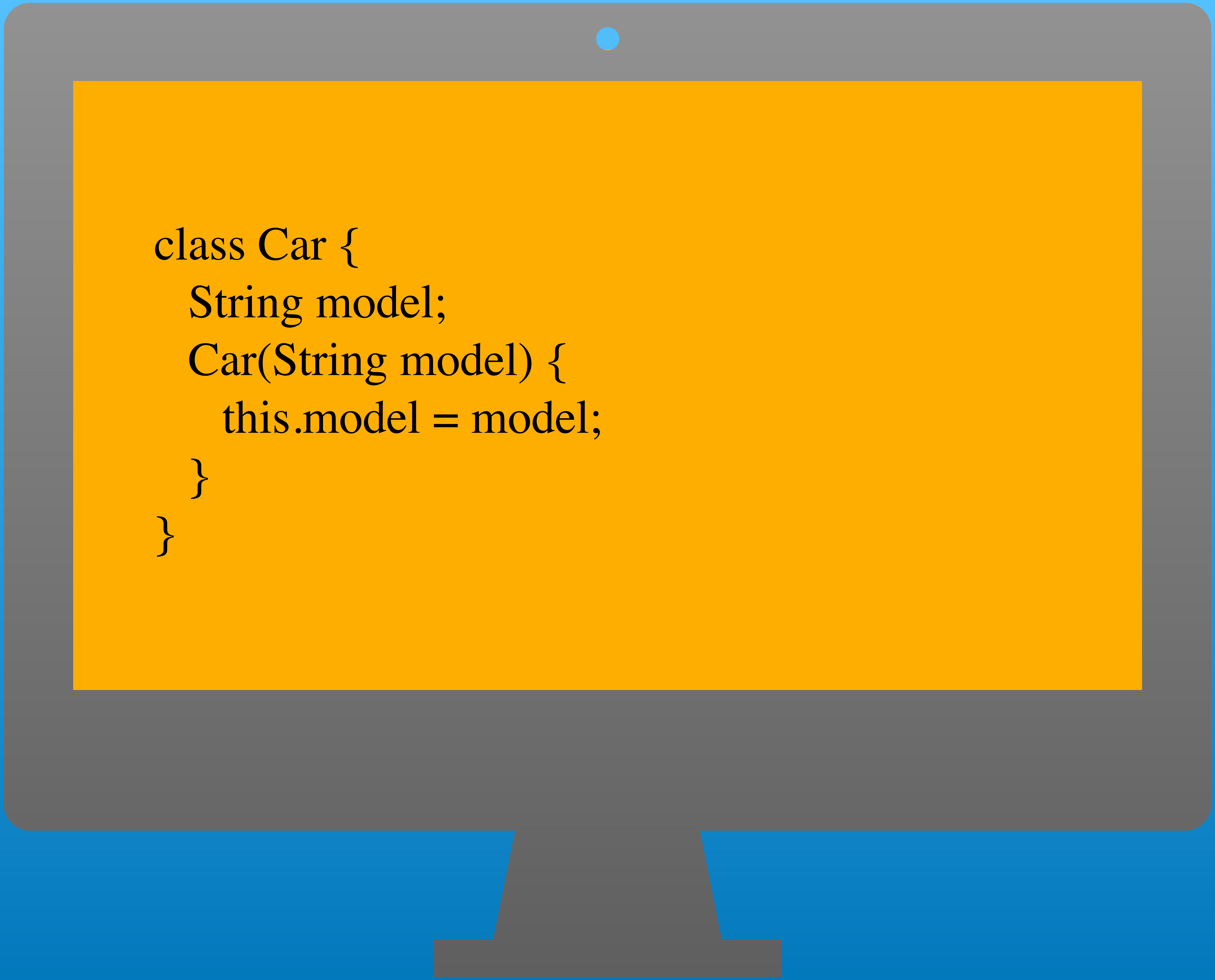
Constructors, Garbage Collection, Object Access Modifiers, and More

Constructors & Garbage Collection, Object Access Modifiers, Method Overloading, `this` keyword, Static (Variable, Method, Block), Final Keyword, and Wrapper Classes,

What is a Constructor?

Constructor initializes objects.
Syntax: Constructor name must match the class name.
Types: Default and Parameterized constructors.



```
class Car {
  String model;
  Car(String model) {
    this.model = model;
  }
}
```

Garbage Collection in Java

Garbage Collection in Java

- Java automatically manages memory using the garbage collector.
- Objects that are no longer referenced are eligible for garbage collection.
- Manual invocation: `System.gc()`
- Important: No guarantee that `gc()` will run immediately.

Introduction to Access Modifiers

Introduction to Access Modifiers

🟢 **public:** Accessible everywhere.

Introduction to Access Modifiers

🟢 **public:** Accessible everywhere.

🔴 **private:** Accessible only within the class.

Introduction to Access Modifiers

🟢 **public:** Accessible everywhere.

🔴 **private:** Accessible only within the class.

🟡 **protected:** Accessible within the package and subclasses.

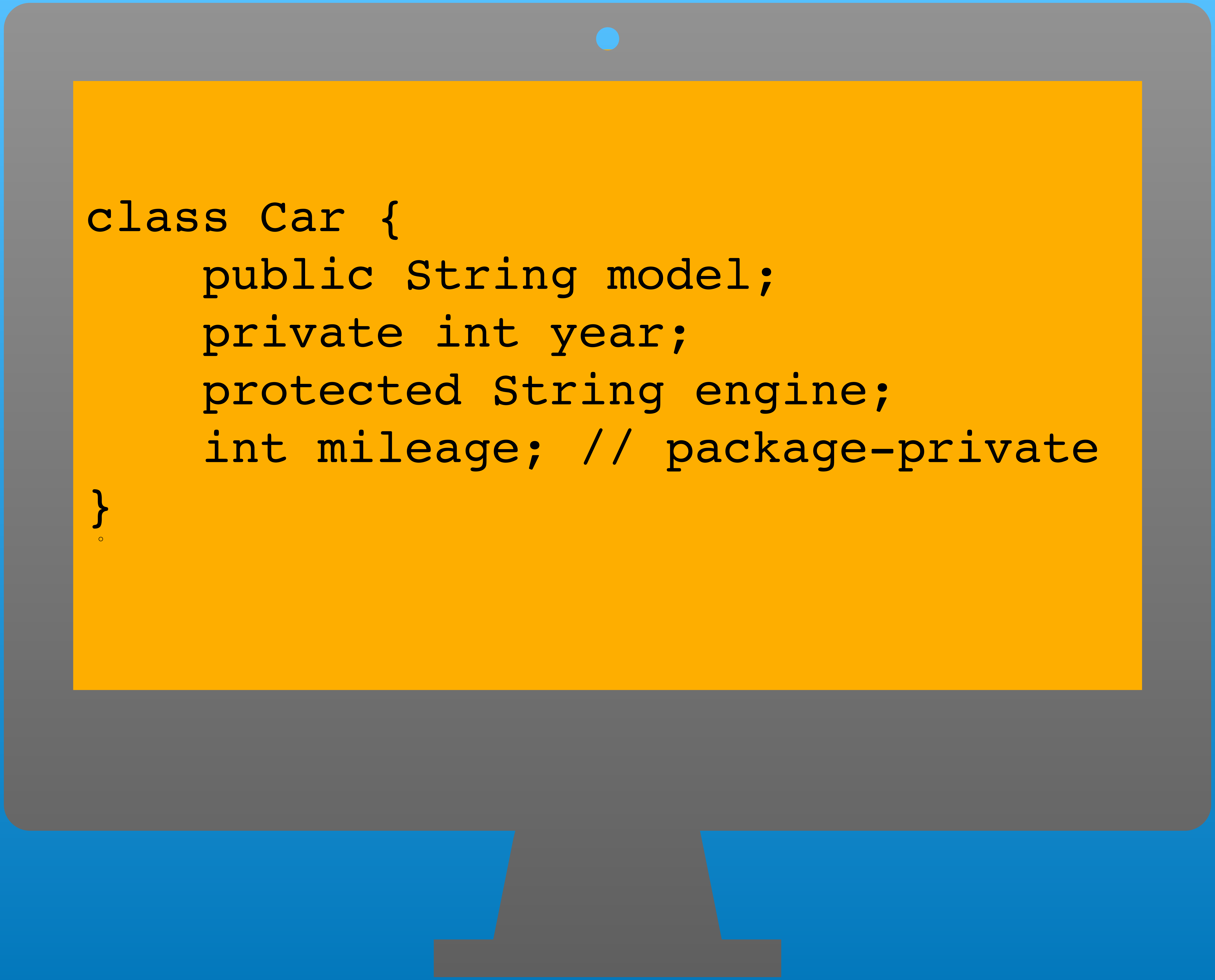
Introduction to Access Modifiers

🟢 **public:** Accessible everywhere.

🔴 **private:** Accessible only within the class.

🟡 **protected:** Accessible within the package and subclasses.

🔵 Default (no modifier): Package-private.



```
class Car {
    public String model;
    private int year;
    protected String engine;
    int mileage; // package-private
}
```

What is Method Overloading?

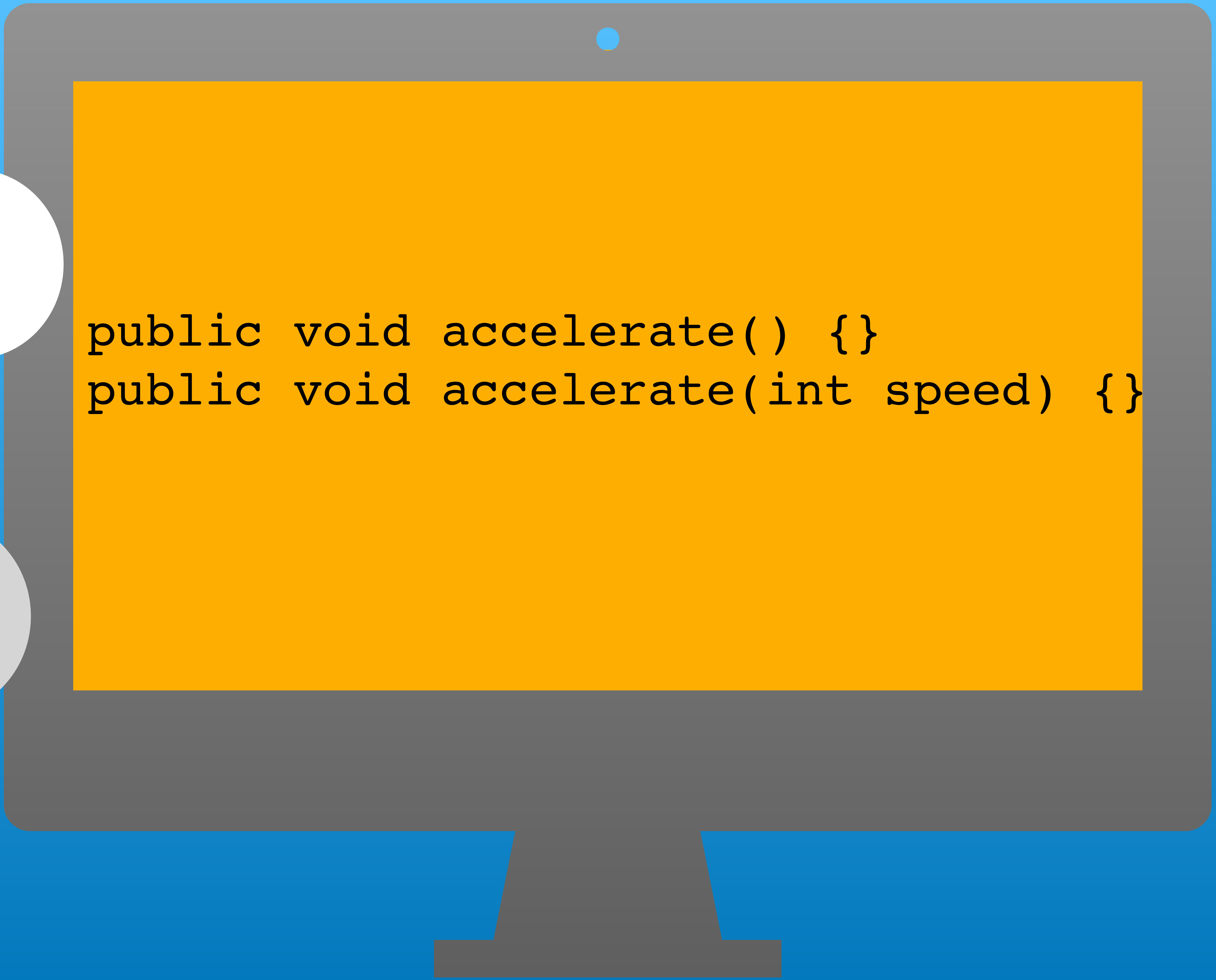
What is Method Overloading?

Multiple methods with the same name but different parameters.

Increases code readability and flexibility.

What is Method Overloading?

Multiple methods with the same name but different parameters. Increases code readability and flexibility.



```
public void accelerate() {}
public void accelerate(int speed) {}
```

Rules for Method Overloading

```
class Addition {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

Must differ in parameter type, number, or both.
Can’t differ only by return type.

Understanding `this` Keyword

```
class Car {
    String model;
    Car(String model) {
        this.model = model;
    }
}
```

Refers to the current instance of the class.
Used to eliminate confusion between class attributes and parameters.

`this` to Call Constructors

`this` to Call Constructors

One constructor can call another using `this()`.

```
class Car {
    String model;
    int year;
    Car(String model) {
        this(model, 2020); // Calling another constructor
    }
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

Static Variables

```
// Shared by all objects of a class.

class Car {
    static int carCount = 0;
    Car() {
        carCount++;
    }
}
```

Static Methods and Static Block

//Static methods can be called without creating an object.
Static block: Executed when the class is loaded.

```
static void showCarCount() {
    System.out.println("Car count: " + carCount);
}


static {
    System.out.println("Static block called");
}
```

Static Methods and Static Block

//Static methods can be called without creating an object.
Static block: Executed when the class is loaded.

```
static void showCarCount() {
    System.out.println("Car count: " + carCount);
}

static {
    System.out.println("Static block called");
}
```

Final Keyword Basics

final variable: Can’t be modified once assigned.
final method: Can’t be overridden.
final class: Can’t be subclassed.

```
final int MAX_SPEED = 120;
final void displaySpeed() {
    System.out.println("Speed: " + MAX_SPEED);
}
```

Introduction to Wrapper Classes

Wrapper classes convert primitive data types into objects.
Examples: `Integer`, `Double`, `Boolean`.
Autoboxing and Unboxing: Automatic conversion between primitives and wrappers.

```
Integer i = 10;  // Autoboxing
int j = i;       // Unboxing
```

Why Use Wrapper Classes?

- Useful in Collections (e.g., `ArrayList<Integer>`)
- Offers utility methods (e.g., `Integer.parseInt()`)

Best Practices for Class Design

- Use meaningful variable and method names.
- Keep methods small and focused.
- Use `this` for clarity when working with class variables.
- Leverage method overloading for flexible and readable code.
- Use access modifiers to encapsulate data.

Thank You