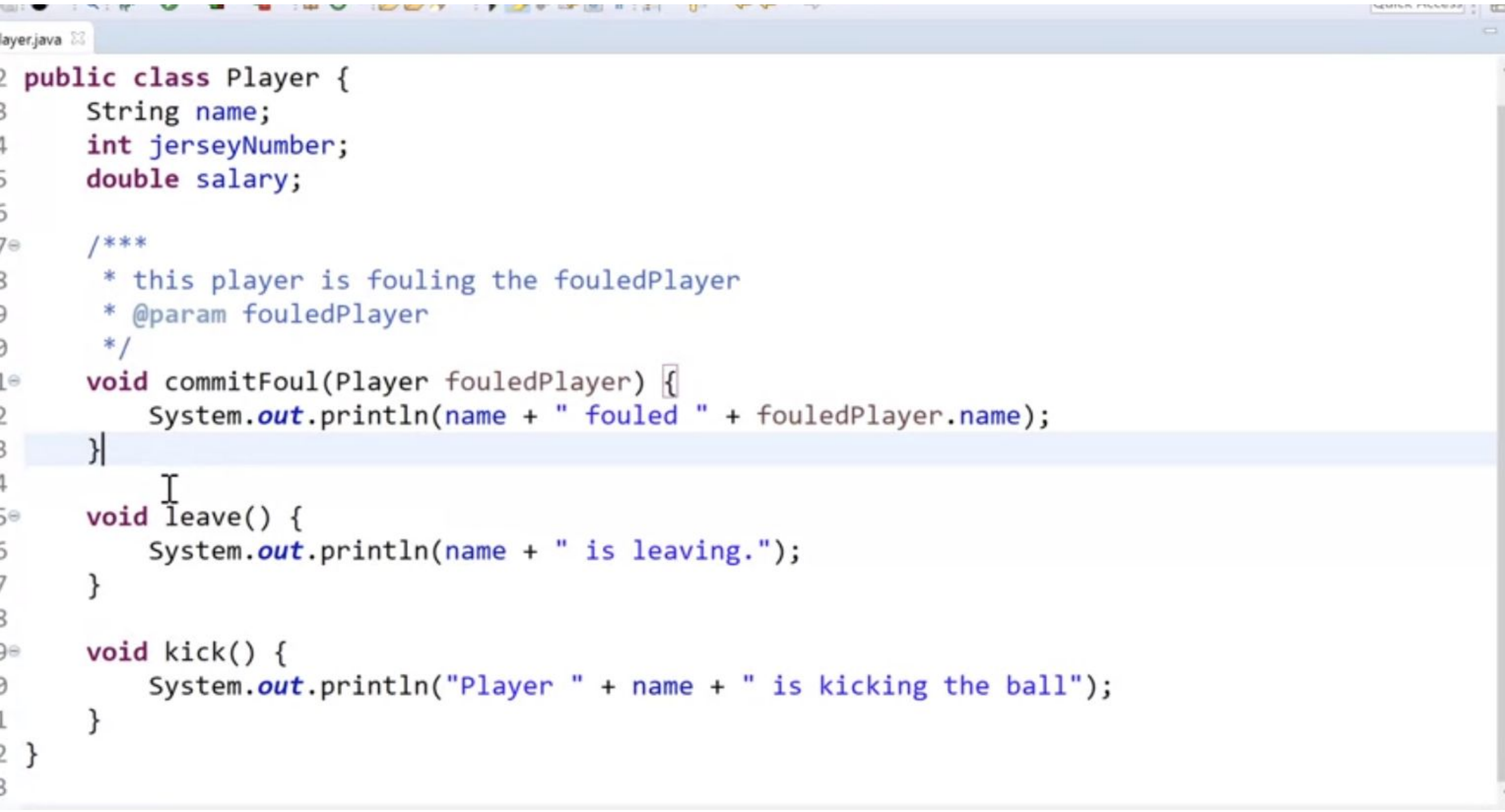
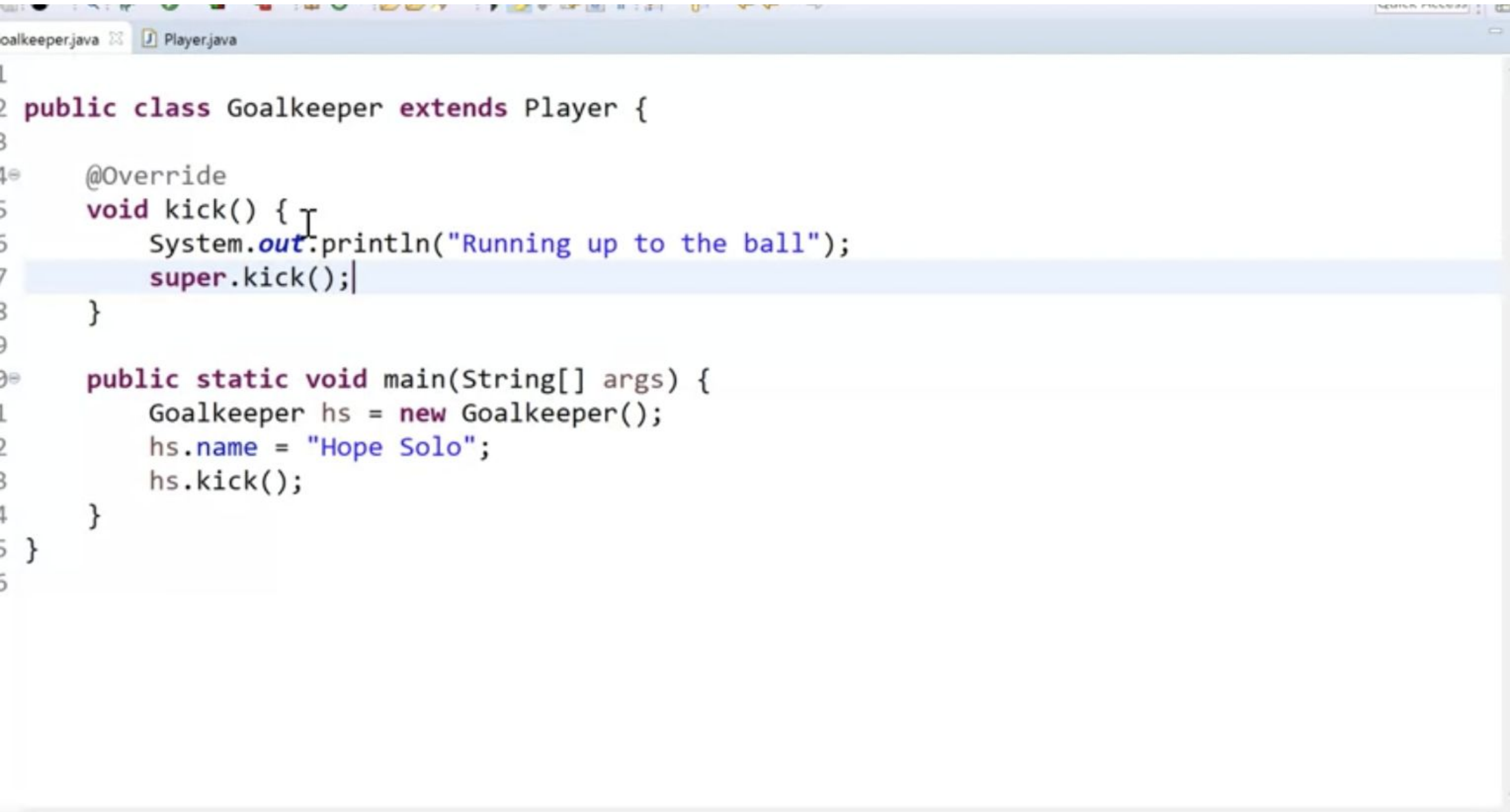
Inheritance and Hierarchies

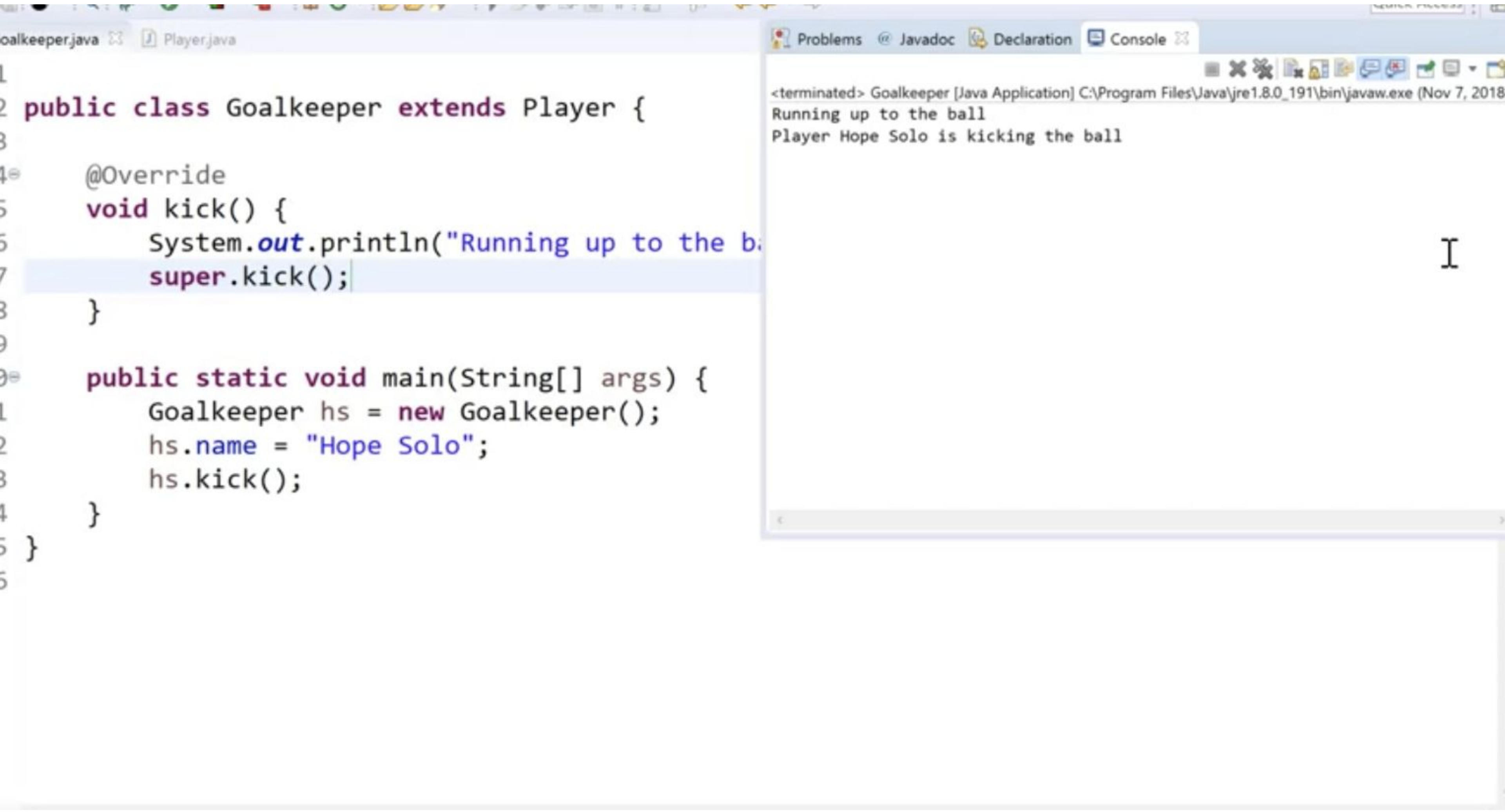
- Create Classes that inherit functionality from another Class
- Uses the `extends` keyword
  - Defines an “is a” relationship
  - Example: Goalkeeper is a Player


layer.java

```java
public class Player {
    String name;
    int jerseyNumber;
    double salary;

    /***
     * this player is fouling the fouledPlayer
     * @param fouledPlayer
     */
    void commitFoul(Player fouledPlayer) {
        System.out.println(name + " fouled " + fouledPlayer.name);
    }

    void leave() {
        System.out.println(name + " is leaving.");
    }

    void kick() {
        System.out.println("Player " + name + " is kicking the ball");
    }
}
```



```java

public class Goalkeeper extends Player {

    @Override
    void kick() {
        System.out.println("Running up to the ball");
        super.kick();
    }

    public static void main(String[] args) {
        Goalkeeper hs = new Goalkeeper();
        hs.name = "Hope Solo";
        hs.kick();
    }
}

```

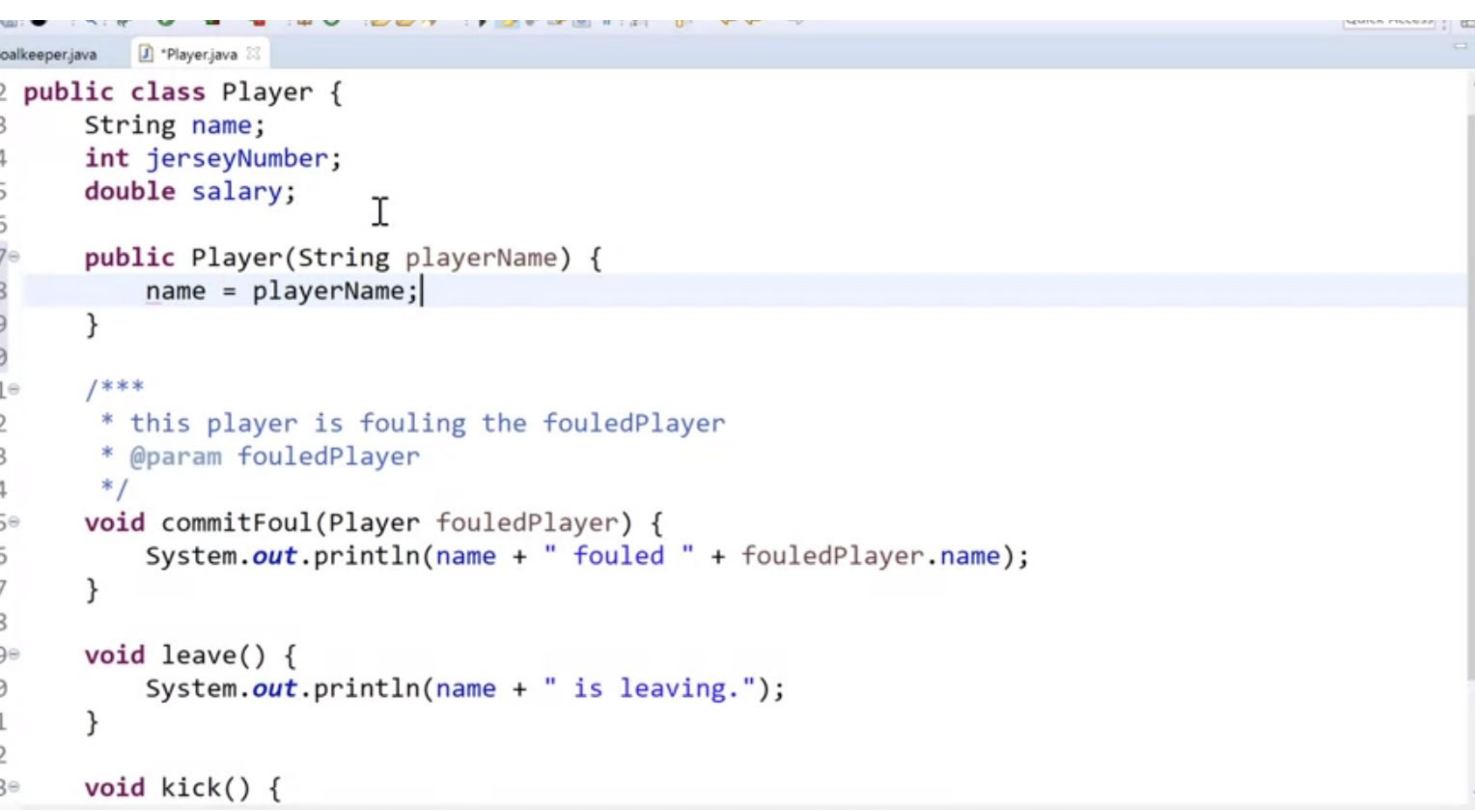


```java

public class Goalkeeper extends Player {

    @Override
    void kick() {
        System.out.println("Running up to the ba
        super.kick();
    }

    public static void main(String[] args) {
        Goalkeeper hs = new Goalkeeper();
        hs.name = "Hope Solo";
        hs.kick();
    }
}

```

```
<terminated> Goalkeeper [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (Nov 7, 2018
Running up to the ball
Player Hope Solo is kicking the ball
```

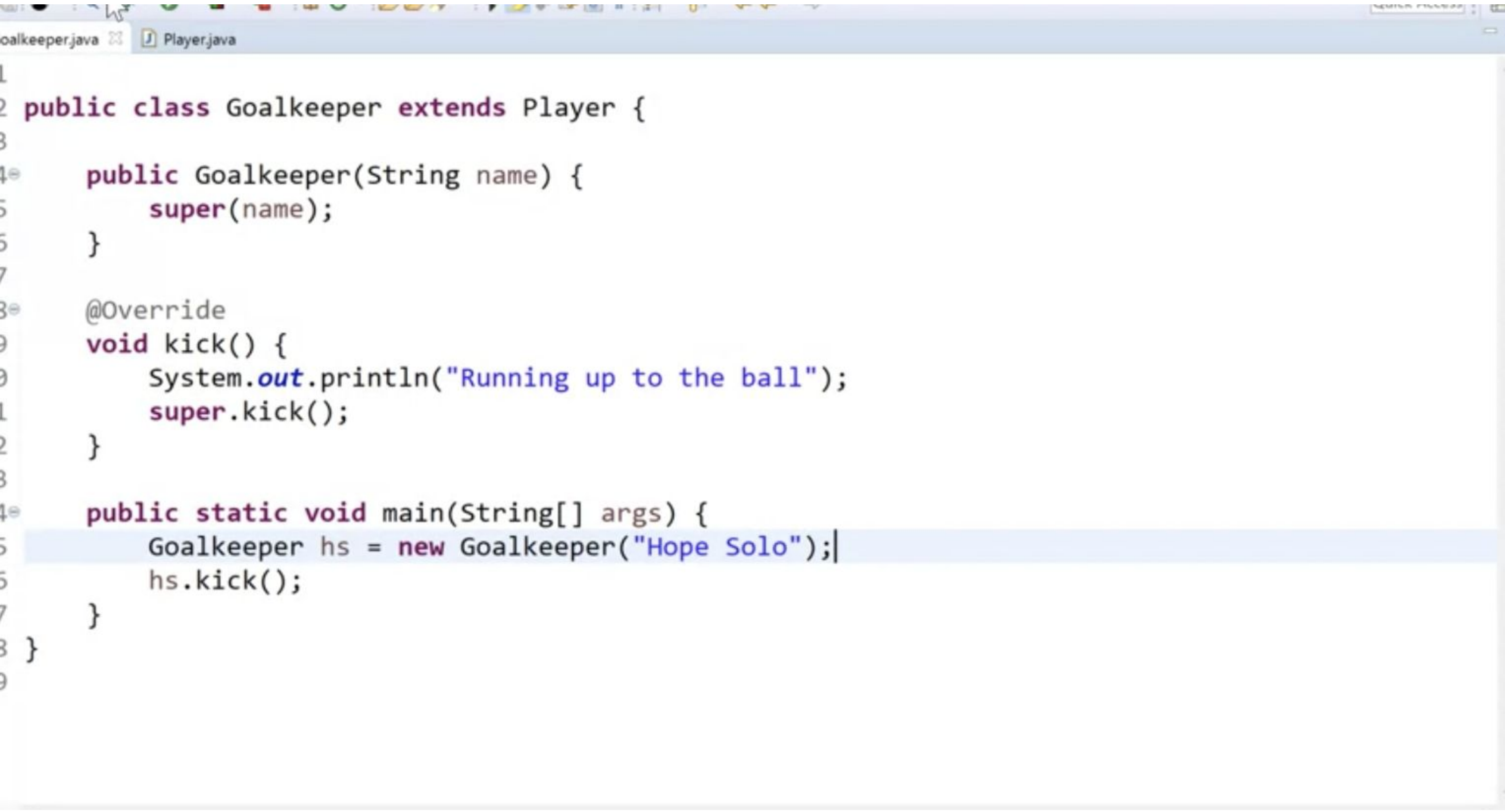
Constructor Inheritance

- Classes try calling their Parent's default constructor
  - Happens implicitly, passing no parameters
- To prevent calling the default constructor, you can use an explicit call
- Explicit call requires one of two keywords
  - `super`
  - `this`
- `super` keyword: Parent class's constructor
  - `super()` calls the Parent class's default constructor
  - `super(parameters)` calls the Parent class's constructor with parameters
- `this` keyword: Child class's constructor



oalkeeper.java   *Player.java

```java
public class Player {
    String name;
    int jerseyNumber;
    double salary;

    public Player(String playerName) {
        name = playerName;
    }

    /***
     * this player is fouling the fouledPlayer
     * @param fouledPlayer
     */
    void commitFoul(Player fouledPlayer) {
        System.out.println(name + " fouled " + fouledPlayer.name);
    }

    void leave() {
        System.out.println(name + " is leaving.");
    }

    void kick() {
```



oalkeeper.java   Player.java

```java

public class Goalkeeper extends Player {

    public Goalkeeper(String name) {
        super(name);
    }

    @Override
    void kick() {
        System.out.println("Running up to the ball");
        super.kick();
    }

    public static void main(String[] args) {
        Goalkeeper hs = new Goalkeeper("Hope Solo");
        hs.kick();
    }
}

```

The Object Class

- All Classes inherit from the Object Class
- Allows flexibility
  - Example: `HashMap<Object> myObjects` can hold anything!
- Provides common functionality
- All Objects inherit default methods
  - `toString()`
  - `equals()`

The `toString()` Method

- All Classes have a default `toString()` method, inherited from Object
- Default method prints the memory-location of that Object
- Can be overridden
- `System.out.println()` automatically calls an Object's `toString()` method

The `equals()` Method

- An Object's value is its memory location
- The default `equals()` method will compare these memory locations
- Should usually be overridden for your class

Equality or `equals()`?

Primitives

- Can always be tested for equality with `==`

```
double x = 2.0;
double y = 3.1;
if( int x == y ){ ... }
```

Objects

- Using `==` for Objects will *always* use the memory location
- Using `.equals()` for Objects will also use the memory location, unless Overridden
- `public boolean equals(Object o)`

Comparing Strings

- The String Class provides the `equals()` method and should be used
- Should be tested with the `equals()` method

```
String a = new String("Hello!");
String b = new String("Hello!");
a == b;          // Returns False
a.equals(b);     // Returns True
```

Summary of Polymorphism

- **Overloading** - two methods with the same name in the same class because they do similar things.
  - usually you see common code within the same class.
- **Overloading, inheritance, abstract classes** - common code across two or more classes.
  - take the common code and move it to a parent class.