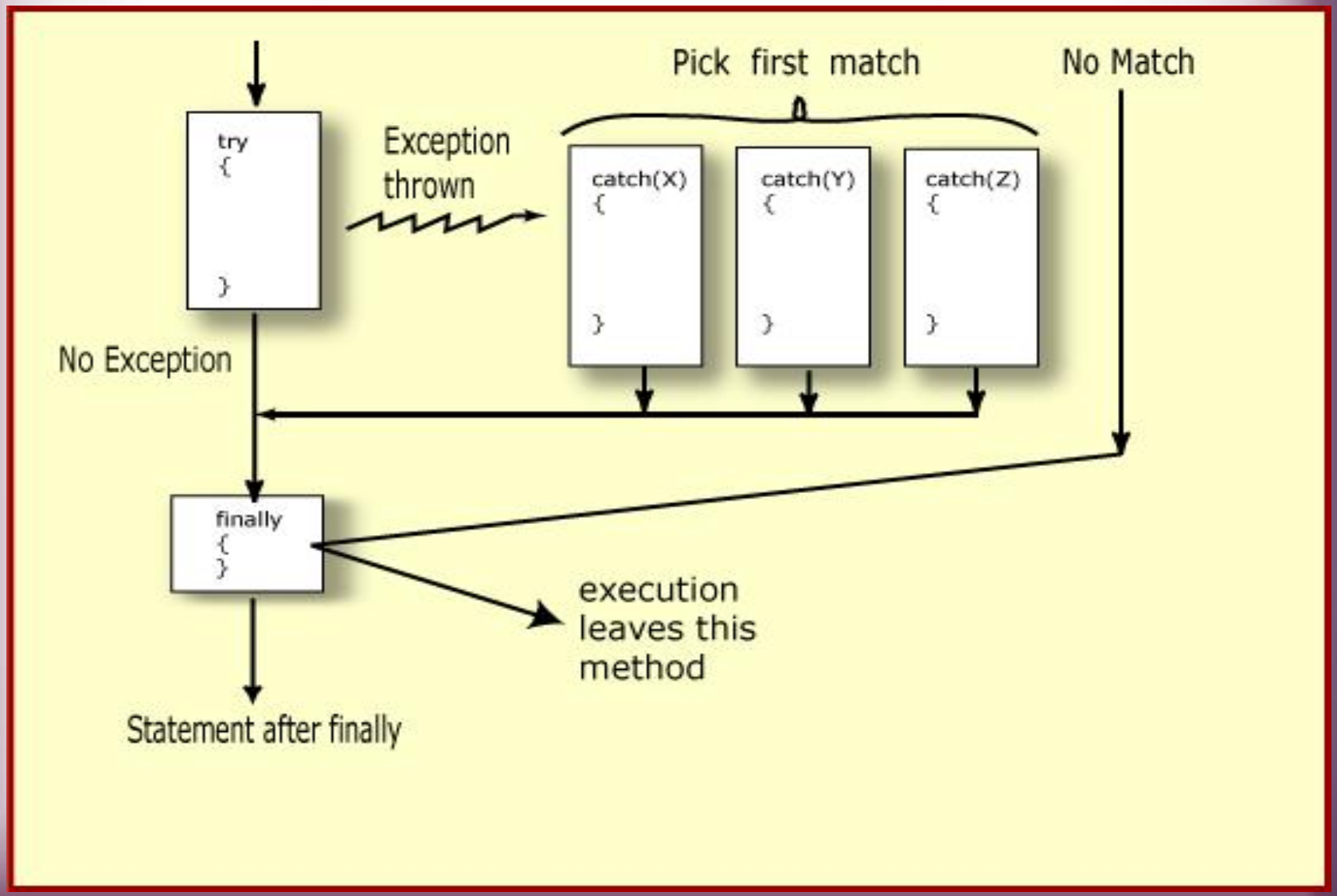
Finally block

- Its optional and will always be invoked,
  - whether an exception in the corresponding *try* is thrown or not,
  - whether a thrown exception is caught or not.

- *finally*-will-always-be-called except if the JVM shuts down.
  - When the try or catch blocks may call System.exit();

- Finally Block is always executed despite of different scenarios:
  - Forced exit occurs using a *return*, a *continue* or a *break* statement
  - Normal completion
  - Caught exception thrown
  - Exception was thrown and caught in the method
  - Uncaught exception thrown
  - Exception thrown was not specified in any catch block in the method

Finally Block



Finally Block

```
public static String tryblkWithReturn(String data ) {
try   {
int no = Integer.parseInt(data);
}
catch (NumberFormatException e)  {
    System.out.println("Entering Catch Block");
return "Hello";
}
finally {
System.out.println("Inside finally");
}
System.out.println("I am back");
return "Hi";
}
```

Finally Block

```
public static String tryblkWithNewException(String data )
    throws Exception  {
try   {
int no = Integer.parseInt(data);
}
catch (NumberFormatException e) {
System.out.println("Entering Catch Block");
throw new Exception();
}
finally {
System.out.println("Inside finally");
}
   System.out.println("I am back");
 return "Hi“;
}
```

Throws Clause

- A method is required to either catch or list all exceptions it might throw
  - Except for *Error* or *RuntimeException*, or their subclasses

- If a method may cause an exception to occur but does not catch it, then it must say so using the *throws* keyword

- Applies to checked exceptions only

```
<type> <methodName> (<parameterList>)  throws <exceptionList>
{
    <methodBody>
}
```

The try-with-resources Statement

- A try statement that declares one or more resources.
- A *resource* is an object that must be closed after the program is finished with it.
- Ensures the resource is closed at the end of the statement.
- Any object that implements java.lang.AutoCloseable, and java.io.Closeable, can be used as a resource.

Exception Handling- Closing the Resource

```
Scanner scan = new Scanner( System.in );
 int num ;
 System.out.print("Enter an integer: ");
try {
      num = scan.nextInt();
    System.out.println("The square of " + num + " is " +
num*num );
}
catch ( InputMismatchException ex ) {
   System.err.println("You entered bad data." );
   System.err.println("Run the program again." );
}
System.err.println("Good-bye" );

 scan.close();
}
```

The try-with-resources Statement

```
System.out.print("Enter an integer: ");
try(Scanner scan = new Scanner( System.in )) {
      int num = scan.nextInt();

System.out.println("square" + num + "is" + num*num );

}
catch ( InputMismatchException ex ) {
   System.err.println("You entered bad data." );
   System.err.println("Run the program again." );
}
    System.out.println("Good-bye" );
}
```

Throwing Exceptions

- **The *throw* Keyword**
- Java allows you to throw exceptions (generate exceptions)
  - **throw <exception object>**
- An exception you throw is an object
  - You have to create an exception object in the same way you create any other object
- Example:
  - **throw new ArithmeticException(“testing...”);**

Re-throw Exception

- Exception of the Catch Parameter type is the one that is rethrown
- To intercept all exceptions and rethrow them
  - Can catch with super class Exception
  - Declare method as throwing an Exception.
- "imprecise rethrow“
  - Throwing a general Exception type (instead of specific ones)

Re-Throw Exception

- Java 7, can be more precise about the exception types being rethrown from a method.
- This leads to improved checking for rethrown exceptions.
- Can handle them a lot better at the calling side

Re-throw Exceptions

```
public static void myExample() throws
    ParseException,IOException {

try {

new SimpleDateFormat("yyyyMMdd").parse("date");
new FileReader("Abc.txt").read();

} catch (Exception e) {

    throw e;
  }
}
```

Method Overriding and Exceptions

- **The overriding method can throw:**
  - No exceptions
  - One or more of the exceptions thrown by the overridden method
  - One or more subclasses of the exceptions thrown by the overridden method
- **The overriding method cannot throw:**
  - Additional exceptions not thrown by the overridden method
  - Superclasses of the exceptions thrown by the overridden method

Overriding –When Method Has Exceptions

- Any exceptions declared in overriding method must be of the same type as those thrown by the super class, or a subclass of that type.

```
class MyBase {

  public void method1() throws Exception
    {  }
  public void method2() throws Runtim
    {  }
}
class Sub extends MyBase
{
  public void method1() throws Throwable {
    }
  public void method2() throws
                        ArithmeticExc
  }
}
```

This is compile Time Exception

This is Allowed

User Defined Exception

- User Defined Exception must be a child of Throwable.
- To write a checked exception that is automatically enforced by the Handle or Declare Rule, need to extend the Exception class.
- To write a runtime exception, need to extend the RuntimeException class.

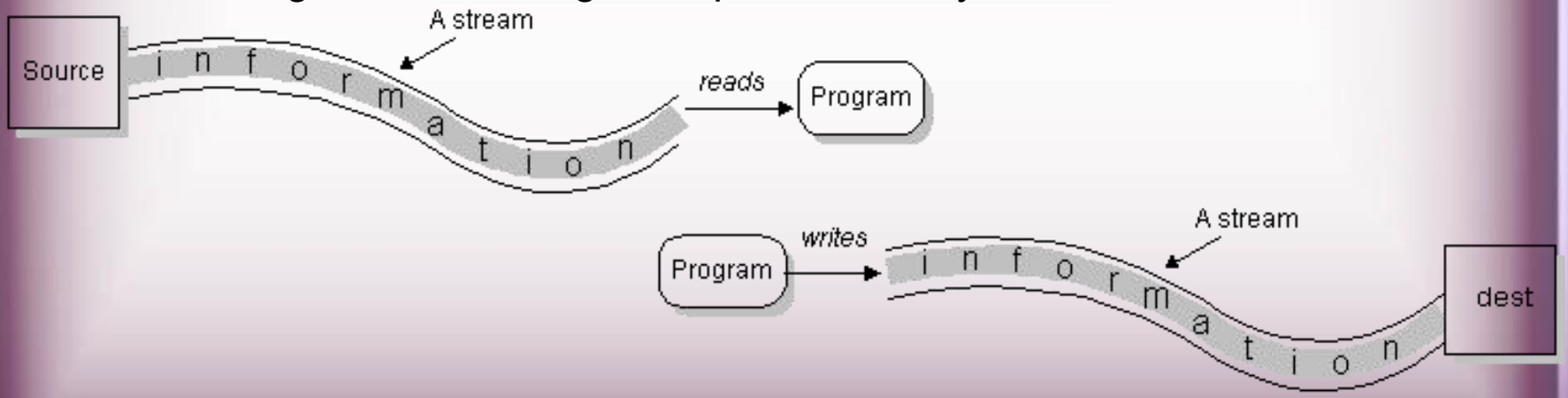
Files & IO

Overview

- At its lowest level, all Java I/O involves a stream of bytes either entering or leaving memory
- Packaged classes exist to make it easy for a program to read and write larger units of data.
- Low-level stream class object are used to handle byte I/O
- High-level stream class object will allow the program to read and write primitive data values and objects

File Classes

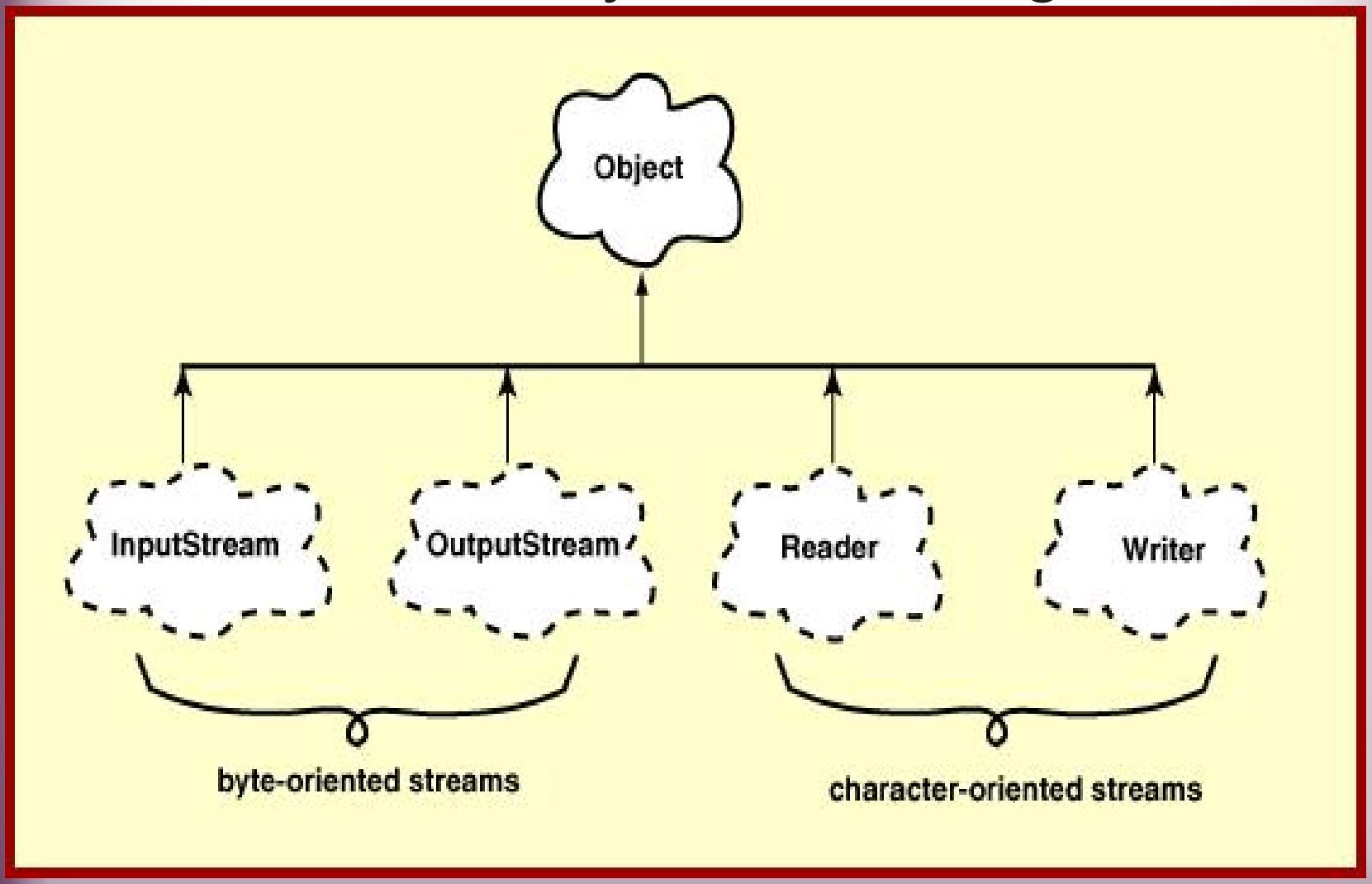
- Java views the data in files as a stream of bytes.
- A stream of bytes from which data are read is called an ***input stream.***
- A stream of bytes to which data are written is called an ***output stream***.
- Java provides classes for connecting to and manipulating data in a stream.
- The classes are defined in the package java.io and are organized in a large complex hierarchy.



File Class

- This is an abstract representation file and directory pathnames
  - Not used to read and write data
  - Used for searching and deleting of files, creating directories and working with path and making directories
  - has methods for getting file/directory info.
  - cannot be used to read or write to a file.
- To make a **File** object, there are three commonly used constructors `File`
  - `File file1 = new File("C:\\Data\\myFile.dat");`
  - public File(String directory, String filename)

Hirearchy of IO Package



FileWriter

- Used to write character to files
- The Write method() of this classes is used to write characters
- Wrapped by BufferedWriter to improve the performance

FileWriter(File file)

FileWriter(File file, boolean append)

All constructors and methods throw **IOException**

Print Writer

- PrintWriter wraps the FileWriter
- Has flexible and powerful methods to write data to files
- format(),printf() and append() are used to work with data
- print() or println() take a single parameter of any data type

FileReader

- Used to read characters from files
- The read() method of this class is used to read characters
- These are wrapped by BufferedReader to improve the performance

Constructors:

FileReader(File file) throws FileNotFoundException

FileReader(String fileName) throws FileNotFoundException

Buffering

- **BufferedReader**
  - Buffers give a temporary holding place to group until it reaches final place.
  - It helps in efficent reading of files
  - Reads a large amount of data from a file at once and keep that data in buffer
  - Minimizes the number of times of file access.
  - Provides a readLine() method , which reads a line of characters from the file

  - ```
    BufferedReader in =
            new BufferedReader(new FileReader("file.txt"));
    ```

- **BufferedWriter**
  - Provides a newLine() method , which that creates line separators automatically
    ```

Write To File

```
 public boolean writeToTextFile(Professor prof,File file) {

 boolean result = false;

 try (PrintWriter writer = new PrintWriter(new
FileWriter(file,true))){

 writer.println(prof);

 result =true;

} catch (IOException e) {
e.printStackTrace();
}
      return result;
       }
```

Read From File

```
public Professor[] readFromTextFile(File file) {

Professor[] list = new Professor[4];

String line =null;

try(BufferedReader reader = new BufferedReader(new
FileReader(file))){

int i = 0;
```

Read From File

```
while( (line = reader.readLine())!=null) {

String[] values = line.split(",");
Professor prof = new
Professor(Integer.parseInt(values[0]),
  values[1],values[2], values[3]);

list[i]= prof;
i++;
}
}catch(IOException e) {
 e.printStackTrace();
  }
return list;
}
```

Serializing

- Creating the sequence of bytes from an object and Recreating the object from the above generated bytes
- Ability to read or write an object to a stream
  - Process of "flattening" an object
- Used to save object to some permanent storage
  - Its state should be written in a serialized form to a file such that the object can be reconstructed at a later time from that file
- Used to pass on to another object via the *OutputStream* class
  - Can be sent over the network

To use serialization

- Most Java classes are serializable
- Classes need to implement the serializable interface.
  - *Serializable* interface is marker interface
  - Class should also provide a default constructor with no args
- Objects of some system-level classes are not serializable
  - Because the data they represent constantly changes
  - Reconstructed object will contain different value anyway
- A *NotSerializableException* is thrown if you try to serialize non-serializable objects

To use serialization

- Only the object's data are preserved, Methods and constructors are not part of the serialized stream ,the class information is included
- Marking a field with the *transient* keyword
  - The *transient* keyword prevents the data from being serialized
  - All non-transient fields are considered part of an object
- Have access to the no-argument (or default) constructor of its first nonserializable superclass (or supersuperclass, supersupersuper class)
- Serializability is inherited

Serialization

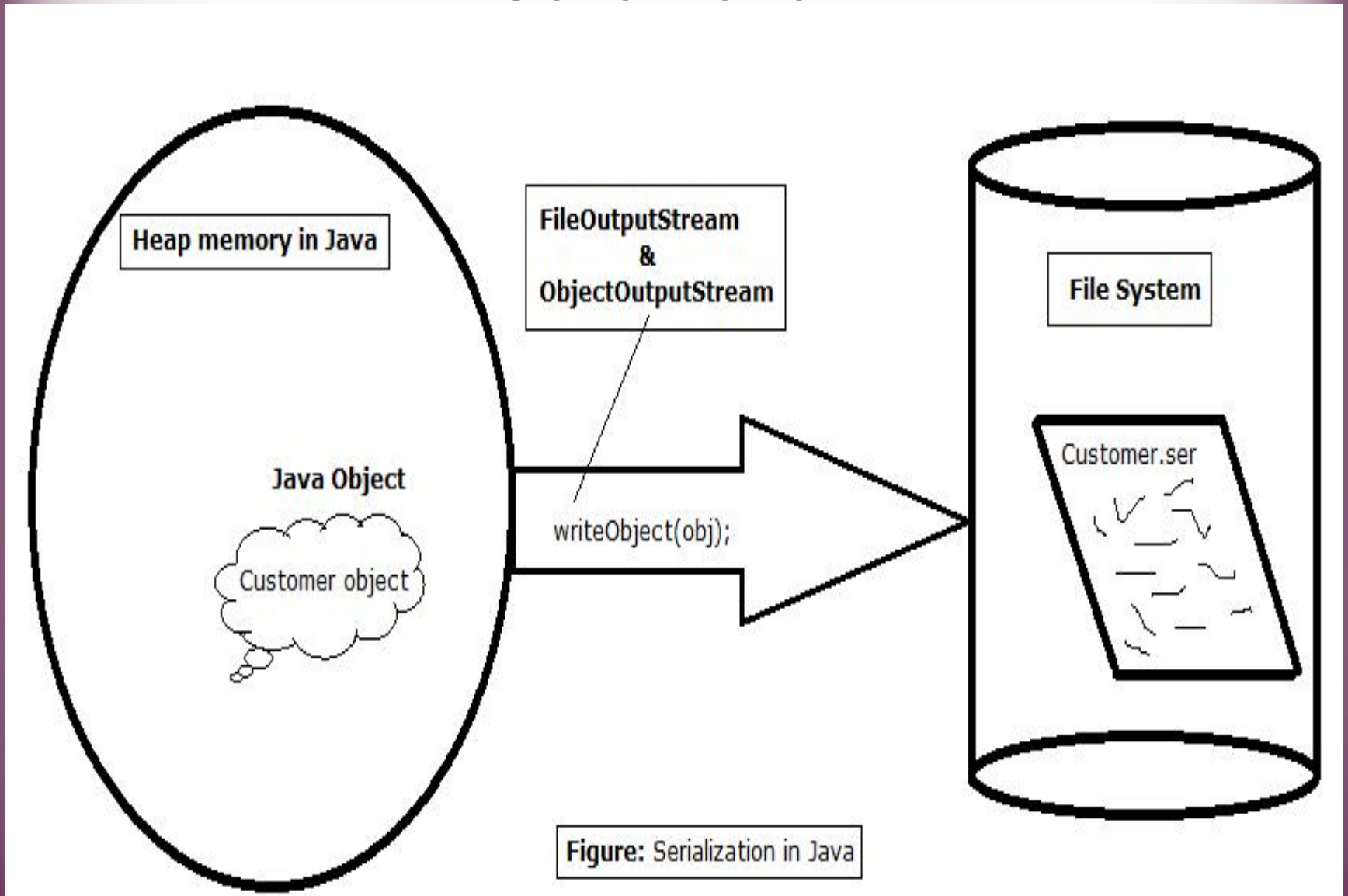


Figure: Serialization in Java

Serialization

```
 public boolean writeObjectToFile(Professor prof,File
file) {
   boolean result = false;

try(ObjectOutputStream outStream = new
ObjectOutputStream(new FileOutputStream(file))){

 outStream.writeObject(prof);
 result =true;
  }catch(IOException e) {
  e.printStackTrace();
 }
    return result;
  }
```

DeSerialization

```
 public Object readObjectFromFile(File file) {

 Object obj=null;

 try(ObjectInputStream inStream = new ObjectInputStream(new
FileInputStream(file))){

 obj = inStream.readObject();

}catch(ClassNotFoundException | IOException e) {

 e.printStackTrace();
 }

 return obj;
  }
```

De Serialization

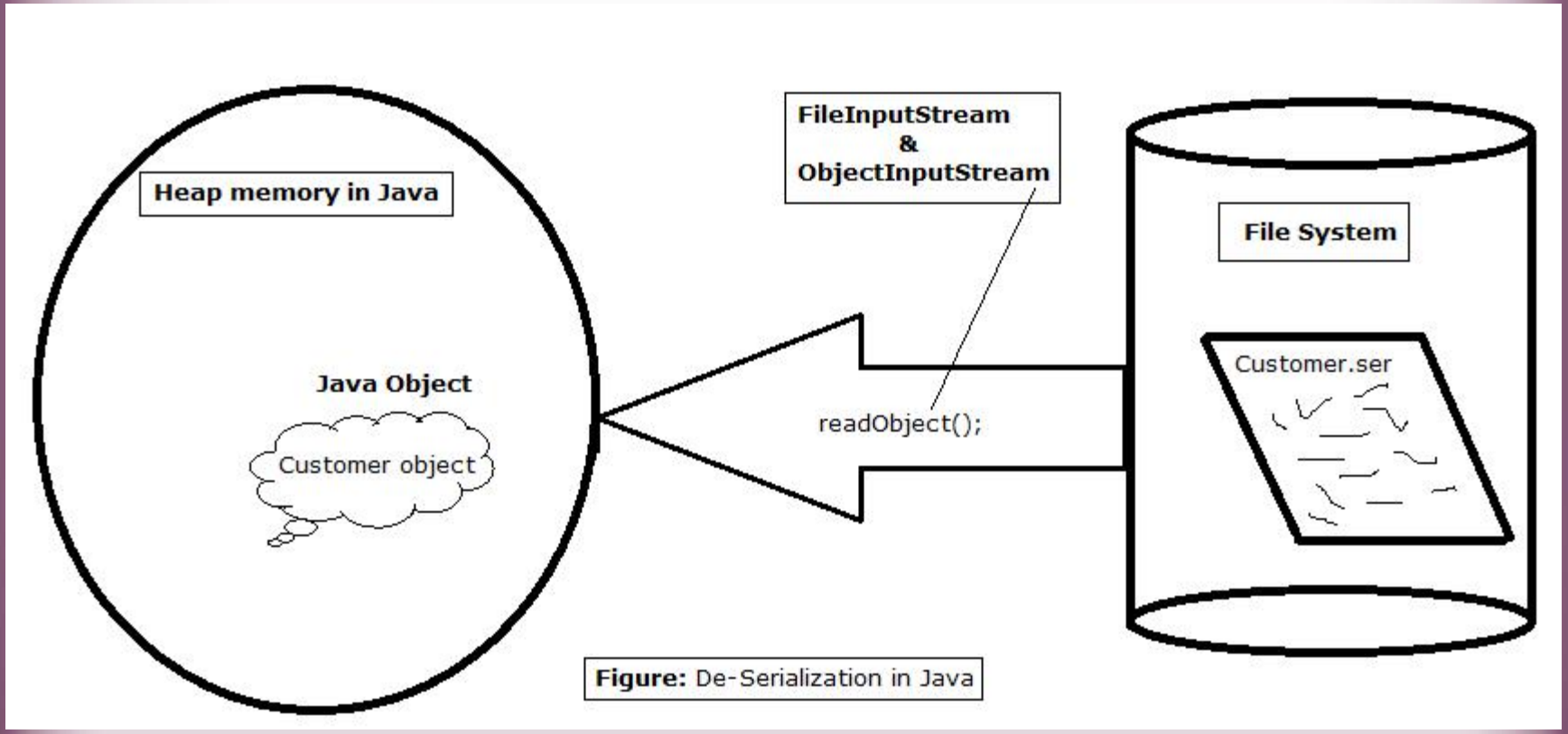


Figure: De-Serialization in Java