LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Reuse

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Polymorphism
- Interfaces
- Collections
- Functional Programming
- Stream
- Exceptions
- Generics
- Arrays

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon
- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

ONLINE VIDEO TUTORIAL

- Learning Programming In Java
  - YouTube Tutorials
- Lab Exercises
  - http://www.learnprogramminginjava.com/

TOPIC DESCRIPTION

Composition is a critical design principle used to build complex classes by combining simpler ones. It allows one class to contain references to objects of other classes, enabling the creation of more complex data structures. This relationship signifies a "has-a" relationship, where one class is composed of one or more instances of other classes.

Benefits of Composition:

- Flexibility: Easily change or replace components without modifying the containing class.
- Encapsulation: Each component can encapsulate its own behavior and data.
- Reusability: Components can be reused in different contexts without duplicating code.

Composition is almost always preferred over inheritance, as it allows for greater flexibility and modularity in code structure.

TOPIC DESCRIPTION

Inheritance: a fundamental object-oriented programming (OOP) principle allowing a new class (a subclass) to inherit properties and behaviors (methods) from an existing class (called a superclass or base class). This relationship is often referred to as an "is-a" relationship, where the subclass is a more specific version of the superclass.

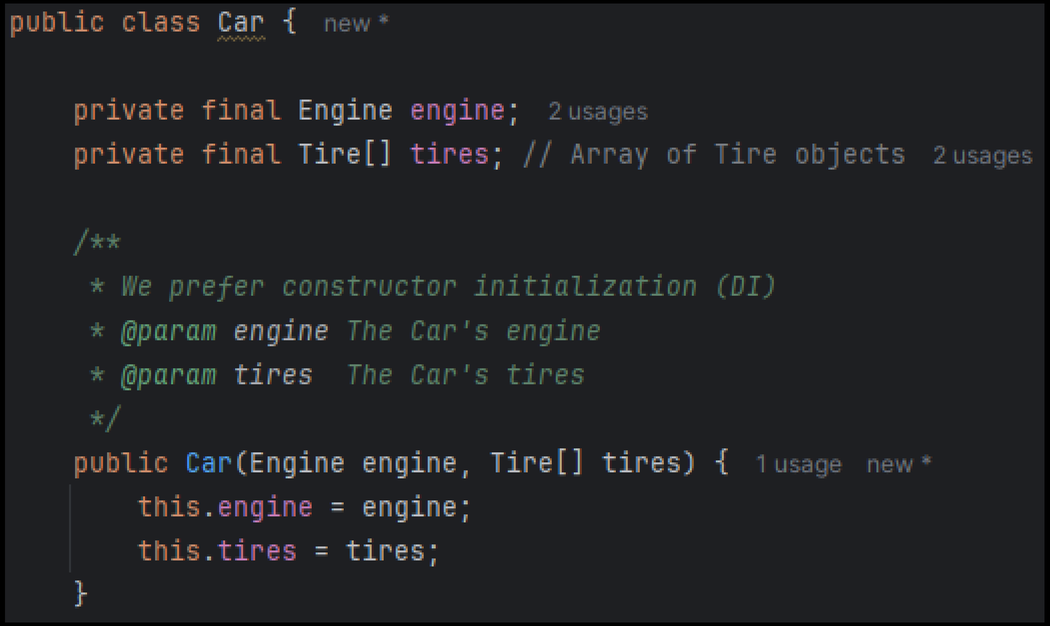
Benefits of Inheritance:

- Flexibility: Subclasses can reuse code from the superclass, reducing redundancy and promoting code maintainability.
- Method Overriding: Subclasses can provide a specific implementation for methods that are already defined in the superclass, allowing for polymorphic behavior.
- Classification: enables the creation of a hierarchy of classes, which makes it easier to organize and manage code. (So, they say)

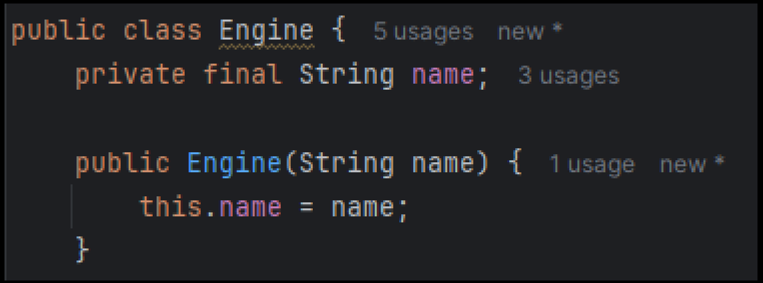
COMPOSITION EXAMPLE

Composition

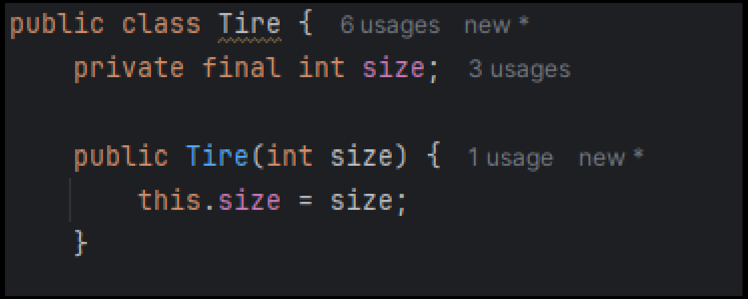
For example, consider a `Car` class that has an `Engine` class and `Tire` class as its components:



```
public class Car {  new *

    private final Engine engine;  2 usages
    private final Tire[] tires; // Array of Tire objects  2 usages

    /**
     * We prefer constructor initialization (DI)
     * @param engine The Car's engine
     * @param tires  The Car's tires
     */
    public Car(Engine engine, Tire[] tires) {  1 usage  new *
        this.engine = engine;
        this.tires = tires;
    }
```



```
public class Engine {  5 usages  new *
    private final String name;  3 usages

    public Engine(String name) {  1 usage  new *
        this.name = name;
    }
```

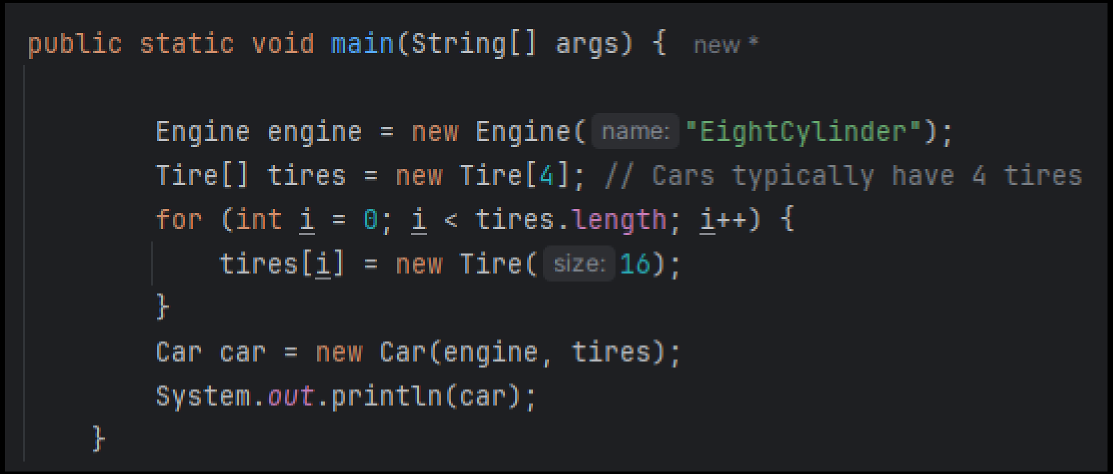


```
public class Tire {  6 usages  new *
    private final int size;  3 usages

    public Tire(int size) {  1 usage  new *
        this.size = size;
    }
```

COMPOSITION EXAMPLE

Composition

For example, consider a `Car` class that has an `Engine` class and `Tire` class as its components:

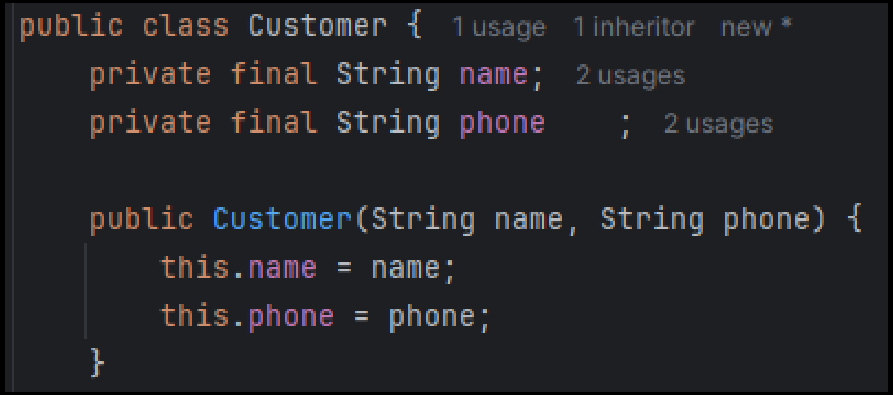


```
public static void main(String[] args) {  new *

        Engine engine = new Engine( name: "EightCylinder");
        Tire[] tires = new Tire[4]; // Cars typically have 4 tires
        for (int i = 0; i < tires.length; i++) {
            tires[i] = new Tire( size: 16);
        }
        Car car = new Car(engine, tires);
        System.out.println(car);
    }
```

Recall we prefer constructor initialization over default constructors. This way we only construct valid objects and do not allow for nulls.

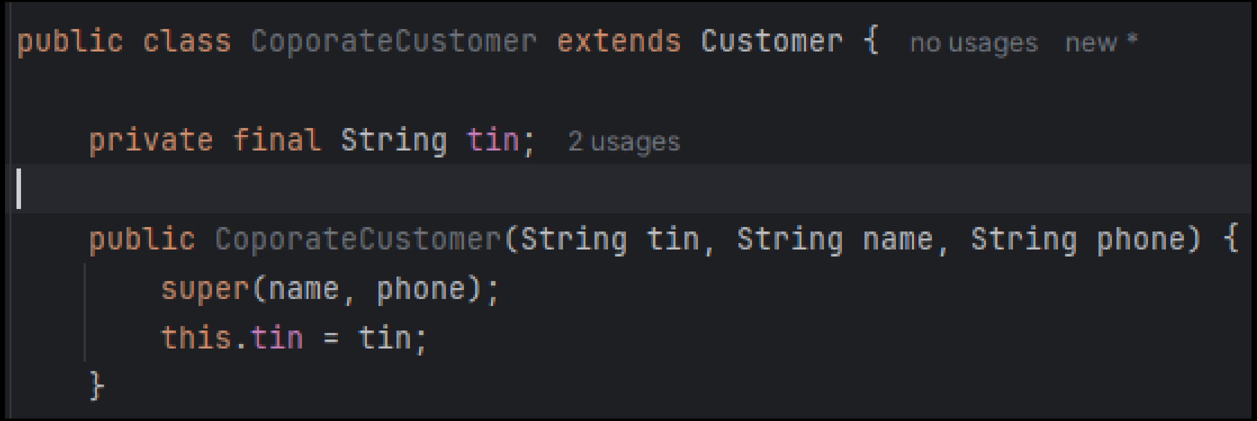
These principles are in the concept called “Dependency Injection”… more to come on that

INHERITANCE EXAMPLE



```
public class Customer {  1 usage  1 inheritor  new *
    private final String name;  2 usages
    private final String phone    ;  2 usages

    public Customer(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
```

The interface consists of all the functions for a Customer class.



```
public class CoporateCustomer extends Customer {  no usages  new *

    private final String tin;  2 usages

    public CoporateCustomer(String tin, String name, String phone) {
        super(name, phone);
        this.tin = tin;
    }
```

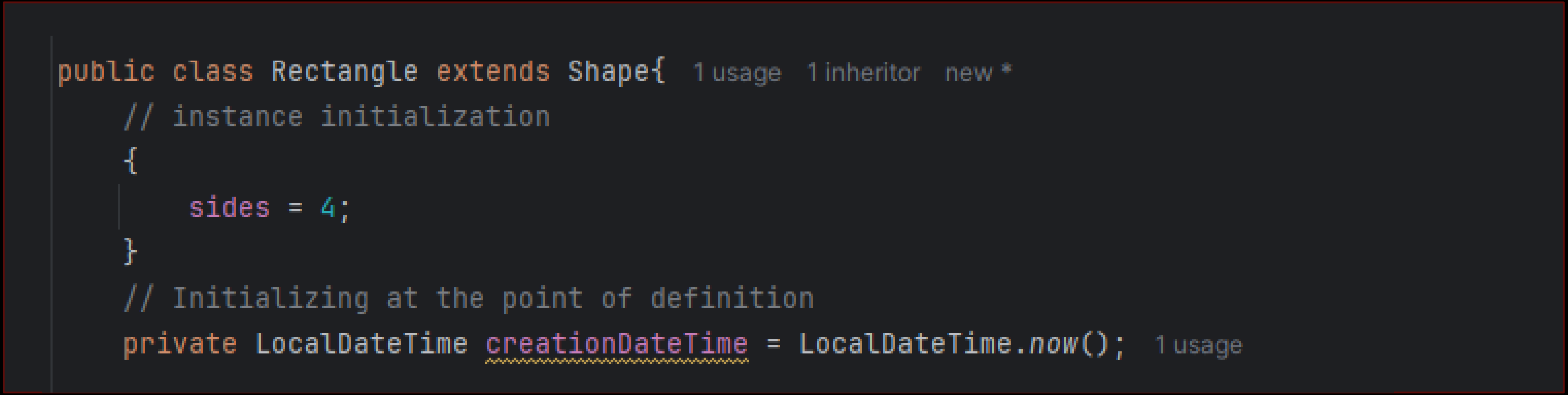
The `CoporateCustomer` Interface consists of all the functions for a Customer **AND** its functions for a total of four functions. The “Is a” relationship.

What if the `Customer` class declares a `getTaxRate`?

INITIALIZE REFERENCES

- When the objects are defined. The objects will always be initialized before the constructor is invoked.
- In the constructor for that class.
- Right before you use the object.
  - Called lazy initialization.
  - Reduces overhead when object creation is expensive and
  - the object doesn’t need to be created every time.
- Using instance initialization.

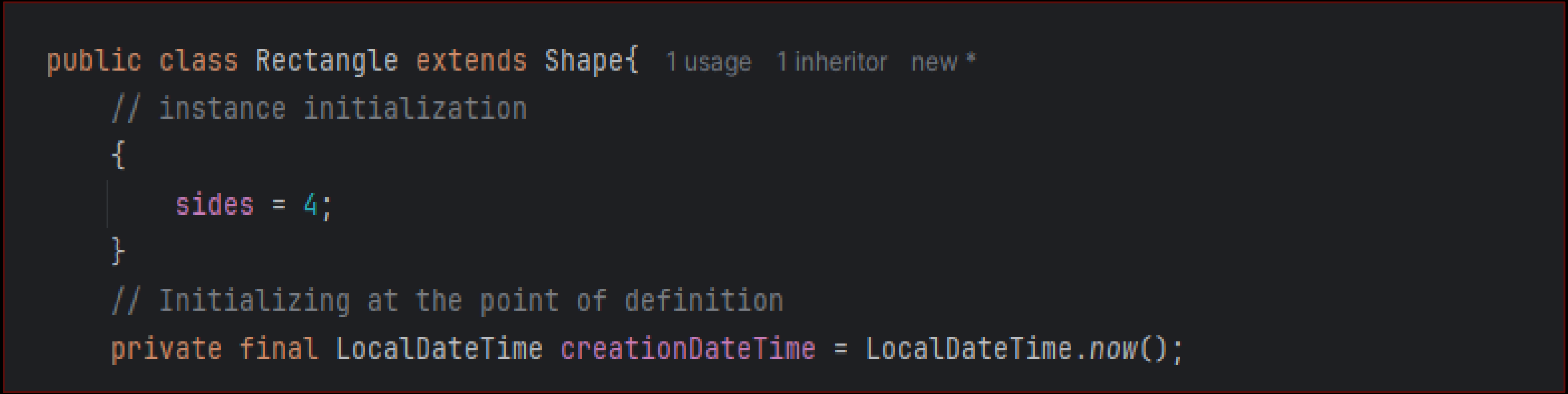
INSTANCE INIT & POINT OF DEFINITION



```
public class Rectangle extends Shape{  1 usage  1 inheritor  new *
    // instance initialization
    {
        sides = 4;
    }
    // Initializing at the point of definition
    private LocalDateTime creationDateTime = LocalDateTime.now();  1 usage
```

- When the objects are defined. The objects will always be initialized before the constructor is invoked.
- Before the constructor is called at the static class level

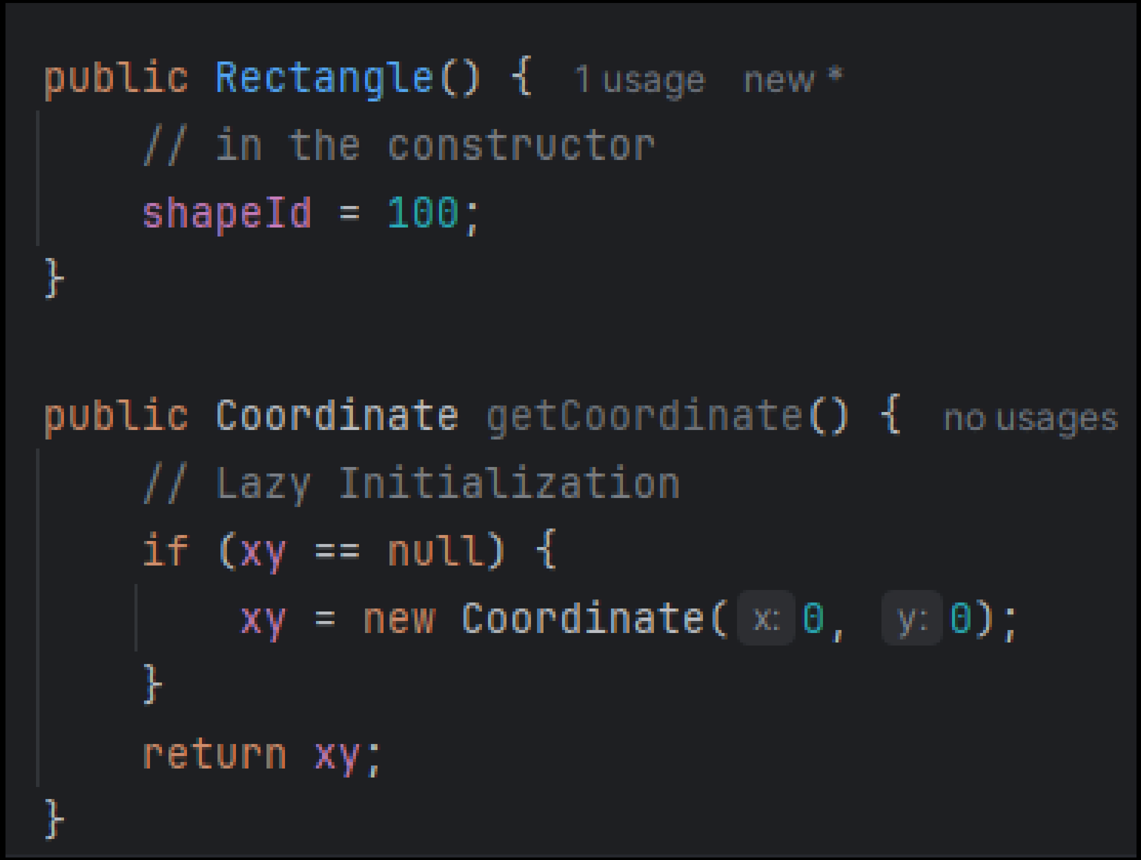
INSTANCE INIT & POINT OF DEFINITION



```
public class Rectangle extends Shape{  1 usage  1 inheritor  new *
    // instance initialization
    {
        sides = 4;
    }
    // Initializing at the point of definition
    private final LocalDateTime creationDateTime = LocalDateTime.now();
```

- When the objects are defined. The objects will always be initialized before the constructor is invoked.
- Before the constructor is called at the static class level

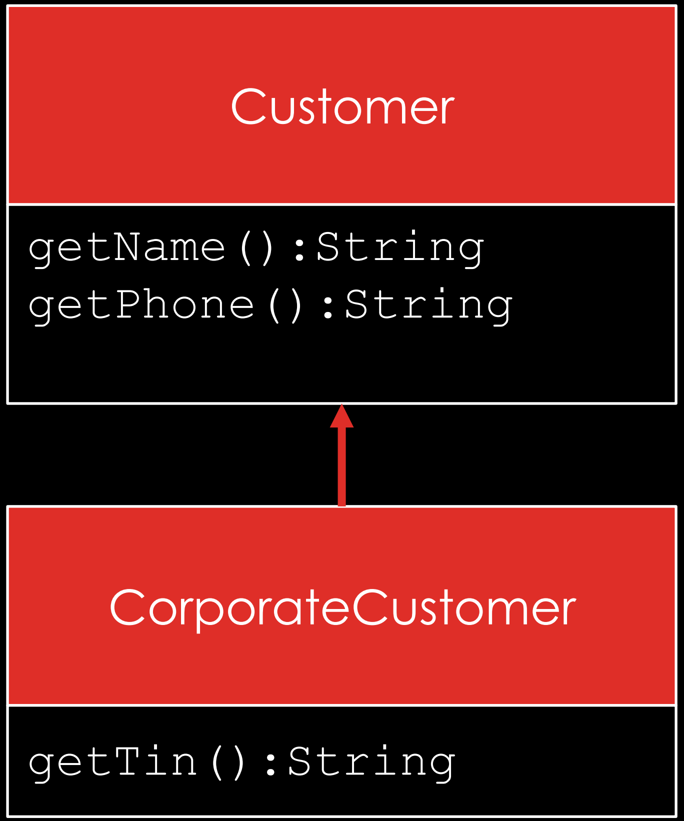
WHEN OBJECTS DEFINED



```
public Rectangle() {  1 usage  new *
    // in the constructor
    shapeId = 100;
}

public Coordinate getCoordinate() {  no usages
    // Lazy Initialization
    if (xy == null) {
        xy = new Coordinate( x: 0,  y: 0);
    }
    return xy;
}
```

- In the constructor for that class.
- Lazy initialization, when the class instance is needed and not sooner

WHAT IS INHERITANCE?



The interface consists of all the functions for a Customer class.

The `CoporateCustomer` Interface consists of all the functions for a Customer **AND** its functions for a total of four functions. The “Is a” relationship.

What if the `Customer` class declares a `getTaxRate`?

INHERITANCE FEATURES

Limitations:

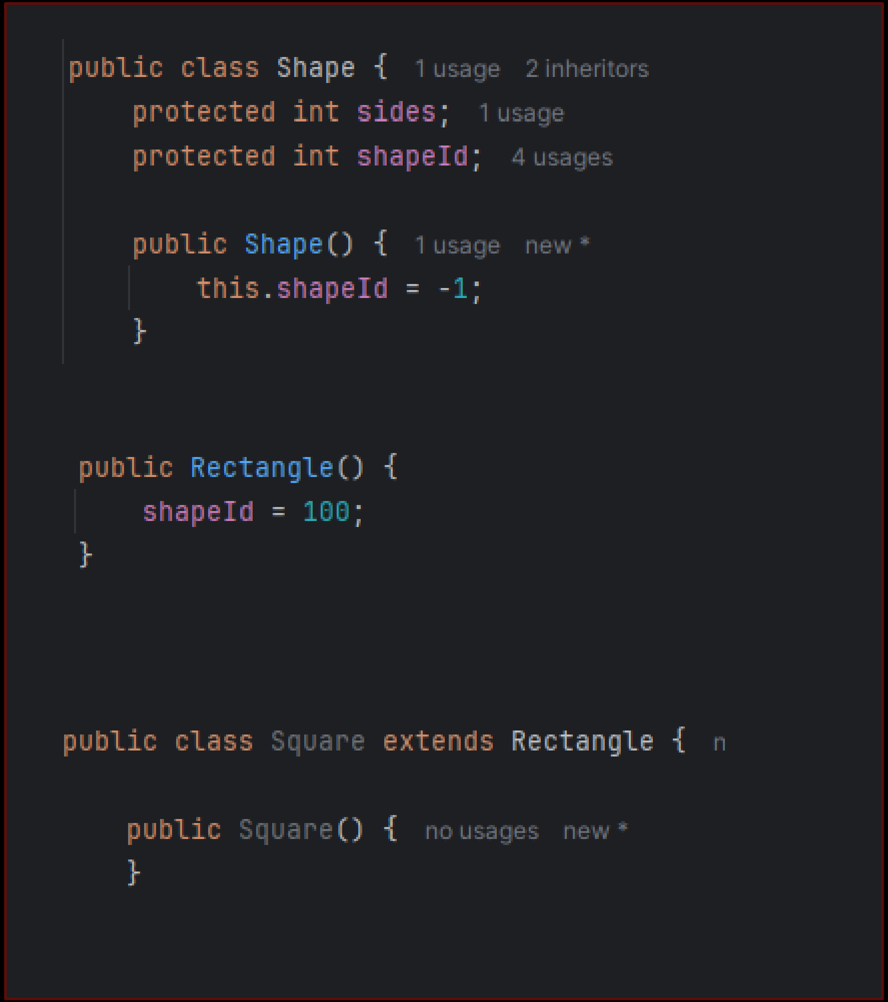
- Java does not support multiple inheritance with classes to avoid ambiguity (the "diamond problem").
- A class can only extend one superclass but can implement multiple interfaces.

Inheritance is a powerful feature in Java that enhances code organization, promotes reuse, and helps in building extensible systems.

INHERITANCE INITIALIZATION

- The intialization happens from the top to the bottom
- The base class[es] each initialize themselves
- The child class can pass it constructed values ”up” the chain with a special keyword called `super`.
- The keyword `super` must be the first line in the constructor
  - not counting comments

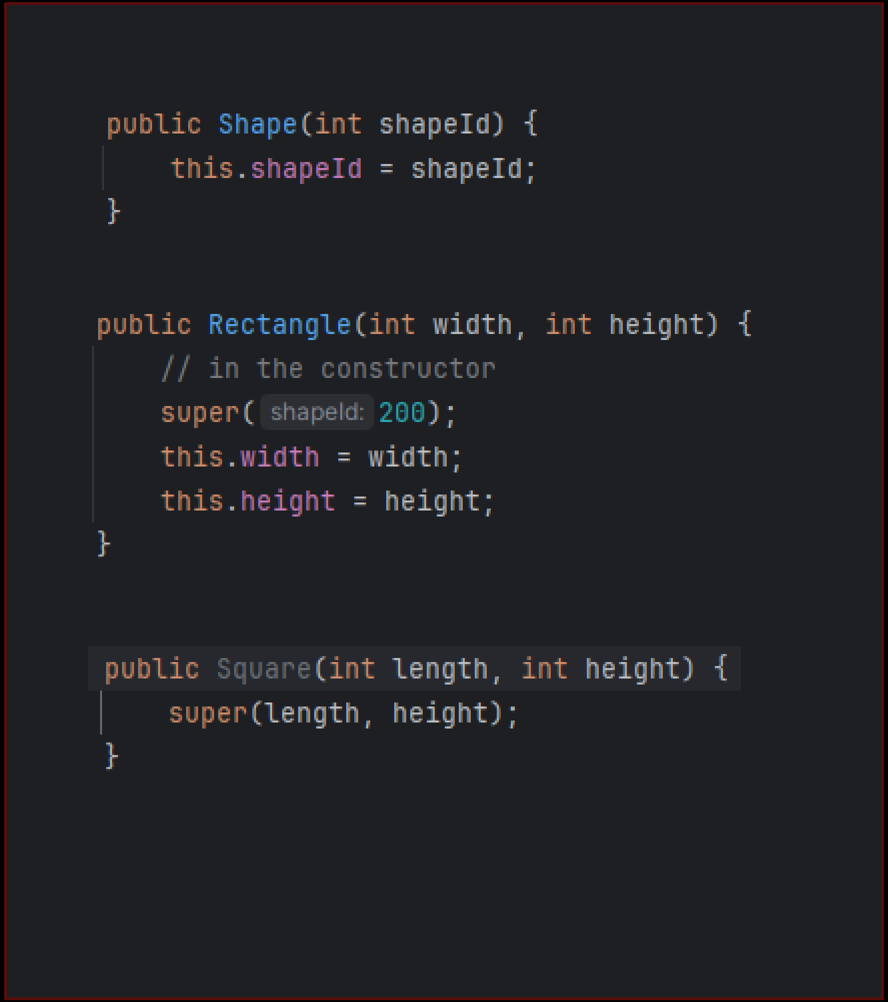
INHERITANCE INITIALIZATION EXAMPLE



```
public class Shape {  1 usage  2 inheritors
    protected int sides;  1 usage
    protected int shapeId;  4 usages

    public Shape() {  1 usage  new *
        this.shapeId = -1;
    }


 public Rectangle() {
     shapeId = 100;
 }



public class Square extends Rectangle {  n

    public Square() {  no usages  new *
    }
```

- When creating a `Square` instance the compiler will start with the highest base class (`Shape`) and initialize the values from the top to the bottom
  - shapeId is set to -1
  - shapeId is then reset to 100
  - `Square` does nothing so `shapeId` remains 100 and the `sides` is un-initialized.

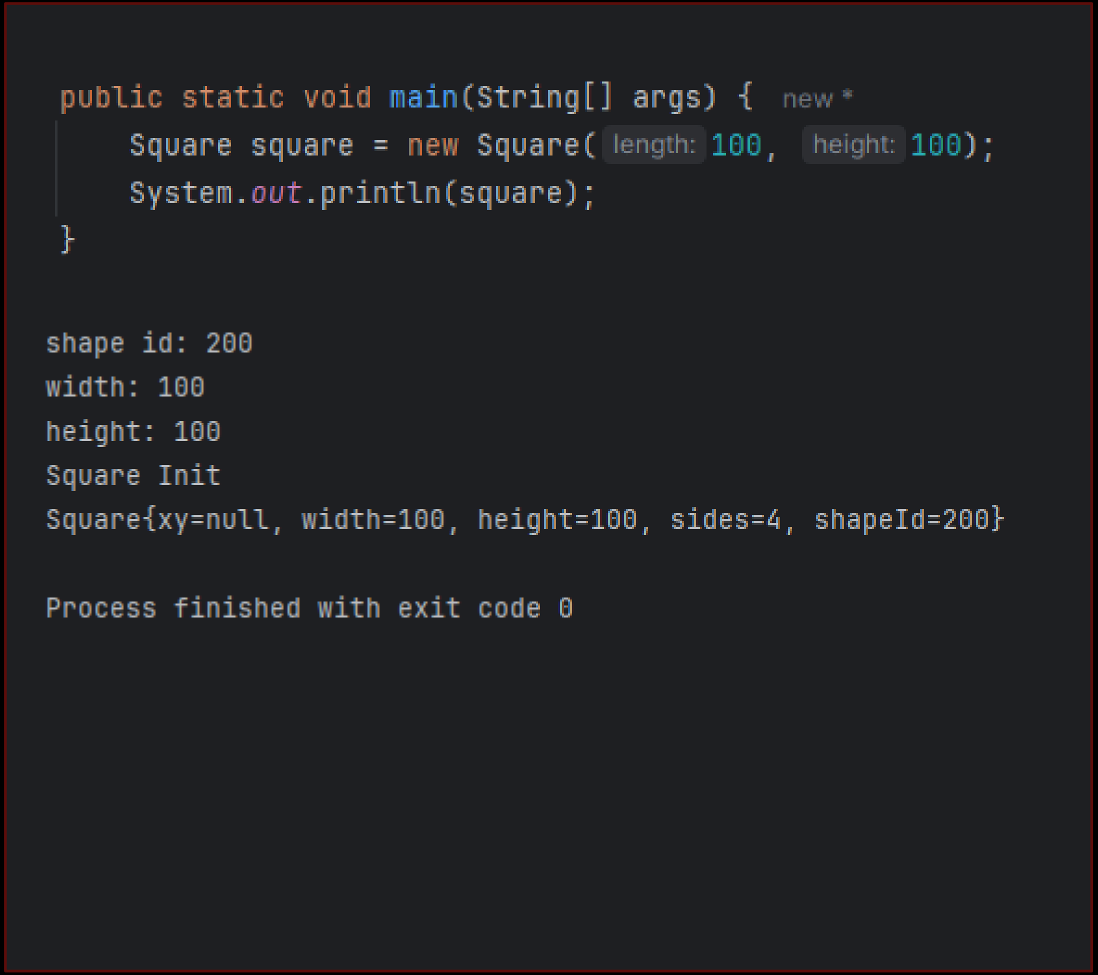
INHERITANCE INITIALIZATION EXAMPLE



```java
public Shape(int shapeId) {
    this.shapeId = shapeId;
}

public Rectangle(int width, int height) {
    // in the constructor
    super( shapeId: 200);
    this.width = width;
    this.height = height;
}

public Square(int length, int height) {
    super(length, height);
}
```

- When creating a `Square` instance given the length and height the compiler will start with the highest base class (`Shape`) and initialize the values from the top to the bottom
  - shapeId is set to 200, passed in from Rectangle's super
  - Width and Height are set from the values from Square
  - `Square` does nothing else after the super

INHERITANCE INITIALIZATION OUTPUT



```
public static void main(String[] args) {  new *
    Square square = new Square( length: 100,  height: 100);
    System.out.println(square);
}

shape id: 200
width: 100
height: 100
Square Init
Square{xy=null, width=100, height=100, sides=4, shapeId=200}

Process finished with exit code 0
```

- When creating a `Square` instance given the length and height the compiler will start with the highest base class (`Shape`) and initialize the values from the top to the bottom
  - shapeId is set to 200, passed in from Rectangle’s super
  - Width and Height are set from the values from Square
  - `Square` does nothing else after the super

DELEGATION (PRIVATE INHERITANCE)

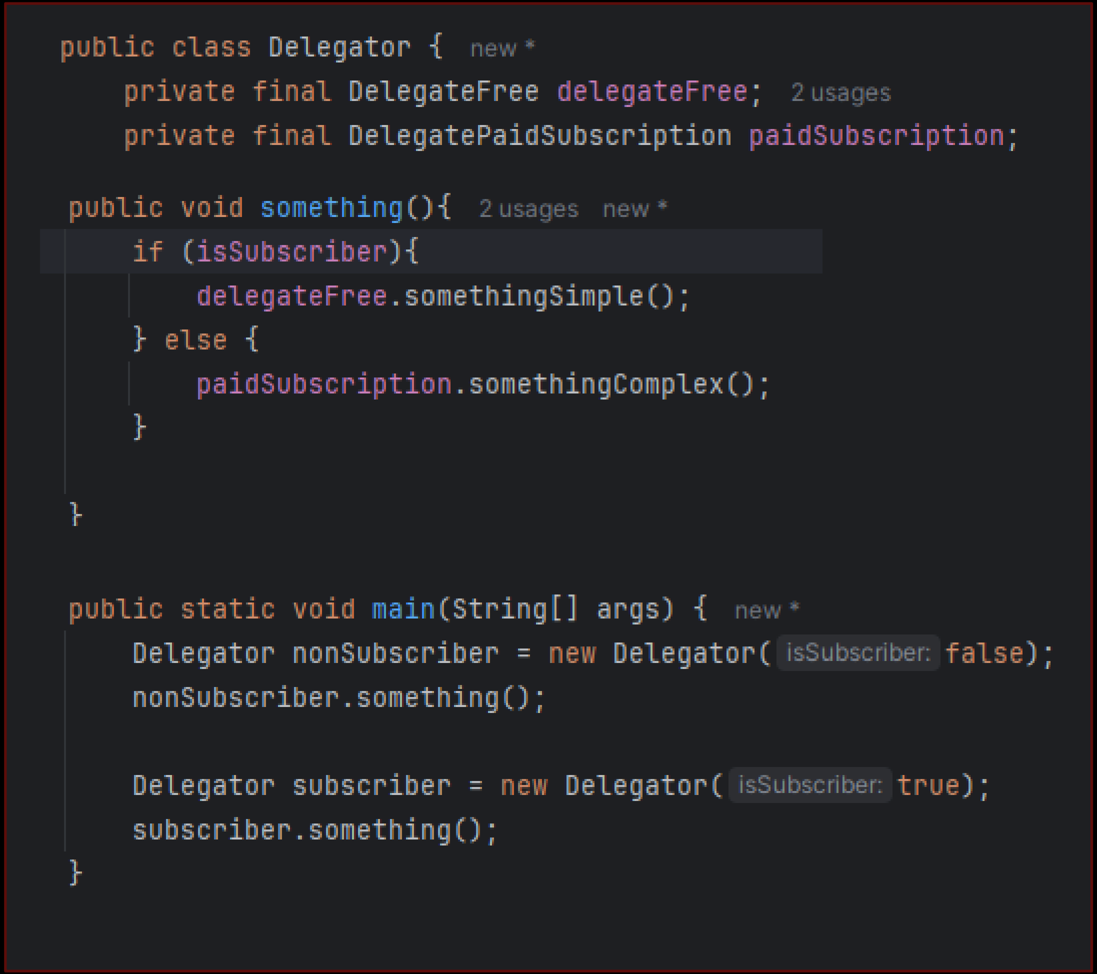
- delegation is a design technique where an object relies on another "helper" object to perform a specific task or behavior, instead of doing it itself.
- This involves an object "delegating" responsibility to another, which helps promote flexibility, reusability, and maintainability of code.
- Delegator Object: The object that has a task to perform but delegates it to another object.
- Delegate Object: The object to which the task is delegated; it performs the actual work.
- The delegator object holds a reference to the delegate and calls methods on it to perform specific operations.

DELEGATION (PRIVATE INHERITANCE)

Benefits

- Decoupling: Separates concerns, allowing objects to focus on a single responsibility.
- Flexibility: Change delegate objects at runtime, allowing more flexible code.
- Reusability: The delegate class can be reused in other parts of the application

DELEGATION EXAMPLE



```
public class Delegator {  new *
    private final DelegateFree delegateFree;  2 usages
    private final DelegatePaidSubscription paidSubscription;

 public void something(){  2 usages  new *
     if (isSubscriber){
         delegateFree.somethingSimple();
     } else {
         paidSubscription.somethingComplex();
     }

 }

 public static void main(String[] args) {  new *
     Delegator nonSubscriber = new Delegator( isSubscriber: false);
     nonSubscriber.something();

     Delegator subscriber = new Delegator( isSubscriber: true);
     subscriber.something();
 }
```

Delegator

- Decoupling: It is the class that defines how to implement the task. Client is unaware of the choice
- Flexibility: Choose a different implementation based on subscription.
- Reusability: The delegate class[es] can be reused in other parts of the application.

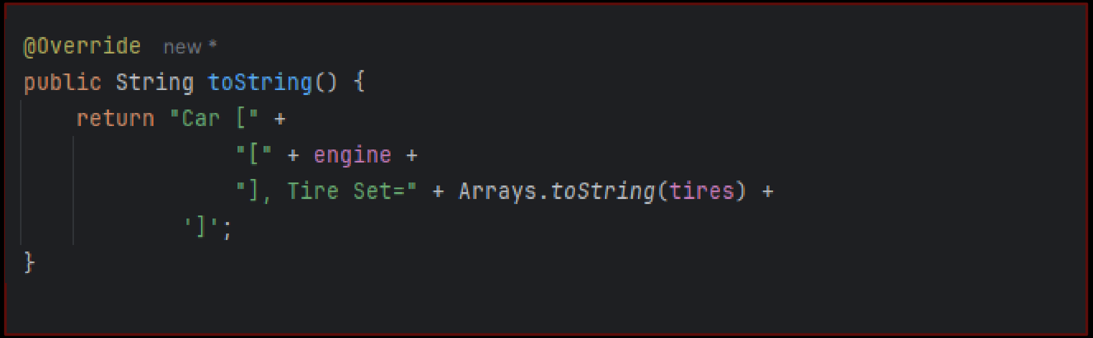
ANNOTATIONS

The `@Override` annotation indicates a method is intended to override a method in a superclass.

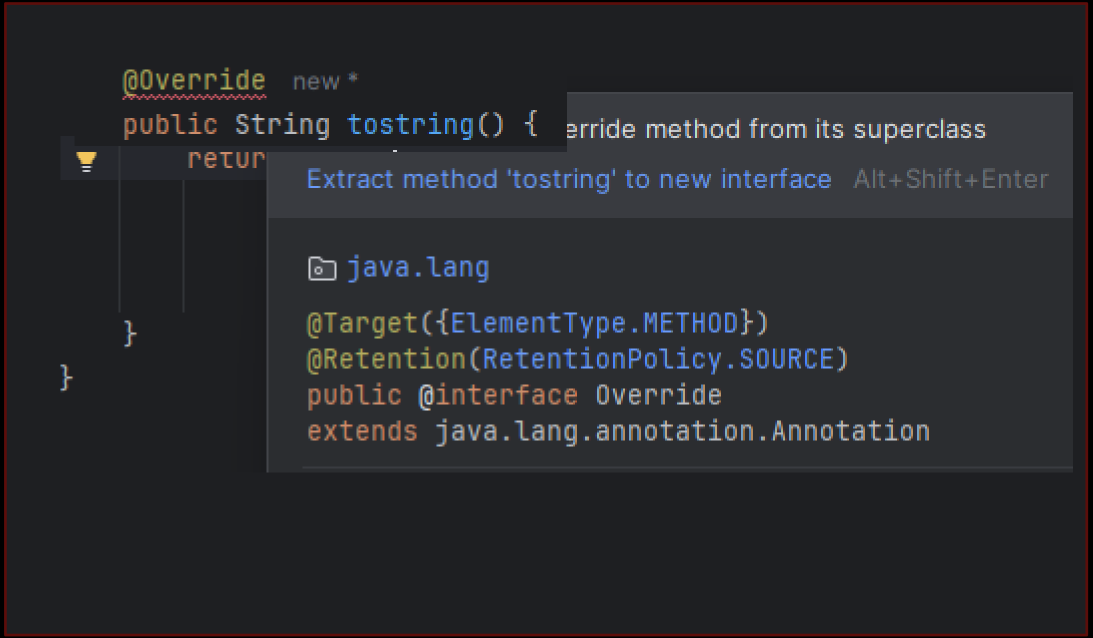
Serves as a form of documentation and enables the compiler to check if the method is indeed overriding a superclass method.

If the annotated method does not actually override any method in the superclass it throws an error/warning

ANNOTATION EXAMPLE



```
@Override  new *
public String toString() {
    return "Car [" +
                "[" + engine +
                "], Tire Set=" + Arrays.toString(tires) +
              ']';
}
```



```
@Override  new *
public String tostring() {
    retur
}
}
```

erride method from its superclass

Extract method 'tostring' to new interface Alt+Shift+Enter

```
java.lang
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface Override
extends java.lang.annotation.Annotation
```

The optional but highly recommend `@Override` annotation for Object's `toSting():String`

If there is a typo in the method name the compiler will display the red error message

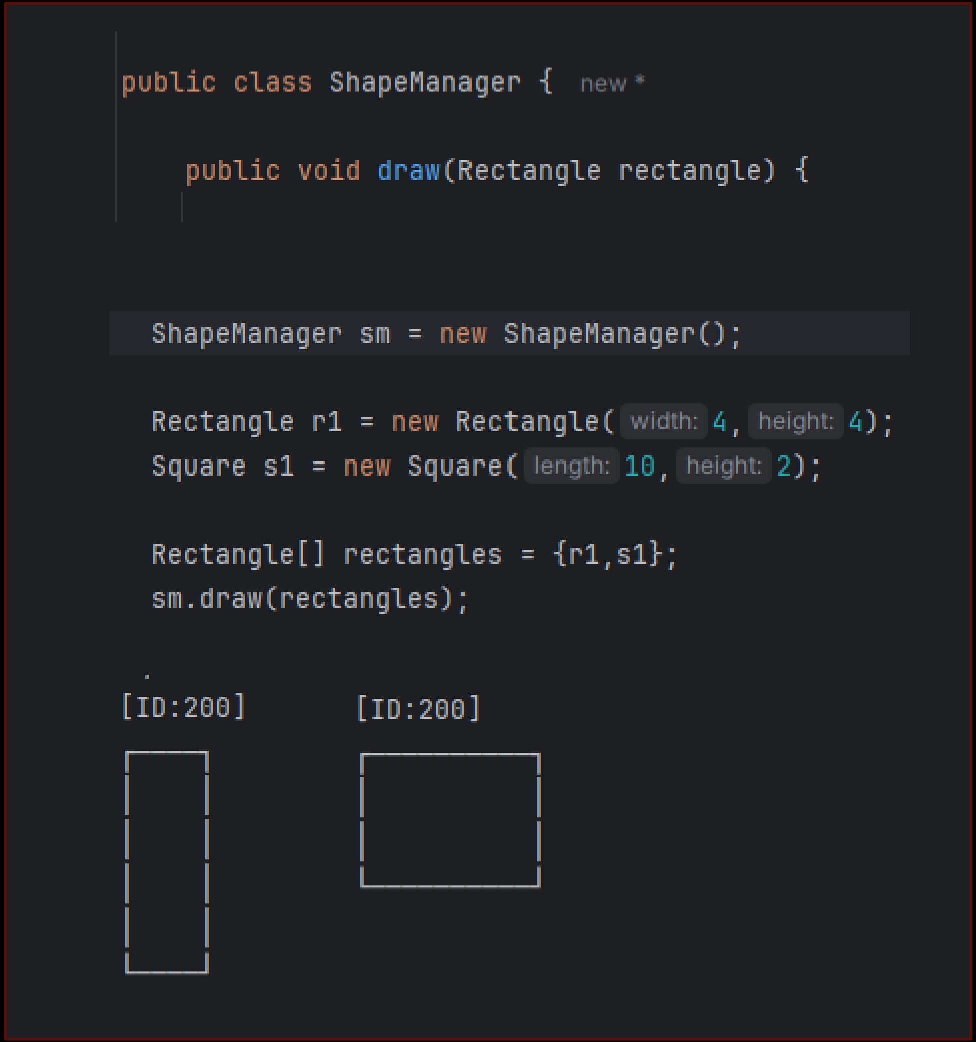
CHOOSING BETWEEN COMPOSITION AND INHERITANCE

- Always choose composition
- Caveat for framework authors

UPCASTING AND OOP

- upcasting refers to the process of converting a subclass type to a superclass type. The Java compiler is responsible for the behavior and the programmer doesn’t require to perform explicit casting.
- Upcasting allows a subclass object to be treated as if it were an instance of its superclass.
- Commonly used in polymorphism, where a superclass reference is used to refer to subclass objects.

UPCASTING EXAMPLE

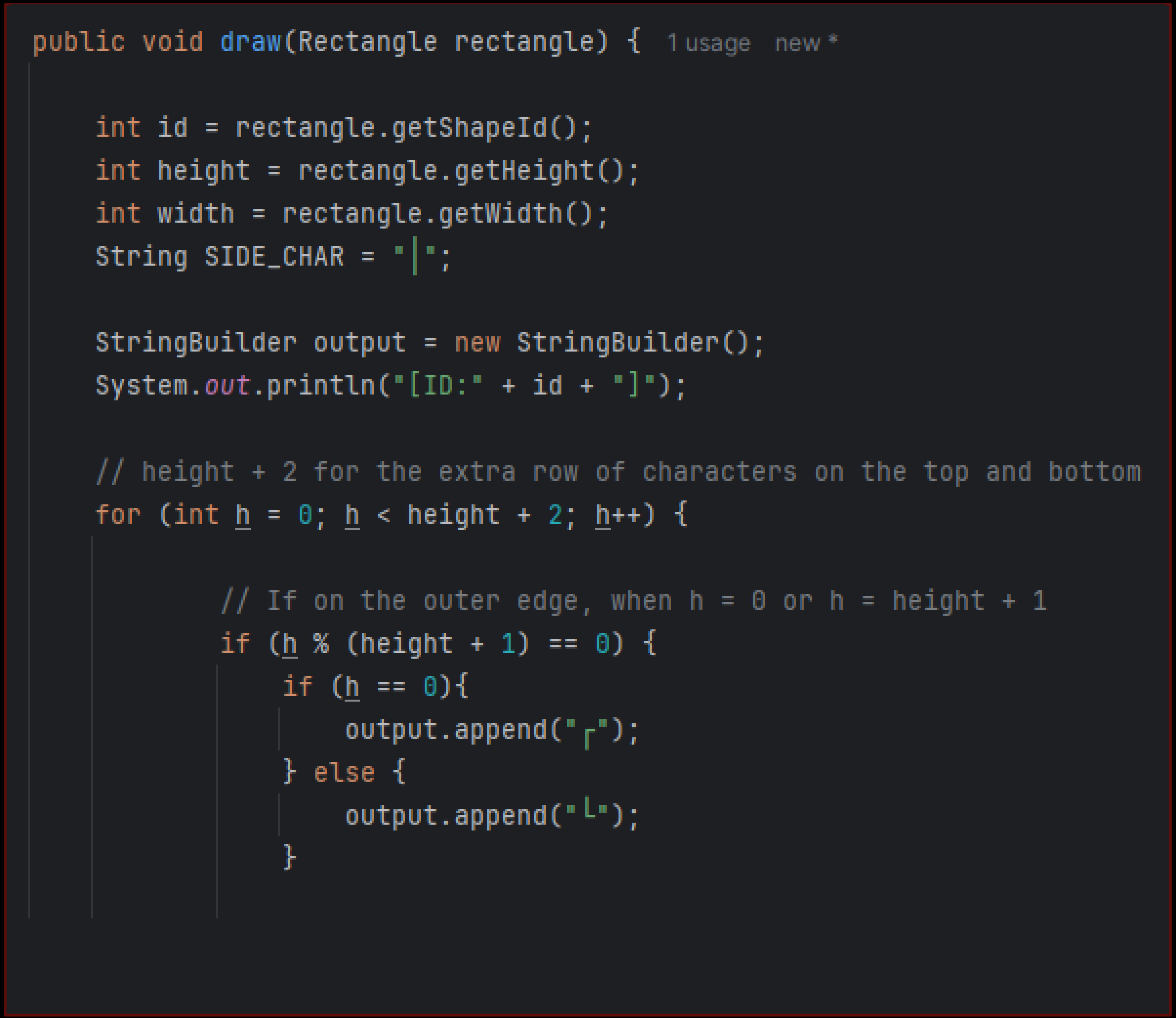


```
public class ShapeManager {  new *

    public void draw(Rectangle rectangle) {


  ShapeManager sm = new ShapeManager();

  Rectangle r1 = new Rectangle( width: 4, height: 4);
  Square s1 = new Square( length: 10, height: 2);

  Rectangle[] rectangles = {r1,s1};
  sm.draw(rectangles);
```

```
[ID:200]        [ID:200]
```

- The `ShapeManager` class has a `draw` method that takes in an array of `Rectangle` classes.
- We can create an array of Rectangles
  - One `Rectangle`
  - One `Square`
  - BUT NOT a `Shape`
- The `draw` method only needs what's in the `Rectangle` interface and a `Square` will suffice but a `Shape` will NOT!

UPCASTING EXAMPLE



```
public void draw(Rectangle rectangle) {  1 usage  new *

    int id = rectangle.getShapeId();
    int height = rectangle.getHeight();
    int width = rectangle.getWidth();
    String SIDE_CHAR = "│";

    StringBuilder output = new StringBuilder();
    System.out.println("[ID:" + id + "]");

    // height + 2 for the extra row of characters on the top and bottom
    for (int h = 0; h < height + 2; h++) {

            // If on the outer edge, when h = 0 or h = height + 1
            if (h % (height + 1) == 0) {
                if (h == 0){
                    output.append("┌");
                } else {
                    output.append("└");
                }
```

- The `draw` method only needs `getShapeId()`, `getWidth`() and `getHeight`().
- All these methods are in the `Rectangle` interface
- As a `Square` "is-a" `Rectangle` then `Square` and any sub-type of `Rectangle` will do!

TYPES OF FINAL FIELDS

final field is a variable declared with the final keyword, which makes it a constant after it's initialized. Once a final field is assigned a value, that value cannot be changed, commonly used to create constants and ensure immutability.

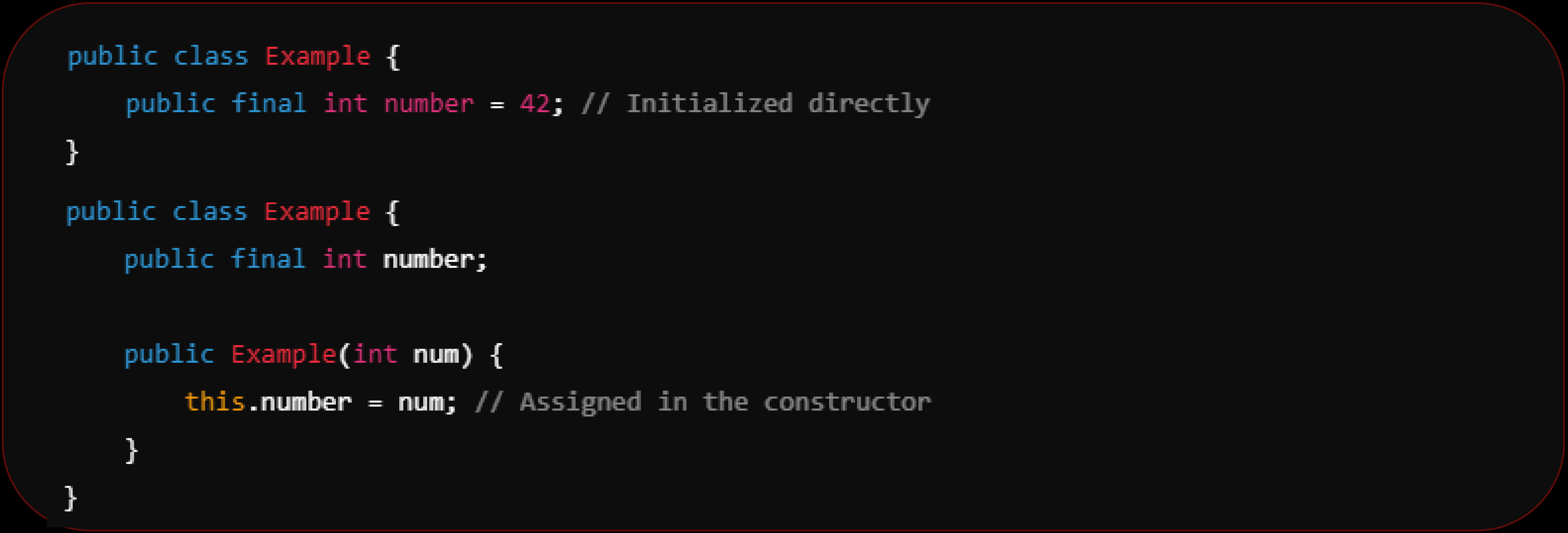
Types of Final Fields:

Instance Final Fields: Declared within a class, but outside any method. They can be assigned only once, either directly when declared or within the constructor of the class.

Static Final Fields: Declared with both static and final keywords. These fields are constants at the class level, shared across all instances of the class, and are usually initialized when they're declared.

FINAL EXAMPLES

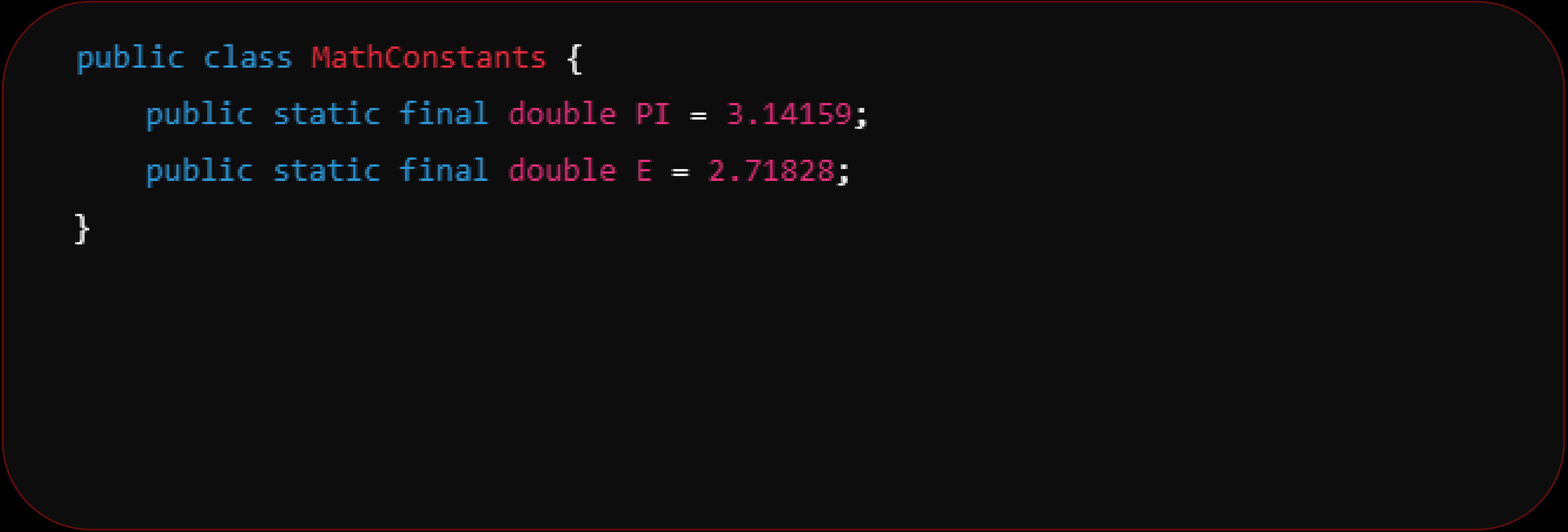
Instance Final Fields: Declared within a class, but outside any method. They can be assigned only once, either directly when declared or within the constructor of the class.



```java
public class Example {
    public final int number = 42; // Initialized directly
}

public class Example {
    public final int number;

    public Example(int num) {
        this.number = num; // Assigned in the constructor
    }
}
```

FINAL EXAMPLES

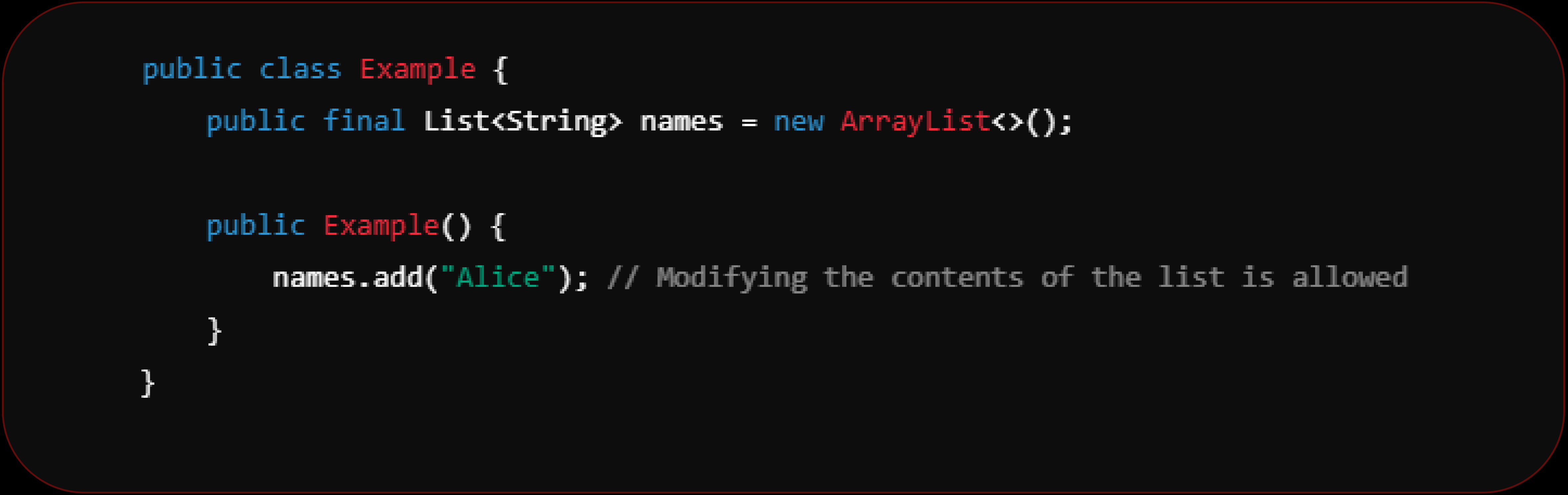
Static Final Fields: Declared with both static and final keywords. These fields are constants at the class level, shared across all instances of the class, and are usually initialized when they're declared.



```
public class MathConstants {
    public static final double PI = 3.14159;
    public static final double E = 2.71828;
}
```

IMMUTABLE REFERENCE, NOT IMMUTABLE OBJECT

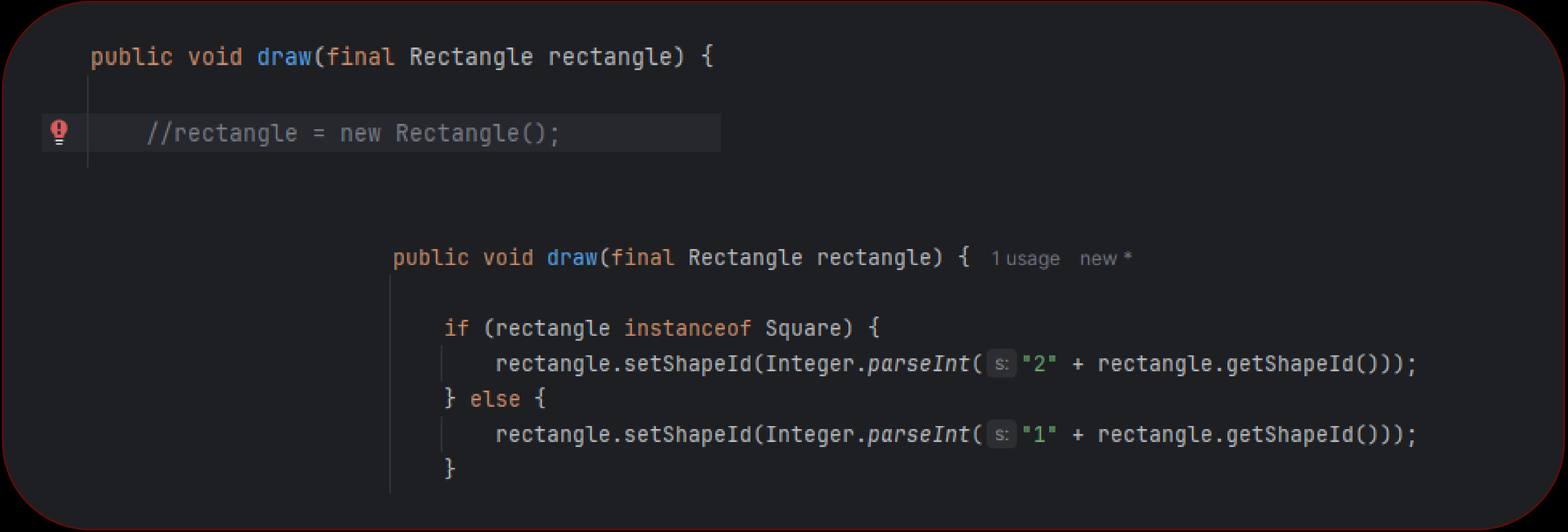
If a final field is an object reference (e.g., final List<String> names), the reference itself cannot change, but the object's contents can still be modified (e.g., adding or removing items from the list).



```java
public class Example {
    public final List<String> names = new ArrayList<>();

    public Example() {
        names.add("Alice"); // Modifying the contents of the list is allowed
    }
}
```

FINAL METHOD ARGUMENTS

If a final method argument means the method will not change the reference to the object, but the contents and fields may change.



```
public void draw(final Rectangle rectangle) {

    //rectangle = new Rectangle();
```

```
public void draw(final Rectangle rectangle) {  1 usage  new *

    if (rectangle instanceof Square) {
        rectangle.setShapeId(Integer.parseInt( s: "2" + rectangle.getShapeId()));
    } else {
        rectangle.setShapeId(Integer.parseInt( s: "1" + rectangle.getShapeId()));
    }
```

FINAL METHODS

final method is a method declared with the final keyword. It cannot be overridden by subclasses. Declaring a method as final is useful when you want to prevent subclasses from modifying or changing the behavior of that method.

Purpose of Final Methods:

Prevent Modification: Final methods ensure that the method’s functionality remains unchanged in any subclasses, which can be useful when you want a specific behavior to be preserved.

Improve Security: By making certain methods final, you protect critical code from being accidentally or maliciously altered in subclasses, which can be particularly important in sensitive or foundational classes.

FINAL METHODS

final method is a method declared with the final keyword. It cannot be overridden by subclasses. Declaring a method as final is useful when you want to prevent subclasses from modifying or changing the behavior of that method.

Purpose of Final Methods:

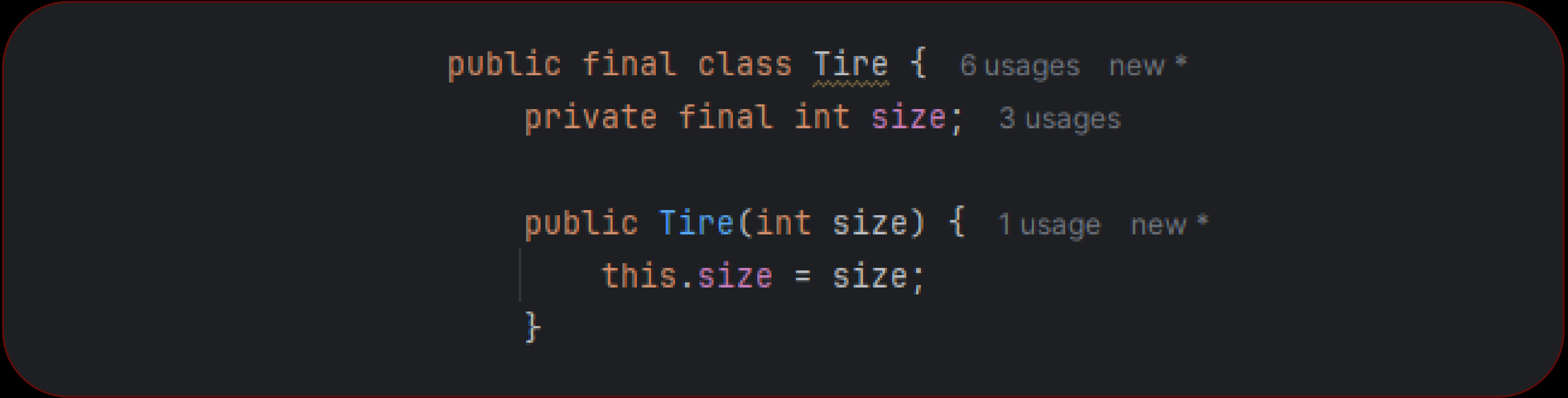
Prevent Modification: Final methods ensure that the method’s functionality remains unchanged in any subclasses, which can be useful when you want a specific behavior to be preserved.

Improve Security: By making certain methods final, you protect critical code from being accidentally or maliciously altered in subclasses, which can be particularly important in sensitive or foundational classes.

FINAL CLASSES

final classes are declared with the final keyword. They cannot be subclassed. Declare a class final when you want to prevent subclasses.

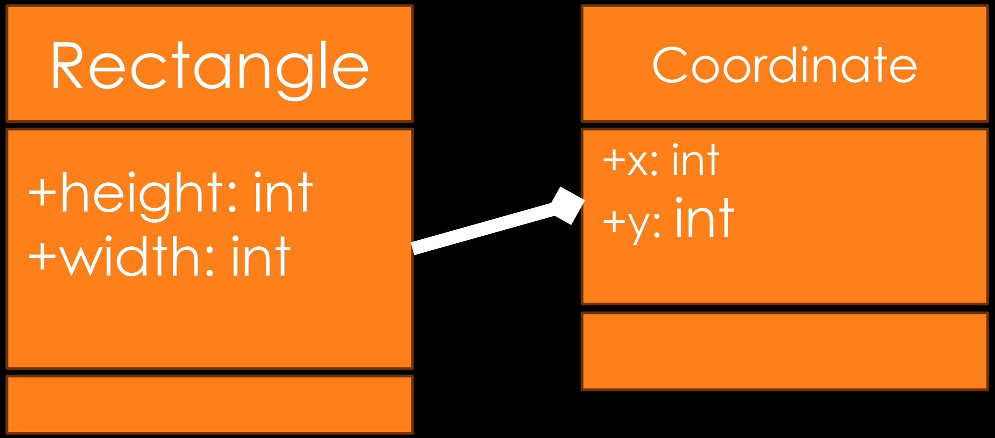
Improve Security: By making the class final, you protect critical classes (like `String`) from being altered in subclasses, which can be particularly important in sensitive and foundational classes.



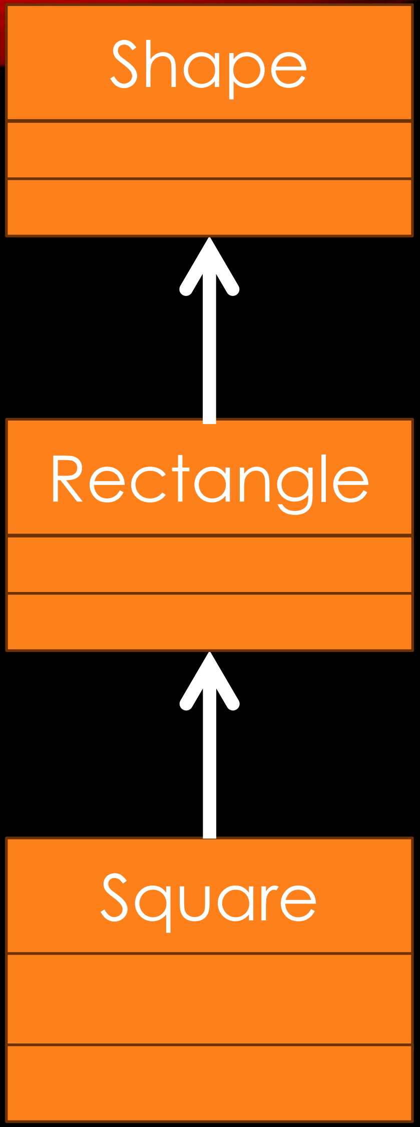
```
public final class Tire {  6 usages  new *
    private final int size;  3 usages

    public Tire(int size) {  1 usage  new *
        this.size = size;
    }
```

TOPIC SUMMARY - COMPOSITION

Composition is a critical design principle used to build complex classes by combining simpler ones. It allows one class to contain references to objects of other classes, enabling the creation of more complex data structures. This relationship signifies a "has-a" relationship, where one class is composed of one or more instances of other classes.



e.g. a Rectangle has a Coordinate

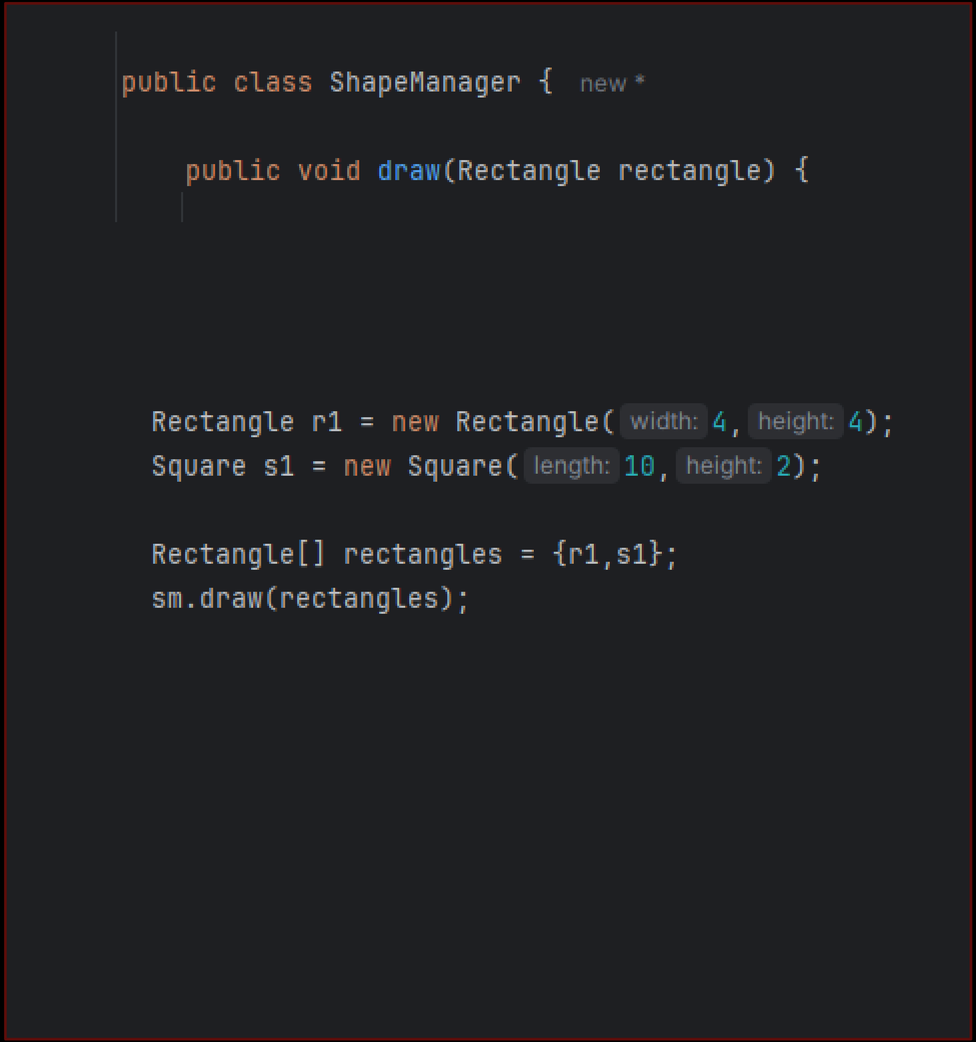


TOPIC SUMMARY - INHERITANCE

Inheritance: a fundamental object-oriented programming (OOP) principle allowing a new class (a subclass) to inherit properties and behaviors (methods) from an existing class (called a superclass or base class).

Referred to as an "is-a" relationship, where the subclass is a more specific version of the superclass. E.g. A square “is-a” rectangle, which in turn “is-a” Shape

TOPIC SUMMARY - UPCASTING



```
public class ShapeManager {  new *

    public void draw(Rectangle rectangle) {




  Rectangle r1 = new Rectangle( width: 4, height: 4);
  Square s1 = new Square( length: 10, height: 2);

  Rectangle[] rectangles = {r1,s1};
  sm.draw(rectangles);
```

- The `ShapeManager` class has a `draw` method that takes in an array of `Rectangle` classes, which is a base class
- We can create an array of Rectangles
  - One `Rectangle`
  - One `Square`
- The `draw` method only needs what's in the `Rectangle` interface and a `Square` will suffice because a `Square` is a `Rectangle`.

FINAL FIELDS, METHODS, ARGUMENTS, CLASSES

final field is a variable declared with the final keyword, which makes it a constant after it's initialized. Once a final field is assigned a value, that value cannot be changed, commonly used to create constants and ensure immutability.

If a final method argument means the method will not change the reference to the object, but the contents and fields may change.

final Methods prevent Modification. Final methods ensure that the method's functionality remains unchanged in any subclasses

final classes are declared with the final keyword. They cannot be subclassed. Declaring a class as final is useful when you want to prevent subclasses.

ONLINE VIDEOS

- Learning Programming In Java
  - YouTube Tutorials
- Lab Exercises
  - http://www.learnprogramminginjava.com/