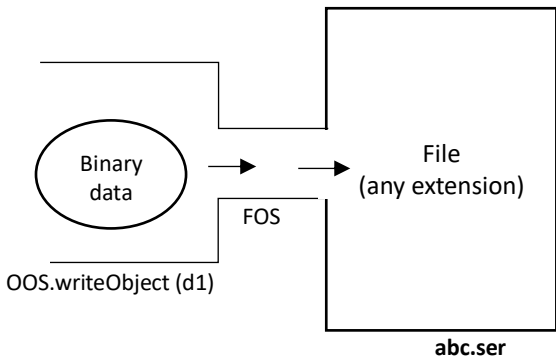
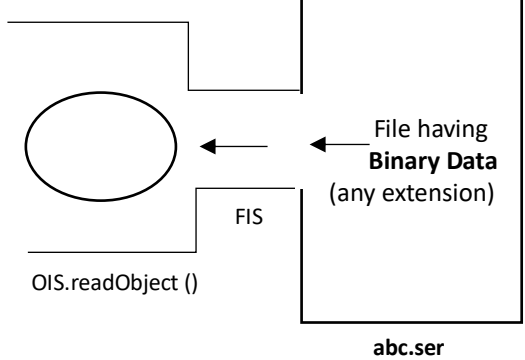


Serialization, Regular Expression, Enum, Assertion, Jar

No.	Topics
1.	Serialization intro (transient)
2.	Object graphs in serialization
3.	Customized serialization
4.	Serialization wrt inheritance
5.	Externalization
6.	SerialVersionUID
1.	Regular Expression
2.	Pattern, Matcher
3.	Character classes, Predefined character classes
4.	Quantifiers
5.	Pattern class split () method & String class split () method
6.	String tokenizer
1.	Enums (enumeration)
2.	Internal implementation of enum
3.	Enum declaration & usage
1.	javac, java, classpath
2.	jar, system Properties
3.	jar Vs war Vs ear
4.	Web application Vs Enterprise application
5.	Web server Vs Application server
6.	JDK Vs JRE Vs JVM
1.	Assertion intro
2.	assert as a keyword & identifier
3.	Types of assert statements
4.	Various possible Runtime flags
5.	Appropriate & inappropriate use of assertions
6.	AssertionError

1. Serialization intro

No.	Serialization	Deserialization
1.	The process of writing state of an object to a file is called Serialization . But strictly speaking, it is the process of converting an object from Java supported form into either File supported form or Network supported form .	The process of reading state of an object from the file is called Deserialization . But strictly speaking, it is the process of converting an object from either File supported form or Network supported form into Java supported Form .
2.	By using FileOutputStream & ObjectOutputStream classes, we can achieve / implement Serialization.	By using FileInputStream & ObjectInputStream classes, we can implement Deserialization.
3.		

- We can serialize only Serializable object. An Object is said to be Serializable iff the corresponding class implements **Serializable interface**. (Marker i/f)
- Serializable interface present in java.io package & it doesn't contain any method. It's a Marker interface.
- If we're trying to serialize a non – serializable object, then we will get **RE: NotSerializableException**.

```
import java.io.*;

class Dog implements Serializable {
    int i = 10, j = 20;
}

class SerializationDemo {
    public static void main (String [] args) throws Exception {
        Dog d1 = new Dog ();
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();

        System.out.println(d2.i + "-----" + d2.j);    // 10 ----- 20
    }
}
```

transient keyword

- transient modifier applicable only for variables but not for methods & classes.
- At the time of declaration, if we don't want to save the value of a particular variable to meet security constraints then we should declare that variable as transient.
- While performing Serialization, JVM ignores the original value of transient variable & save default value to the file. Hence **transient means not to serialize**.
- static variable is not part of Object state & hence it won't participate in Serialization. Due to this, declaring static variable as transient has no use.
- final variables participate in Serialization directly by the value. Hence, declaring final variable as transient has no use.

Declaration	O/p
int i = 10; int j = 20;	10 ---- 20
transient int i = 10; int j = 20;	0 ----- 20
transient static int i = 10; transient int j = 20;	10 ---- 0
transient int i = 10; transient final int j = 20;	0 ----- 20
transient static int i = 10; transient final int j = 20;	10 ---- 20

- We can serialize any no. of objects to the File but the order of serialization & deserialization should be same i.e., the order of object is important in Serialization.
- If we don't know the order of objects in serialization, we can use instanceof operator to check the type of object.

2. Object Graphs in Serialization

- Whenever we're serializing an Object, the set of all objects which are reachable from that object will be serialized automatically. This group of Objects is nothing but **Object Graph**.
- In Object graph, every object should be serializable. If at least one object is not serializable then we will get RE: NotSerializableException

```
class Dog implements Serializable {
    Cat c = new Cat ();
}
class Cat implements Serializable {
    Rat r = new Rat ();
}
class Rat implements Serializable {
    int j = 20;
}
class SerializationDemo {
    public static void main (String [] args) throws Exception {
        Dog d1 = new Dog ();
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();

        System.out.println(d2.c.r.j); // 20
    }
}
```

The diagram illustrates an object graph. A rounded rectangle contains four nodes: a circle labeled 'd1' with an arrow pointing to a circle labeled 'c'; a circle labeled 'c' with an arrow pointing to a circle labeled 'r'; a circle labeled 'r' with an arrow pointing to a circle labeled 'j = 20'. The label 'Object Graph' is centered below the rectangle.

- In the above program, whenever we're serializing Dog object, automatically Cat & Rat objects will get serialized because these are part of object graph of Dog.
- Among Dog, Cat, Rat objects, even if at least one object is not serializable then we will get RE: NotSerializableException.

3. Customized Serialization

- During default Serialization, there may be a chance of loss of information because of transient keyword.

```
import java.io.*;

class Account implements Serializable {
    String username = "sam";
    transient String pwd = "anushka";
}

class CustSerializationDemo {
    public static void main (String [] args) throws Exception {
        Account a1 = new Account ();
        System.out.println(a1.username + " ----- " + a1.pwd);    // sam ----- anushka
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account)ois.readObject();
        System.out.println(a2.username + " ----- " + a2.pwd);    // sam ----- null
    }
}
```

- In the above example, before Serialization, Account object can provide proper username & password but after deserialization, Account object can provide only username but not password. This is because pwd variable is declared as transient.
- Hence, during default Serialization there may be a chance of loss of information because of transient keyword.
- To recover this loss of information, we should go for Customized Serialization.
- We can implement Customized Serialization by using the following 2 methods:
 - 1) **private void writeObject (ObjectOutputStream oos) throws Exception**
 - This method will be executed automatically at the time of serialization. Hence at the time of Serialization if we want to perform any activity, we have to define that in this method only.
 - 2) **private void readObject (ObjectInputStream ois) throws Exception**
 - This method will be executed automatically at the time of deserialization. Hence, at the time of deserialization, if we want to perform any activity, we have to define that in this method only.
- The above methods are Call – back methods because these are executed automatically by the JVM.

E.g.1,

```
import java.io.*;

class Account implements Serializable {
    String username = "sam";
    transient String pwd = "anushka";

    private void writeObject (ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String epwd = "123" + pwd;
        oos.writeObject(epwd);
    }

    private void readObject (ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        String epwd = (String)ois.readObject();
        pwd = epwd.substring(3);
    }
}
```

```

class CustSerializationDemo {
    public static void main (String [] args) throws Exception {
        Account a1 = new Account ();
        System.out.println(a1.username + " ----- " + a1.pwd);    // sam ----- anushka
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account)ois.readObject();
        System.out.println(a2.username + " ----- " + a2.pwd);    // sam ----- anushka
    }
}

```

- In the above problem, before serialization & after serialization, Account object can provide proper username & password.
- Programmer can't call private methods directly from outside of the class but JVM can call private methods directly outside of the class.

E.g.2,

```

import java.io.*;

class Account implements Serializable {
    String username = "sam";
    transient String pwd = "anushka";
    transient int pin = 1234;
    private void writeObject (ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String epwd = "123" + pwd;
        oos.writeObject(epwd);
        int epin = 4444 + pin;
        oos.writeInt(epin);
    }

    private void readObject (ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        String epwd = (String)ois.readObject();
        pwd = epwd.substring(3);
        int epin = ois.readInt();
        pin = epin - 4444;
    }
}

class CustSerializationDemo {
    public static void main (String [] args) throws Exception {
        Account a1 = new Account ();
        System.out.println(a1.username + " ----- " + a1.pwd);    // sam ----- anushka
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account)ois.readObject();
        System.out.println(a2.username + " ----- " + a2.pwd);    // sam ----- anushka
    }
}

```

4. Serialization wrt Inheritance

Rule 1: Serializable nature is inheriting from Parent to Child. Hence, if parent is serializable then by default every child is serializable even though child class doesn't implement Serializable i/f.

Rule 2: Even though Parent class doesn't implement Serializable i/f, we can serialize child class object if child class implements Serializable i/f i.e., to serialize child class object, Parent class need not to be Serializable.

Rule 3: At the time of Serialization, JVM will check is any variable inheriting from Non – Serializable parent or not. If any variable inheriting from non – serializable parent then JVM ignores original value & save default value to the file.

Refer notes for more

5. Externalization

- In Serialization, everything is taken care by JVM & Programmer doesn't have any control.
- In Serialization, it's always possible to save total object to the file & it's not possible to save part of the object, which may create performance problems.
- To overcome this problem, we should go for Externalization.
- The main adv. of Externalization over Serialization is everything is taken care by Programmer & JVM doesn't have any control.
- Based on our requirement, we can save either total object or part of the object, which improves performance of the system.
- To provide Externalizable activity for any Java object, compulsory the corresponding class should implement Externalizable i/f.
- Externalizable i/f defines 2 methods: **writeExternal ();** **readExternal ();**
- Externalizable i/f is the child i/f of Serializable.

Serialization Vs Externalization

No.	Serialization	Externalization
1.	It is meant for default Serialization	It is meant for Customized Serialization.
2.	Here, everything is taken care by JVM & Programmer doesn't have any control.	Here, everything is taken care by Programmer & JVM doesn't have any control.
3.	In this case, it is always possible to save total object to the file & it's not possible to save part of the object.	Based on our requirement, we can save either total object or part of the object.
4.	Relatively Performance is low	Relatively Performance is high.
5.	It is the best choice if we want to save total object to the file.	It is the best choice if we want to save part of the object to the File.
6.	Serializable i/f doesn't contain any method & it is a marker i/f.	Externalizable i/f contains 2 methods writeExternal () & readExternal ().
7.	Serializable implemented class not required to contain public no – arg constructor.	Externalizable implemented class should compulsorily contain public no – arg constructor other RE: InvalidClassException.
8.	transient keyword will play role in Serialization.	transient keyword won't play any role in Externalization.

6. serialVersionUID

- In Serialization, both sender & Receiver need not be same person, need not to use same machine & need not be from same location, the persons may be different, machines may be different & locations may be different.
- In Serialization, both sender & receiver should have .class file at the beginning only, Just the state of object (i.e., instance variable value) is travelling from sender to receiver.
- At the time of serialization, for every Object sender – side JVM will save a unique identifier. JVM is responsible to generate this unique identifier based on .class file.
- At the time of deserialization, Receiver – side JVM will compare unique identifier associated with the object with local class unique identifier.
- If both are matched only then deserialization will be performed otherwise we will get **RE: InvalidClassException**.
- This unique identifier is nothing but **SerialVersionUID**.

Problems of depending on default serialVersionUID generated by JVM

- a) Both sender & receiver should use same JVM wrt vendor, platform & version otherwise receiver will unable to deserialize because of different serialVersionUID.
- b) Both sender & receiver should use same .class file version. After serialization if there is any change in .class file at receiver's side the receiver will unable to deserialize.
- c) To generate serialVersionUID internally JVM may use complex algorithm which may create performance problems.

We can solve above problems by configuring our own serialVersionUID by using:

```
private static final long SerialVersionUID = 1l;
```

1.	Regular Expression
2.	Pattern, Matcher
3.	Character classes, Predefined character classes
4.	Quantifiers
5.	Pattern class split () method & String class split () method
6.	String tokenizer

1. Regular Expression

- If we want to represent a group of Strings according to a particular pattern then we should go for Regular Expression.
- E.g.1. We can write a Regular Expression to represent all valid Mobile number.
- E.g.2. We can write a Regular Expression to represent all mail ids.
- The main important application areas of Regular Expressions are:
 - a) To develop validation framework.
 - b) To develop pattern matching application.
 - c) To develop translators like Compiler, Assembler, interpreters etc.
 - d) To develop digital circuits.
 - e) To develop communication protocol like TCP/IP, UDP etc.

2. Pattern & Matcher

Pattern

- A Pattern object is a compiler version of Regular Expression i.e., it's a java equivalent object of Pattern.
- We can create a Pattern object by using compile () method of Pattern class

```
public static Pattern compile (String re)
```

E.g., Pattern p = Pattern.compile ("ab");

Matcher

- We can use Matcher object to check the given Pattern in the target String.
- We can create a Matcher object by using matcher () method of Pattern class.

```
public Matcher matcher (String target)
```

E.g. Matcher m = p.matcher ("ababab");

Imp. methods of Matcher class

No.	Methods	Description
1.	boolean find ()	It attempts to find next Match & returns true if it is available else false.
2.	int start ()	Returns start index of the match
3.	int end ()	Returns (end + 1) index of the match.
4.	String group ()	Returns the matched pattern.

- Pattern & Matcher classes present in java.util.regex package & introduced in 1.4 version.

3. Character classes

Predefined Character class				
No.	Character classes	Description	Character classes	Description
1.	[a, b, c]	Either 'a' or 'b' or 'c'	\s	Space Character
2.	[^abc]	Except 'a' & 'b' & 'c'	\S	Except space Character
3.	[a-z]	Any lowercase alphabet symbol from a to z	\d	Any digit from 0 to 9
4.	[A-Z]	Any uppercase alphabet symbol from A to Z	\D	Except digit, any character
5.	[a-zA-Z]	Any alphabet symbol	\w	Any word character [0-9a-zA-Z]
6.	[0-9]	Any digit from 0 to 9	\W	Except word character [Special Character]
7.	[0-9a-zA-Z]	Any alphanumeric symbol	.	Any character
8.	[^0-9a-zA-Z]	Except alphanumeric characters [Special Symbols]		

4. Quantifiers

- We can use Quantifiers to specify number of occurrences to match.

No.	Quantifier	Description
1.	a	Matches exactly one 'a'
2.	a+	Matches at least one 'a'
3.	a*	Matches any no. of a's including zero number
4.	A?	Matches at most one 'a'

5. Pattern class **split ()** method Vs String class **split ()** method

- We can use Pattern class split () method to split the target String according to a particular Pattern.
- String class also contains split () method to split the target String according to a particular pattern.
- Pattern class split () method can take target String as Argument whereas String class split () method can take pattern as argument.

6. StringTokenizer

- It is specially designed class for Tokenization activity.
- StringTokenizer is present in java.util package.
- The default Regular expression for StringTokenizer is "space"

Enums (Enumeration)

- If we want to represent a group of Named constants then we should go for enum.
- The main objective of enum is to define our own language (Enumerated data type)
E.g. enum Month {JAN, FEB, MAR, ..., DEC};
- enum concept is introduced in 1.5 version
- When compared with old language enum, Java enum is more powerful.

Internal Implementation of enum

- 1) Every enum is internally implemented by using class concept.
- 2) Every enum constant is always public, static, final.
- 3) Every enum constant represents an Object of the type enum.

enum declaration & usage

- Every enum constant is always public, static, & final and hence, we can access enum constants by using enum name.
- Inside enum, toString () method is internally implemented to return name of the constant.
- We can declare enum either within the class or outside of the class but not inside a method. If we're trying to declare inside a method then we will get **CE: enum types must not be local.**
- If we declare enum outside of the class, the applicable modifiers are public, <default>, strictfp.
- If we declare enum inside a class, the applicable modifiers are public, <default>, strictfp, private, protected, static.

enum Vs switch

- Until 1.4 version, the allowed argument types for the switch statement are byte, short, char, int but from 1.5 version onwards corresponding wrapper & enum types are allowed.
- From 1.7 version onwards, String types also allowed.

enum Vs Inheritance

- Every enum is always direct child class of java.lang.Enum & hence our enum can't extend any other enum (becoz java won't provide support for multiple inheritance)
- Every enum is always final implicitly & hence for our enum, we can't create child enum.
- Because of above reasons we can conclude inheritance concept is not applicable for enum explicitly & we can't use extends keyword for enum.
- Anyway, an enum can implement any no. of i/fs

java.lang.Enum

- Every enum in Java is the direct child class of java.lang.Enum & hence this class acts as base class for all java enums.
- It is an abstract class & it is direct child class of Object.

value () method

- Every enum implicitly contains value () method to list out all values present inside enum.
e.g. Beer[] b = Beer.values();
- value () method is not present in Object or Enum class. enum keyword implicitly provides this method.

ordinal () method

- Inside enum, order of constants is important & we can represent this order by using ordinal value.
- We can find ordinal value of enum constant by using ordinal () method
- Ordinal value is 0 – based like array_index.

public final int ordinal ()

Speciality of Java enum

- In old language enum, we can take only constants but in java enum, in addition to constants we can take methods, constructors, normal variables etc., hence Java enum is more powerful than old language enum.
- Even inside java enum, we can declare main () method & we can run enum class directly from command prompt.
- In addition to constants, if we're taking any extra member like a method then list of constants should be in the 1st line & should end with ;

enum Vs Constructor

- An enum can contain constructor
- Enum constructor will be executed separately for every enum constants at the time of enum class loading automatically.
- We can create enum object directly & hence we can't invoke enum constructor directly.
- Inside enum, we can declare methods but should be concrete methods only & we can't declare abstract methods.

enum Vs Enum Vs Enumeration

enum – is a keyword in java which can be used to define a group of named constants.

Enum – is a class in java present in java.lang package. Every enum in java should be direct child class of Enum class.

Enumeration – is an i/f present in java.util package. We can use Enumeration object to get object one by one from the Collection.