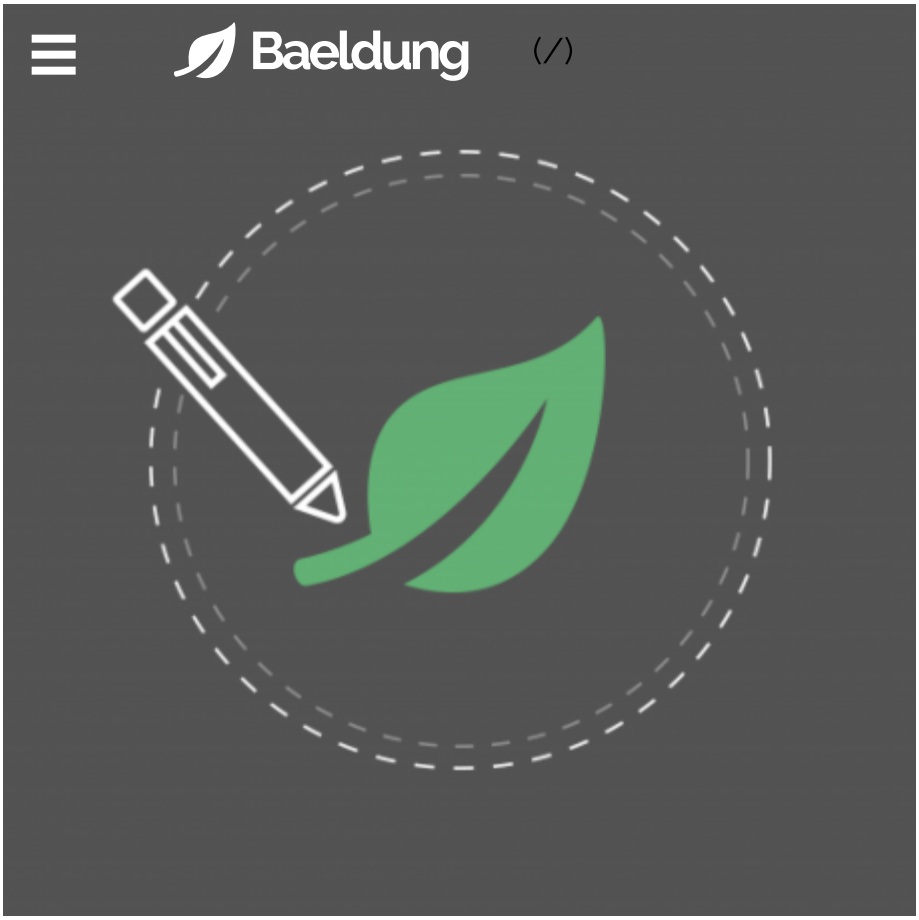
(/)

Java Serialization: readObject() vs. readResolve()

Last updated: December 7, 2023



Written by: baeldung (https://www.baeldung.com/author/baeldung)

Java (https://www.baeldung.com/category/java) +

Core Java (https://www.baeldung.com/tag/core-java)

Serialization (https://www.baeldung.com/tag/serialization)

1. Overview (/)

In this tutorial, we'll look at how to use *readObject()* and *readResolve()* methods in the Java deserialization API. Furthermore, we'll examine the difference between these two methods.

2. Serialization

Java Serialization (/java-serialization) covers how serialization and deserialization work in greater depth. In this article, we'll focus on the *readResolve()* and *readObject()* methods, which frequently raise questions when employing deserialization.

3. Use of *readObject()*

A Java object is converted into a stream of bytes during serialization to be saved in a file or transferred over the internet. The serialized stream of bytes is transformed back into the original object during deserialization using *ObjectInputStream*'s *readObject()* method, which internally calls *defaultReadObject()* for default deserialization.

If the *readObject()* method is present in our class, *ObjectInputStream*'s *readObject()* method will use our class's *readObject()* method for reading the object from the stream.

For instance, in some cases, we can implement *readObject()* in our class to deserialize any field in a specific way.

(/)

Before we present our use case, let's check the syntax for implementing the *readObject()* method in our class:

```
private void readObject(ObjectInputStream stream) throws
IOException, ClassNotFoundException;
```

Now, let's suppose we have a *User* class with two fields:

```
public class User implements Serializable {

    private static final long serialVersionUID =
3659932210257138726L;
    private String userName;
    private String password;
    // standard setters, getters, constructor(s) and toString()
}
```


Furthermore, we don't want to serialize the *password* i. c' ar text, so what can we do? Let's see how Java's *readObject()* can help us here.

3.1. Add *writeObject()* for Custom Change During Serialization

First, we can make specific changes to the object's fields during serialization,

like encoding the *password*, in the *writeObject()* method.

So, for our *User* class, let's implement *the writeObject()* method and add an extra string prefix to our password field during serialization:


```
private void writeObject(ObjectOutputStream oos) throws IOException
{
    this.password = "xyz" + password;
    oos.defaultWriteObject();
}
```

3.2. Test Without *readObject()* Implementation

Now, let's test our *User* class, but without implementing *readObject()*. In this case, the *ObjectInputStream* class's *readObject()* will be called:

```
@Test                                (/)
public void testDeserializeObj_withDefaultReadObject() throws
ClassNotFoundException, IOException {
    // Serialization
    FileOutputStream fos = new FileOutputStream("user.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    User acutalObject = new User("Sachin", "Kumar");
    oos.writeObject(acutalObject);

    // Deserialization
    User deserializedUser = null;
    FileInputStream fis = new FileInputStream("user.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    deserializedUser = (User) ois.readObject();
    assertNotEquals(deserializedUser.hashCode(),
acutalObject.hashCode());
    assertEquals(deserializedUser.getUserName(), "Sachin");
    assertEquals(deserializedUser.getPassword(), "xyzKumar");
}
```

Here, we can see that the password is *xyzKumar* as we don't yet have any *readObject()* in our class that could retrieve the original field and make custom changes.

3.3. Add *readObject()* for Custom Change During Deserialization

Next, we can make specific changes to the object's fields during deserialization, such as decoding the *password*, in the *readObject()* method.

Let's implement *the readObject()* method in our *User* class and remove the extra string prefix that we added to our password field during serialization:

```
private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException {
    ois.defaultReadObject();
    this.password = password.substring(3);
}
```

3.4. Test with *readObject()* Implementation

Let's test our *User* class again, only this time, we have a custom *readObject()* method that will be called during deserialization:


```
@Test
public void testDeserializeObj_withOverriddenReadObject() throws
ClassNotFoundException, IOException {
    // Serialization
    FileOutputStream fos = new FileOutputStream("user.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    User acutalObject = new User("Sachin", "Kumar");
    oos.writeObject(acutalObject);

    // Deserialization
    User deserializedUser = null;
    FileInputStream fis = new FileInputStream("user.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    deserializedUser = (User) ois.readObject();
    assertNotEquals(deserializedUser.hashCode(),
acutalObject.hashCode());
    assertEquals(deserializedUser.getUserName(), "Sachin");
    assertEquals(deserializedUser.getPassword(), "Kumar");
}
```


Here, we can notice a few things. First, the objects are different, and second, our custom *readObject()* is called, and the password field is converted correctly.

4. Use of *readResolve()*

In Java Deserialization, **the *readResolve()* method is used to replace the object that is created during deserialization with a different object**. This can be useful in situations where we need to ensure that only a single instance of a particular class exists in our application or when we want to replace an object with a different instance that may already exist in memory.

Let's review the syntax for adding the *readResolve()* in our class:

```
ANY-ACCESS-MODIFIER Object readResolve() throws
ObjectStreamException;
```

One thing to notice in the *readObject()* example is that the object *hashCode* is different. That's because, during deserialization, the new object gets created from the streamed object.

A common scenario where we might want to use *readResolve()* is when creating singleton instances. We can use *readResolve()* to ensure that the deserialized object is the same as the existing instance for a singleton instance.

Let's take an example of creating a singleton Object:

```
public class Singleton implements Serializable {

    private static final long serialVersionUID = 1L;
    private static Singleton INSTANCE = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```


4.1. Test Without *readResolve()* Implementation

At this point we haven't added any *readResolve()* method. Let's test our *Singleton* class:

```
@Test
public void testSingletonObj_withNoReadResolve() throws
ClassNotFoundException, IOException {
    // Serialization
    FileOutputStream fos = new FileOutputStream("singleton.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    Singleton actualSingletonObject = Singleton.getInstance();
    oos.writeObject(actualSingletonObject);

    // Deserialization
    Singleton deserializedSingletonObject = null;
    FileInputStream fis = new FileInputStream("singleton.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    deserializedSingletonObject = (Singleton) ois.readObject();
    assertNotEquals(actualSingletonObject.hashCode(),
deserializedSingletonObject.hashCode());
}
```

Here, we can see that both objects are different, defeating the goal of our *Singleton* class.

4.2. Test With *readResolve()* Implementation

To fix this, let's add the *readResolve()* method in our *Singleton* class:

```
private Object readResolve() throws ObjectStreamException {
    return INSTANCE;
}
```

Now, let's test again with the *readResolve()* method in our *Singleton* class:

```
@Test
public void testSingletonObj_withCustomReadResolve() throws
ClassNotFoundException, IOException {
    // Serialization
    FileOutputStream fos = new FileOutputStream("singleton.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    Singleton actualSingletonObject = Singleton.getInstance();
    oos.writeObject(actualSingletonObject);

    // Deserialization
    Singleton deserializedSingletonObject = null;
    FileInputStream fis = new FileInputStream("singleton.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    deserializedSingletonObject = (Singleton) ois.readObject();
    assertEquals(actualSingletonObject.hashCode(),
deserializedSingletonObject.hashCode());
}
```

Here, we can see that both objects have the same *hashCode*.

5. *readObject()* vs. *readResolve()*

Let's see a quick summary of the differences between these two:

readResolve()	*readObject()*
The method return type is Object	The method return type is void
No method parameters	*ObjectInputStream* as parameter

readResolve() (/)	*readObject()*
Typically used to implement the Singleton pattern, where the same object needs to be returned after deserialization.	Used to set the values of the object's non-transient fields that were not serialized, such as fields derived from other fields or fields that are initialized dynamically.
throws *ClassNotFoundException*, *ObjectStreamException*	throws *ClassNotFoundException*, *IOException*
Faster than *readObject()* since it does not read the entire object graph.	Slower than *readResolve()* since it reads the entire object graph.

6. Conclusion

In this article, we learned about the *readObject()* and *readResolve()* methods of the Java Serialization API. Furthermore, we've seen the difference between these two. As always, the example code for this article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-serialization).