

- Change the Customer class clone() method as follows:

```
public class Customer implements Cloneable
{
    Activity _activity;
    ...
    public Customer clone() throws CloneNotSupportedException
    {
        Customer clonedCustomer = (Customer) super.clone();

        // -- Clone the object referenced objects ---
        if ( _activity != null )
        {
            clonedCustomer.setActivity( (Activity) _activity.clone() );
        }
        return clonedCustomer;
    }
}
```

Note that only mutable objects needs to be cloned. References to unchangeable objects such as String be used in the cloned object without worry.

Creating object receiving from a remote source

When an object is sent through a network, the object needs to be **recreated** at the receiving host.

Object Serialization

The term Object Serialization refers to the act of converting the object to a byte stream. The byte stream can be stored on the file system, or can be sent through a network.

At the later time the object can be re-created from that stream of bytes. The only requirement is that the same class has to be available at both times, when the object is serialized and also when the object is re-created. If that happens in different servers, then the same class must be available on both servers. Same class means that exactly the same version of the class must be available, otherwise the object won't be able to be re-created. This is a maintenance problem to those applications where java serialization is used to persist object or sent the object through the network.

When a class is modified, there could be a problem re-creating those objects that were serialized using an earlier version of the class.

Java has built in support for serialization, using the Serializable interface; however, a class must first implement the Serializable interface.

By default, a class will have all of its fields serialized when converted into a data stream (with transient fields being skipped.) If additional handling is required beyond the default of writing all fields, you need to provide an implementation for two methods:

```
private void writeObject(java.io.ObjectOutputStream out) throws
IOException;
private void readObject(java.io.ObjectInputStream in) throws
IOException, ClassNotFoundException;
private void readObjectNoData() throws ObjectStreamException;
```

If the object needs to write or provide a replacement object during serialization, it needs to implement the following two methods, with any access specifier:

```
Object writeReplace() throws ObjectOutputStreamException;
Object readResolve() throws ObjectOutputStreamException;
```

Normally, a minor change to the class can cause the serialization to fail. You can still allow the class to be loaded by defining the serialization version id:

```
private static final long serialVersionUID = 42L;
```

Interfaces

Navigate **Classes and Objects** topic:

- Defining classes
- Creating objects
- Interfaces
- Using static members
- Destroying objects
- Overloading methods and constructors
- Class Loading

Interfaces

Java does not allow you to create a subclass from two classes. There is no multiple inheritance. The major benefit of that is that all java objects can have a common ancestor. That class is called `Object`. All java classes can be up-casted to `Object`. Example:

```
class MyObject
{
    ...
}
```

When you type the above code, it actually means the following:

```
class MyObject extends Object // -- The compiler adds 'extends Object'.
    if not specified
{
    ...
}
```

So it can be guaranteed that certain methods are available in all java classes. This makes the language simpler.

To mimic multiple inheritance, java offers *interfaces*, which are similar to abstract classes. In interfaces all methods are abstract by default, without the *abstract* key word. Interfaces have no implementation and no variables, but constant values can be defined in interfaces - however, a single class can implement as many interfaces as required.

```
public interface MyInterface
{
    public static final String CONSTANT = "This value can not be
changed";
```

```
    public String methodOne(); // This method must be implemented by
    class who implements this interface
    ...
}
...
public class MyObject implements MyInterface
{
    // --- Implement MyInterface interface
    public String methodOne()
    {
        ...
    }
}
```

External links

- [Interfaces, interactive Java Lesson](#) ^[30]
- [Interfaces 2, interactive Java Lesson](#) ^[31]

Using Static Members

Navigate **Classes and Objects** topic:

- [Defining classes](#)
- [Creating objects](#)
- [Interfaces](#)
- [Using static members](#)
- [Destroying objects](#)
- [Overloading methods and constructors](#)
- [Class Loading](#)

What does static mean?

When you declare a

- method, or
- member variable

static, it is independent of any particular, but rather it is shared among all instances of a class. To access a static method or member variable, no instance needs to be created.

The **static** keyword is used to declare a method, or member variable static.