The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global point of access to it.

Basic implementation of a Singleton class:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor prevents instantiation from other classes
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

There are three major ways in which you can break a singleton pattern.

1. Reflection

Java Reflection is an API used to examine or modify the behavior of methods, classes, and interfaces at runtime. It can be used to bypass the private constructor and create multiple instances of a Singleton class.

```
public class ReflectionTest {
    public static void main(String[] args) {
        try {
            Singleton instance1 = Singleton.getInstance();
            Singleton instance2 = null;
            Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();
            constructor.setAccessible(true); // Make the private constructor accessible
            instance2 = constructor.newInstance();

            System.out.println("Instance 1: " + instance1.hashCode());
            System.out.println("Instance 2: " + instance2.hashCode());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Preventing Reflection Attack

To prevent such reflection attacks, you can modify the Singleton class to throw an exception if an instance already exists:

```
    private Singleton() {
        if (instance != null) {
            throw new IllegalStateException("Instance already created.");
        }
    }
```

2. Deserialization

In serialization, we can save the object of a byte stream into a file or send over a network. Deserialization of a Singleton object can create a new instance of the class.

```
public class SerializationTest {
    public static void main(String[] args) {
        try {
            Singleton instance1 = Singleton.getInstance();

            ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("singleton.ser"));
            out.writeObject(instance1);
            out.close();
            ObjectInputStream in = new ObjectInputStream(new FileInputStream("singleton.ser"));
            Singleton instance2 = (Singleton) in.readObject();
            in.close();

            System.out.println("Instance 1: " + instance1.hashCode());
            System.out.println("Instance 2: " + instance2.hashCode());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Preventing Deserialization Attack

To overcome this issue, we need to override `readResolve()` method in the `Singleton` class and return the same Singleton instance.

```
 protected Object readResolve() {
          return instance;
    }
```

3. Cloning

We can create a copy of the original object using the clone method. If the Singleton class implements the `Cloneable` interface, the `clone()` method can produce a new instance.

```
public class Singleton implements Cloneable {
    private static Singleton instance;

    private Singleton() {
        // Private constructor prevents instantiation from other classes
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Let's clone the Singleton object:

```
public class CloningTest {
    public static void main(String[] args) {
        try {
            Singleton instance1 = Singleton.getInstance();
            Singleton instance2 = (Singleton) instance1.clone();

            System.out.println("Instance 1: " + instance1.hashCode());
            System.out.println("Instance 2: " + instance2.hashCode());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

Preventing Cloning Attack

To overcome the above issue, we need to implement/override the `clone()` method and throw an exception `CloneNotSupportedException` from the `clone` method. If anyone tries to create a `clone` object of `Singleton`, it will throw an exception.

```
    @Override
    protected Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException("Singleton instance cannot be cloned.");
    }
```