Singleton Revisited

A Singleton

```
import java.util.Arrays;

public class Elvis {

    private static final Elvis INSTANCE = new Elvis();
    private Elvis() {}

    private final String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    public static final Elvis getInstance() {
        return INSTANCE;
    }
}
```

Is this a Singleton?

```
import java.util.Arrays;

public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() {}

    private final String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

What about Serializable?

- If the two previous classes implement java.io.Serializable, are either of the two previous classes Singletons?

- No. If you serialize an object and then deserialize it, you get a copy (an Elvis impersonator).

    - readObject returns a copy of the object.

- How to avoid copies that appear when deserializing?

Now Singleton?

```
import java.io.Serializable;
import java.util.Arrays;

public class Elvis implements Serializable {

    public static final Elvis INSTANCE = new Elvis();
    private Elvis() {}

    private final String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

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

Not Singleton

- favoriteSongs is non-transient.
- Adding a non-transient field to a “Singleton” makes it not a Singleton.
- When an object is being deserialized, it is temporarily around in memory.
- When deserializing the non-transient field, one can grab a reference to the Elvis impersonator and store it for later.
- readResolve only works if all fields are transient.
    - The *transient* keyword identifies a field that should not be serialized.

There is a better way

- A single element enum

```
import java.util.Arrays;

public enum Elvis {
    INSTANCE;

    private final String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel"};

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

Enums

- All enums are serializable
- No need for readResolve. The JVM guarantees there will never be a second instance of any of the values in an enum type.

Final Thoughts

- Be careful when implementing Serializable
    - It decreases flexibility to change the class's implementation once the class has been released
        - Byte stream encoding becomes part of API
- Increases chances of bugs and security holes
- Increases testing costs

Can you find the memory leak?