The Question : **How session.load() Knows When a Getter Is Called ?**

How does session.load() know that we have called a getter method on the object?

When we call:

```
Student student = (Student) session.load(Student.class, 27);
System.out.println("Rno  : " + student.getRno());
System.out.println("Name : " + student.getName());
System.out.println("Per  : " + student.getPer());
```

Hibernate somehow knows exactly when we try to access entity fields (like getName()), and at that moment it fires an SQL SELECT query if the data is not yet loaded.

- The Trick: Hibernate Does Not Return Your Object Directly

When you call session.load(), Hibernate does not return your plain Student object.
Instead, it returns a proxy object, a runtime-generated subclass of Student.

Example (what Hibernate generates internally):

```
class Student_$$_javassist_0 extends Student implements HibernateProxy {
    private LazyInitializer hibernateInterceptor;

    @Override
    public String getName() {
        if (hibernateInterceptor.isUninitialized()) {
            hibernateInterceptor.initialize(); // triggers SQL SELECT
        }
        return super.getName();
    }

    @Override
    public String getPer() {
        if (hibernateInterceptor.isUninitialized()) {
            hibernateInterceptor.initialize();
        }
        return super.getPer();
    }
}
```

So the proxy is your entity + hidden interceptor logic.
That's how Hibernate knows when you call getName() → because you're not calling it on your entity, but on this special subclass.

- Step-by-Step Flow

You call

```
Student student = (Student) session.load(Student.class, 27);
```

At this point, Hibernate does not hit the database.
It creates a proxy object Student_$$_javassist_0.

You call a getter

```
student.getName();
```

This actually calls the overridden method inside the proxy.

Proxy delegates to LazyInitializer
The overridden method asks the LazyInitializer:

"Am I initialized?"

If no, Hibernate fires a SELECT query to fetch the data.

The proxy's fields are populated.

Return the value
Now the proxy calls the parent Student's getter and gives you the real data.

- Why Does Hibernate Do This?

Hibernate follows the philosophy of "POJO persistence":

You write simple entity classes (Student with fields, getters, setters).

Hibernate enhances them with proxies, but you don't have to write any special code.

Proxies allow lazy loading: only fetch data when it's actually needed.

Without proxies, Hibernate would have to fetch all related data immediately — which kills performance.

- Example with Output

Code:

```
Student student = (Student) session.load(Student.class, 27);

System.out.println("Load method called...");

System.out.println("Class returned: " + student.getClass().getName());

System.out.println("Rno  : " + student.getRno());   // triggers DB if uninitialized
System.out.println("Name : " + student.getName());  // triggers DB if uninitialized
System.out.println("Per  : " + student.getPer());   // triggers DB if uninitialized
```

Output:

```
Load method called...
Class returned: com.tca.entities.Student_$$_javassist_0
Hibernate: select student0_.RNO, student0_.NAME, student0_.PER
        from student student0_ where student0_.RNO=?
Rno  : 27
Name : Ramesh
Per  : 90.94
```

Notice that the class is not Student but Student_$$_javassist_0.

- Philosophical Explanation

Think of it like this:

Your entity (Student) is the idea.

Hibernate's proxy (Student_$$_javassist_0) is the mask that looks exactly like your entity but secretly carries tools.

When you interact with it (call a getter), the mask peeks into the database if needed, then gives you the real value.

This philosophy is what enables transparent persistence:

You use your entities like normal POJOs.

Hibernate adds intelligence invisibly through proxies.

✅ Answer in one line:
session.load() returns a proxy subclass of your entity which overrides the getters/setters. When you call a getter, Hibernate intercepts it through the proxy, checks if the entity is initialized, and fires SQL only if required.