1. **What is Serialization? Explain it with one example.**

In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example, as an argument or the result of an RMI. Deserialization consists of restoring the state of an object or a set of objects from their serialized form. It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form. Therefore some information about the class of each object is included in the serialized form.

This information enables the recipient to load the appropriate class when an object is deserialized. The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name  
of the class and its instance variables, methods and interfaces. The process that deserializes an object can check that it has the correct version of the class.

Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles*. In this case, the handle is a reference to an object within the serialized form – for example, the next number in a sequence of positive integers. The serialization procedure must ensure that there is a 1–1 correspondence between object references and handles. It must also ensure that each object is written once only – on the second or subsequent occurrence of an object, the handle is written instead of the object.

To serialize an object, its class information is written out, followed by the types and names of its instance variables. If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables. This recursive procedure continues until the class information and types and names of the instance variables of all of the necessary classes have been written out. Each class is given a handle, and no class is written more than once to the stream of bytes (the handles being written instead where necessary).

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class. Strings and characters are written by its *writeUTF* method using the Universal Transfer Format (UTF-8), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

1. **What is marshalling and de-marshalling?**

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

Thus marshalling consists of the translation of structured data items and primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

1. **How parameter passing will be done in JAVA RMI?**

In Java RMI, the parameters of a method are assumed to be input parameters and the result of a method is a single output parameter. Java serialization is used for marshalling arguments and results in Java RMI. Any object that is serializable – that is, that implements the Serializable interface – can be passed as an argument or result in Java RMI. All primitive types and remote objects are serializable. Classes for arguments and result values are downloaded to the recipient by the RMI system where necessary.

Passing remote objects:

When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference.

Passing non-remote objects:

All serializable non-remote objects are copied and passed by value. When an object is passed by value, a new object is created in the receiver’s process. The methods of this new object can be invoked locally, possibly causing its state to differ from the state of the original object in the sender’s process.

1. **What is the dynamic class loading in JAVA RMI?**

The proxy just described is static, in the sense that its class is generated from an interface definition and then compiled into the client code. Sometimes this is not practical, though. Suppose that a client program receives a remote reference to an object whose remote interface was not available at compile time. In this case it needs another way to invoke the remote object.

Dynamic invocation gives the client access to a generic representation of a remote invocation like the doOperation method used, which is available as part of the infrastructure for RMI. The client will supply the remote object reference, the name of the method and the arguments to doOperation and then wait to receive the results.

Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. We have seen that non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs. If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically. Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically. This has two advantages:

* There is no need for every user to keep the same set of classes in their working  
  environment.
* Both client and server programs can make transparent use of instances of new  
  classes whenever they are added.

1. **Explain Static versus Dynamic remote method invocations?**

Static remote method invocation

* Predefined interface definition
* Interface of an object is known when the client application is being developed
* If interface changes, then the client application must be recompiled before it can make use of the new interfaces
* Ex. Fobject.append(int) //file object

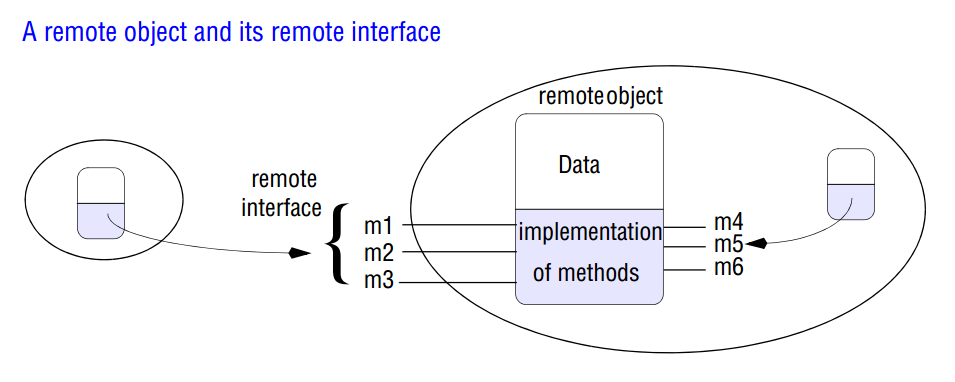
Dynamic remote method invocation

* Select at runtime which method it will invoke
* General form: invoke(object, method, input\_parameters, output\_parameters)
* Ex. invoke(fobject, id(append), int)

1. **Which interfaces and classes are used in java to implement RMI?**

Remote interfaces in Java RMI

Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi* package. The methods must throw *RemoteException*, but application-specific exceptions may also be thrown.



Consider the interface *Shape* first: the *getVersion* method returns an integer, whereas the *getAllState* method returns an instance of the class *GraphicalObject*. Now consider the interface *ShapeList*: its *newShape* method passes an instance of *GraphicalObject* as its argument but returns an object with a remote interface (that is, a remote object) as its result.

An important point to note is that both ordinary objects and remote objects can appear as arguments and results in a remote interface. The latter are always denoted by the name of their remote interface. In the next subsection, we discuss how ordinary objects and remote objects are passed as arguments and results.

Java classes supporting RMI

The only class that the programmer need be aware of is UnicastRemoteObject, which every simple servant class needs to extend. The class UnicastRemoteObject extends an abstract class called RemoteServer, which provides abstract versions of the methods required by remote servers. UnicastRemoteObject was the first example of RemoteServer to be provided. Another called Activatable is available for providing activatable objects. Further alternatives might provide for replicated objects.

The class RemoteServer is a subclass of RemoteObject that has an instance variable holding the remote object reference and provides the following methods:

**equals:** This method compares remote object references.

**toString:** This method gives the contents of the remote object reference as a String.  
**readObject, writeObject:** These methods deserialize/serialize remote objects.  
In addition, the instanceOf operator can be used to test remote objects.