

Table of Contents

1. Best Practises in OOP For Software Desing	3
Object Oriented Programming	3
Inheritance	3
Interfaces	3
Abstract	4
Virtual methods	4
Storage structure	4
Generics.....	5
2. How my application adheres to these best practice.....	6
Inheritance	6
Interfaces	7
Abstract Classes.....	8
Virtual methods	9
Storage structure	9
ArrayList	9
3. Class attributes and methods and relationships with other classes	10
Sign.....	10
Opening Hours.....	12
Fare Display.....	12
Full Sign	13
Fare interface	14
Hourly payment	14
Weekly payment	14
Monthly payment	15
Barrier	15
Sensor.....	17
ID Reader	18
Vehicle	22
Vehicle type	22
Car park	23
4. Static Memory Allocation and Dynamic Memory Allocation	25
Static Memory Allocation	25
Dynamic Memory Allocation.....	25
Comparison table	26
5. Best data structures approach for the Car Park	27
Hash map.....	27
Array List	27
Other Data Structures Considered but not use:	28
6. Unit Testing.....	29
Vehicle type Enum	29
Payment classes.....	29
Barrier	29
Sensor.....	29

Sign classes.....	29
Vehicle	29
ID Readers.....	29
7. Screenshots of working application	30
User Guide.....	34
Launch the application:.....	34
Choices on the main menu:.....	34
When entering:	34
While Leaving:	34
General:	34
1. Bibliography	35

YouTube video link

<https://youtu.be/JYIAmc9QhXk>

1. Best Practises in OOP For Software Desing

Object Oriented Programming

Object-Oriented Programming continues to be an essential component of current software development. Its key attributes allow for modular, easily maintained, and scalable system development. This overview summarises the published research on recommendations for designing and implementing OOP features in Java, specifically the use of inheritance, interfaces, abstract classes, polymorphism, virtual methods, storage structures, and generics.

Inheritance

Inheritance promotes the reuse of code and classification hierarchy through the ability of subclasses or child classes to inherit attributes and behaviour from their superclass the parent classes. A famous example is a basic class named animal, that specifies generic behaviours like make a sound. A subclass, such as dog, may inherit from animal and overwrite its sound behaviour to be bark. This design encourages reuse of code while maintaining a coherent class structure. However, research has highlighted the potential risks of extensive or highly rigid inheritance hierarchies, that may lead to instability and tight coupling among parent and child classes [1]. To prevent this, software engineers should favour shallow hierarchy and differentiate among the right implementation, good use of inheritance and interface inheritance, with the former used only when strict behaviour sharing is necessary [2].

Composition is frequently suggested as a better alternative to inheritance, particularly when designed for flexibility or extensibility [3]. Inheritance should only be used when there is a clear is a connection and child class's truly must extend the parent class's behaviour rather than simply reusing its structure, according to best practices.

Interfaces

Interfaces are contracts which may be implemented through multiple classes, enabling decoupling and polymorphism. They are essential for attaining design abstraction and improving reliability.[4], interfaces are particularly helpful in big structures since they enable components to vary independently, facilitating dependency injection and separation of responsibilities. For instance, consider an interface named Payable, which has a method such as process payment. This interface's functionality can be implemented by a variety of independent classes, including credit card, PayPal, and bank transfer. This method allows objects to be treated polymorphically using shared behaviour, enabling modular and testable code.

Interfaces, unlike classes that are abstract, provide multiple levels of inheritance, which makes them excellent for cases where cross cutting functionality such as logging and serialisation can be shared across unrelated class hierarchies. Interfaces that been designed perfectly have strong cohesion and small method sets, which are consistent with the interface segregation principle.

Abstract

Abstract classes serve as partly implemented templates whereby specific classes are derived. They support the insertion using both defined and undefined abstract methods. [5] for example a class called Shapes could consist of a common field named area with an abstract method called calculate area. Child classes such as a circle and rectangle may then inherit this parent class and apply their own area calculation accordingly. This method discourages code duplication and promotes uniform interfaces. As outlined recommendations for using abstract classes when several subclasses have common behaviour as well as when state or partial implementation should be inheritable. However, developers should avoid relying too heavily on abstract core classes, as this may give rise to rigidity and restrict composition. Abstract classes are best used to extend functionality rather than for basic contract definition, which would be more suited to interfaces.

Virtual methods

Polymorphism enables things to be treated according to shared types rather than their actual implementations. This is made possible in Java by using virtual method calls, which resolves method calls at runtime by considering the actual object type. Virtual methods enable dynamic dispatch and extension, but their use necessitates careful consideration of method overriding and behaviour predictability [6]. Studies emphasise the necessity of the Liskov Substitution Principle when using polymorphism subclasses must be substitutable for their base types without affecting program correctness. Violating LSP weakens the very abstraction that polymorphism is designed to give [7].

Storage structure

Performance, memory use, and adaptability are all directly affected by the storage structures used. In Java, dynamic structures like ArrayList, HashMap, and LinkedList provide flexibility of memory allocation and are commonly used for collecting data where the size is unknown in advance. Pheng [8] support for picking storage structures depending on actual level of complexities, such as HashMap for rapid key-value lookup, ArrayList for indexed access, and LinkedList for regular entries and removals. When efficiency is essential, the best technique is to minimise resizing and provide proper starting capacity.

Generics

Generics in Java enable type safe operations while reducing the requirement for explicit type casting. They improve code reuse and compile time type checking. The addition of generics to Java resulted in considerable gains in software safety, notably in API development. an optimal practice for generics is keeping the type parameters descriptive for example using T, E etc. using bounded wildcards for flexibility [9]. To make sure type safety is enforced the use of raw types must be avoided. Generics support the Don't Repeat Yourself concept by allowing container and utility classes to be reused across many kinds.

2. How my application adheres to these best practice

The following part demonstrates how each of the previously stated object-oriented characteristics have been successfully implemented in the Car Park application. Each code snippet demonstrates adherence to an appropriate principle and is taken straight from the system that was put in place.

Inheritance

Inheritance is employed to create hierarchical structures. For instance, IDReader is a parent class that gives a similar structure for several types of reader subclasses available such as Barcode Reader, QR Code Reader, Number Plate Reader and Ticket Reader all extend ID Reader and implement the read method defined in the superclass in their own way.

```
public class BarcodeReader extends IDReader  
{
```

```
public class NumberPlateReader extends IDReader  
{
```

```
public class TicketReader extends IDReader  
{
```

```
public class QRCodeReader extends IDReader  
{
```

Interfaces

The Fare Strategy interface provides a consistent payment contract throughout the various fare categories such as hourly, weekly and monthly payment that provides concrete implementations and allow specialised behaviours for each which is compatible with the concepts mentioned earlier.

```
public interface Fare 11 usages 3 implementations
{
    double calculateFare(int hoursParked);
}
```

```
public class HourlyPayment implements Fare 7 usages
{
    private static final double ratePerHour = 1.0; 1 usage

    @Override 9 usages
    public double calculateFare(int hoursParked)
    {
        return Math.max(1, hoursParked) * ratePerHour;
    }
}
```

```
public class MonthlyPayment implements Fare 9 usages
{
    private static final double monthlyFare = 220.0;
    @Override 9 usages
    public double calculateFare(int hoursParked)
    {
        return monthlyFare;
    }
}
```

```
public class WeeklyPayment implements Fare 11 usages
{
    private static final double weeklyFare = 60.0;
    @Override 9 usages
    public double calculateFare(int hoursParked)
    {
        return weeklyFare;
    }
}
```

Abstract Classes

The sign class is an example of abstract classes it's a superclass for displaying all related components and subclasses such as the Full Sign, the Fare Display and Opening Hours override available methods while inheriting a common structure the parent class provides.

```
public abstract class Sign 3 usages
{
```

```
    public abstract void show();
```

```
public class FareDisplay extends Sign
{
```

```
    @Override 6 usages
    public void show()
    {
        if (!isActive)
        {
            System.out.println("Inactive, Sorry ");
            return;
        }

        System.out.println("Fares:");
        System.out.println(" - £1 per hour");
        System.out.println(" - £60 per week");
        System.out.println(" - £220 per month");
    }
}
```

```
public class FullSign extends Sign 9
{
```

```
    @Override 6 usages
    public void show()
    {
        if (!isActive)
        {
            System.out.println("Inactive, Sorry unable to show uptod");
            return;
        }

        System.out.println("Full sign: " + (lightOn ? "On" : "OFF"))
    }
}
```

```
    @Override 6 usages
    public void show()
    {
        if (!isActive)
        {
            System.out.println("Inactive, sorry unable to sh");
            return;
        }

        System.out.println("Opening hours:");
        System.out.println(" - Monday to Friday: 24 hours");
        System.out.println(" - Saturday: 7:00 AM - 8:00 PM");
        System.out.println(" - Sunday: 9:00 AM - 6:00 PM");
    }
}
```

```
public class OpeningHours extends Sign
{
```

Virtual methods

Polymorphism is performed via sharing interfaces and overriding methods. For example, the Fare payment references can refer to various specific implementations. The right calculates Fare method is called dynamically according to the present object.

```
Fare fares = new HourlyPayment();  
double fare = fares.calculateFare()
```

Storage structure

Runtime data is efficiently managed with dynamic collections like HashMap and ArrayList. Such structures enable the effective search and monitoring of vehicles as they arrive and leave the car park.

```
private Map<String, Vehicle> activeVehicles = new HashMap<>();  
private Map<String, Vehicle> subscriptionRecords = new HashMap<>();  
private List<Vehicle> entryOrder = new ArrayList<>(); 3 usages  
private List<String> logs = new ArrayList<>(); 5 usages
```

ArrayList

The application stores vehicle entries in ArrayList because of its effective dynamic scaling and index access, both of which are required for tracking vehicle order as they arrive and leave the car park. ArrayList was chosen expressly to ensure that vehicle objects could be stored and retrieved in a reliable and efficient manner.

```
private List<Vehicle> entryOrder = new ArrayList<>(); 3 usages
```

3. Class attributes and methods and relationships with other classes

Sign

Key attributes:

```
private String id; 3 usages
private String location; 3 usages
protected boolean isActive = true; 6 usages
private boolean isClean = true; 3 usages
private boolean isFaulty = false; 3 usages
private boolean isRusty = false; 3 usages
private boolean isBroken = false; 3 usages
private LocalDate lastInspectionDate = LocalDate.now();
private String statusNotes = ""; 3 usages
```

Id indicates the id of the sign, location indicates the location of the sign for instance entrance and exit. Is clean indicates whether the sign is clean, is faulty indicates if the sign is faulty, is broken indicates whether the sign is broken, last inspected indicates last time the sign was inspected for any damage and status notes given on the sign. This class has many methods that just print the status of the sign which includes. Inspect method that records the present date as the last check date and adds a message with the inspector's name and time to the sign's status notes.

Deactivate which deactivates the sign and writes the cause in the state comments as an incident.

```
public void deactivate(String reason) no usages
{
    setActive(false);
    reportIssue(note: "Deactivated: " + reason);
}
```

Activate method that marks the sign as being operational and writes a message to the status notes stating the date of reactivation.

```
public void activate() no usages
{
    setActive(true);
    setStatusNotes(getStatusNotes() + "Reactivated on " + LocalDate.now() + "\n");
}
```

Clean method that declares the sign as clean and marks the cleaning event, with the date in the status note.

```
public void clean() no usages
{
    setClean(true);
    setStatusNotes(getStatusNotes() + "Cleaned on " + LocalDate.now() + "\n");
}
```

mark Rusty method that marks the sign as rusty and adds a note containing the detected date to the status notes.

```
public void markRusty() no usages
{
    setRusty(true);
    setStatusNotes(getStatusNotes() + "Rust detected on " + LocalDate.now() + "\n");
}
```

Mark broken marks the sign broken and write down the occurrence in the status notes, including the current date.

```
public void markBroken() no usages
{
    setBroken(true);
    setStatusNotes(getStatusNotes() + "Marked as broken on " + LocalDate.now() + "\n");
}
```

Show status methods that combines all the methods and indicates the status of the sign.

```
public void showStatus() no usages
{
    System.out.println("----- Sign Report -----");
    System.out.printf("ID: %-15s Location: %-20s\n",
        (getId() != null ? getId() : "N/A"),
        (getLocation() != null ? getLocation() : "Unknown"));
    System.out.printf("Active: %-10s Clean: %-10s\n",
        (isActive() ? "Yes" : "No"),
        (isClean() ? "Yes" : "Needs cleaning"));
    System.out.printf("Faulty: %-10s Rusty: %-10s Broken: %-10s\n",
        (isFaulty() ? "Yes - Needs fix" : "No"),
        (isRusty() ? "Yes - Check" : "No"),
        (isBroken() ? "Yes - Replace" : "No"));
    System.out.printf("Last Inspected: %s\n", getLastInspectionDate());
}
```

This class is related to other classes as it passes behaviour and attributes to the subclasses, and it is used in the car park for display.

Opening Hours

The opening hours class has 5 attributes weekdays, Saturday, Sunday, holiday mode, and emergency mode, and it inherits all the attributes from the super class sign. There is one method show which print the opening hours and special holiday hours and if there is an emergency. The opening hours is used in the car park class to inform the users of operating times and any special schedules.

```
private String weekdayHours;
private String saturdayHours;
private String sundayHours;
private boolean isHolidayMode;
private boolean emergencyMode;

public void show()
{
    if (!isActive)
    {
        System.out.println("Inactive, sorry unable to show operating hours");
        return;
    }
    System.out.println("Opening hours:");
    System.out.println(" - Monday to Friday: 24 hours");
    System.out.println(" - Saturday: 7:00 AM - 8:00 PM");
    System.out.println(" - Sunday: 9:00 AM - 6:00 PM");
}
```

Fare Display

This class has the following attributes, hourly, weekly, monthly and holiday rate to reflect the rates for different passes and it inherits all the attributes from the sign class. This class has one method called show that display the parking fares. The fare display is used primarily in car park class to display current parking fees for customers before entering the car park.

```
private double hourlyRate;
private double weeklyRate;
private double monthlyRate;
private double holidayRate;

public void show()
{
    if (!isActive)
    {
        System.out.println("Inactive, Sorry unable to show fare.");
        return;
    }
    System.out.println("Fares:");
    System.out.println(" - £1 per hour");
    System.out.println(" - £60 per week");
    System.out.println(" - £220 per month");
}
```

Full Sign

The class inherits attributes from the superclass and has two attributes light on and off which indicates whether the full light is on. There are few methods switch on and off and light on and off to toggle the current car park status. This class is used in the car park to indicate if the car par is full or spaces available.

```
private boolean lightOn;  
private boolean lightOff;
```

```
public void switchOn() 5  
{  
    lightOn = true;  
    lightOff = false;  
}
```

```
public void switchOff() 5  
{  
    lightOn = false;  
    lightOff = true;  
}
```

```
public void show()  
{  
    if (!isActive)  
    {  
        System.out.println("Inactive, Sorry unable to show uptodate  
        return;  
    }  
    System.out.println("Full sign: " + (lightOn ? "On" : "OFF"));
```

Fare interface

There are no attributes defined in this interface as these won't be constant variables and need to define in subclasses based on hourly, weekly and monthly payments. This has one method called calculate fare that takes the number of parked hours as a parameter it defines how a fare is calculated based on the number of hours parked. this interface is then implemented by other classes such as hourly, weekly, and Monthly payment which allows polymorphic behaviour for payment calculation.

```
public interface Fare 12 usages 3 implementations
{
    double calculateFare(int hoursParked); 10 usages
}
```

Hourly payment

This class has only one Private attribute hourly rate which defines the fee for parking for each hour parked.it has one method that calculates the cost of parking per hour called calculate Fare that takes in number of hours as a parameter. The price is calculated and returned based on the number of hours parked, with a minimum charge of one hour. This class implements the superclass Fare superclass. Also, the Vehicle class uses this to determine short term parking prices.

```
private static final double ratePerHour = 1.0;
```

```
public double calculateFare(int hoursParked)
{
    return Math.max(1, hoursParked) * ratePerHour;
}
```

Weekly payment

This class has only one attribute weekly rate which defines the fee for parking for a week.it has one method that calculates the cost of parking per week. The price is calculated and returned based on a weekly basis regardless of hours parked. This class implements the fare class. Also, the Vehicle class uses this for vehicles with weekly subscription pass.

```
private static final double weeklyFare = 60.0;
```

```
public double calculateFare(int hoursParked)
{
    return weeklyFare;
}
```

Monthly payment

The monthly fare has one attributes monthly rate for the fixed cost of monthly parking pass subscription. One method is defined calculate fare that returns the monthly cost of parking regardless of the number of hours parked. This class implements the fare strategy class. Also, the Vehicle class uses this for vehicles with monthly subscription pass.

```
private static final double monthlyFare = 220.0;
@Override 9 usages
public double calculateFare(int hoursParked)
{
    return monthlyFare;
}
```

Barrier

This class has many attributes like location which indicates the physical location of the barrier for example, entry or exit. Is active which determines whether the barrier is at present functional as expected. Is locked indicates if the barrier has been locked and shouldn't be raised or lowered for any reason. Is faulty indicates whether the barrier is broken or needs repair. Is manual mode on indicates whether the barrier's manual override mode is activated.

There are several methods such as raise that mimic lifting the barrier at its current location and prints an associated message. Lower which mimics lowering the barrier and prints a message. Update returns an update message reflecting that the barrier's state has been updated. Lock which locks the barrier and informs that user. Unlock which unlocks the barrier and displays a message. Enable Manual override which allows manual control of the barrier and displays a status message. disable manual override indicated turning off manual override and publishes the modification made.

The barrier class is used in the car park system to control entry and exits points of vehicles from the car park. It also works with other classes such as sensor in the car park class.

```
private String location;
private boolean isActive;
private boolean isLocked;
private boolean isFaulty;
private boolean isManualMo
private String serviceDate
```

```
public void raise() 4 usages
{
    System.out.println(getLocation() + " barrier raised.");
}
```

```
public void lower() 4 usages
{
    System.out.println(getLocation() + " barrier lowered.");
}
```

```
public void update() 1 usage
{
    System.out.println("Barrier at " + getLocation() + " updated.");
}
```

```
public void lock() 2 usages
{
    setLocked(true);
    System.out.println(getLocation() + " barrier locked.");
}
```

```
public void unlock() 2 usages
{
    setLocked(false);
    System.out.println(getLocation() + " barrier unlocked.");
}
```

```
public void enableManualOverride() no usages
{
    setManualModeOn(true);
    System.out.println("Manual mode enabled for " + getLocation() + " bar
}
```

```
public void disableManualOverride() no usages
{
    setManualModeOn(false);
    System.out.println("Manual mode disabled for " + getLocation() + " b
}
```

Sensor

This includes attributes like location which indicates the physical location of the sensor for example, entry or exit. The sensitivity level indicates the effectiveness the sensor is at detecting vehicles. Is operational determines whether the sensor is active and functional. Is faulty indicates whether the sensor is faulty, or it requires maintenance. Requires checkup indicates whether the sensor require repair or examination. Trigger count counts the total amount of times the sensor was successfully triggered. Last checkup is data in which the sensor was last checked up.

The sensor has methods like car present which asks the consumer for confirmation whether a car is present at the sensor's location, mimicking detection of a vehicle. update returns a message indicating that the sensor's state has been updated. Set sensitivity changes the sensitivity degree of the sensor and returns the new value. Increment trigger count calculates and presents the overall number of times a sensor has been triggered. Deactivate turns the sensor off and notifies a state change. Activate reinstates the sensor and returns confirmation. The sensor class is used in the car park class along the barrier in entry and exit to mimic the detection of a vehicle.

```
private String location; 3 usages
private int sensitivityLevel;
private boolean isOperational;
private boolean isFaulty; 3 usages
private boolean requiresCheckup;
private int triggerCount; 3 usages
private String lastCheckupDate;
```

```
public boolean carPresent() 2 usages
{
    Scanner stdin = new Scanner(System.in);
    System.out.print("Sensor triggered is there a vehicle at " + getLocation());
    return stdin.nextLine().trim().equalsIgnoreCase("y");
}
```

```
public void setSensitivity(int level) 1 usage
{
    setSensitivityLevel(level);
    System.out.println(getLocation() + " sensor sensitivity set to " + level);
}
```

```
public void deactivate() no usages
{
    setOperational(false);
    System.out.println(getLocation() + " sensor is now offline.");
}
```

```
public void deactivate() no usages
{
    setOperational(false);
    System.out.println(getLocation() + " sensor is now offline.");
}
```

ID Reader

Serial number is a unique device reader identifier. Retry attempts counts the number of read tries allowed. Is invalid indicates whether the reader has failed validation process. The brightness level indicates the current scanning brightness setting level. Is connected indicates whether the reader is connected to the system.

Retry increments and tracks the number of retry tries for reading. adjust Brightness determines the brightness level and records what was changed. mark Invalid marks the reader as faulty and displays the status. check Connection indicates and confirms that the reader is connected. read subclasses implement this abstract method to do actual reading logic. update indicates that the reader's status has been changed.

This class is the abstract superclass for all ID Readers such as Number Plate Reader, Ticket Reader, Barcode Reader and QR Code Reader. Each subclass customises the read logic using realistic restrictions and enhances I Reader behaviour to support numerous input types.

```
private String serialNumber;  
private int retryAttempts;  
private boolean isInvalid;  
private int brightnessLevel;  
private boolean isConnected;
```

```
public abstract boolean read(Scanner stdin);
```

```
public void adjustBrightness(int level) 1 usage  
{  
    brightnessLevel = level;  
    System.out.println("Brightness adjusted to " + level + " for IDReader [" + serialNumber + "]
```

```
public void retry() 11 usages  
{  
    retryAttempts++;  
    System.out.println("Retry attempt #" + retryAttempts + " for IDReader [" + serialNumber + "].
```

```
public void checkConnection() 3 usages  
{  
    isConnected = true;  
    System.out.println("IDReader [" + serialNumber + "] is connected.");
```

```
public void markInvalid() 6 usages  
{  
    isInvalid = true;  
    System.out.println("IDReader [" + serialNumber + "] marked as invalid.");
```

```
public void update() no usages  
{  
    System.out.println("IDReader [" + getSerialNumber() + "] update called.");  
}
```

Numberplate Reader

simulates license plate identification; may be affected by dirt or bad illumination.

```
private boolean isObstructed; 5 us
private boolean isDirty; 5 usages
private boolean lowLightDetected;
private boolean isAngleIncorrect;
```

```
public boolean read(Scanner stdin)
{
    System.out.print("Enter your number plate: ");
```

Ticket Reader

Accepts ticket codes and has logic for jammed or invalid tickets.

```
private boolean isFolded; 3 usages
private boolean isInsertedUpsideDown;
private boolean isUnreadableCode; 3 u
private boolean isJammed; 3 usages
private boolean isInvalid; 3 usages
```

```
public boolean read(Scanner stdin)
{
    System.out.print("Enter ticket code (TICKET-####): ");
    String ticket = stdin.nextLine().trim();
```

```

if (isFolded())
{
    System.out.println("Ticket scan failed ticket is folded.");
    retry();
    return false;
}

if (isInsertedUpsideDown())
{
    System.out.println("Ticket scan failed ticket inserted upside d
    retry();
    return false;
}

if (isUnreadableCode())
{
    System.out.println("Ticket scan failed code is unreadable.");
    markInvalid();
    return false;
}

if (isJammed())
{
    System.out.println("Ticket reader jammed clear and try again.")
    retry();
    return false;
}

```

Barcode Reader

Validates codes using a certain format and handles difficulties such as poor scan distance.

```

private boolean isTooFar; 3 usages
private boolean isMisaligned; 3 usages
private boolean backgroundInterference;

```

```
System.out.print("Enter barcode like this (BAR-XXXX): ");
String barcode = stdin.nextLine().trim();

if (isTooFar())
{
    System.out.println("Failed: barcode is far move it closer."
    retry();
    return false;
}

if (isMisaligned())
{
    System.out.println("Failed: barcode is not aligned correctly
    retry();
    return false;
}

if (barcode.matches(regex: "BAR-[A-Z0-9]{4}"))
{
    checkConnection();
    return true;
} else {
    retry();
    markInvalid();
    return false;
}
```

QR Code Reader

simulates QR scanning circumstances such as glare and out-of-focus scans.

```
private boolean hasGlare; 5 us
private boolean isOutOfFocus;
private double scanDistance;
System.out.print("Enter QR code (QR-XXXX): ");
String qr = stdin.nextLine().trim();
```

Vehicle

This class represents one vehicle in the car park, encapsulating every important identifying, timing, and payment information. It has attributes such as number plate, make, model, colour and type to distinguish vehicles in the car park and makes it easier to read logs. Entry time uses local timestamp for entries. Fare payment reference for fare calculations. Subscription expiry and the end date and time of monthly and weekly passes.

The class contains methods like has valid subscription which returns true if the vehicle has a valid subscription for weekly or monthly payment. Calculates Fare assigns the designated fare payment to determine fare. To string method gives a formatted string illustrating the car and its key features.

```
private String numberPlate; 3 usages
private VehicleType type; 3 usages
private String make; 3 usages
private String model; 3 usages
private String color; 3 usages
private LocalDateTime entryTime; 5 usages
private Fare fare; 2 usages
private LocalDateTime subscriptionExpiry;
```

```
return String.format(
    "Vehicle [Plate: %s, Type: %s, Make: %s, Model: %s, Color: %s, Entry: %s, Expiry: %s]"
    numberPlate, type, make, model, color,
    entryTime, (subscriptionExpiry != null ? subscriptionExpiry : "N/A")
```

Vehicle type

Outlines the range of vehicle types that are allowed in the parking system. Simplifies validation and logic in classes like vehicle and car park and ensures type safety by limiting vehicle classification to recognised categories.

```
public enum VehicleType {
    CAR, 1 usage
    VAN 1 usage
}
```

Car park

This class controls the parking system centrally. It manages vehicle entering and leaving, tracks parking availability, maintains subscriptions passes, and keeps administration records. It works with different system components to imitate real-world parking procedures. It includes attributes like capacity and frees spaces to keep track of the overall number of parking spaces that are available. Active vehicles organise parked vehicles by plate number. Subscription records track vehicles with active weekly or monthly subscriptions passes Entry order logs vehicles in the order they arrived. logs stores system events like entries, exits, and rejections.

The scanner takes input from users for simulation. Initiated components such as sensor and barrier at entrance and exit and display signs like full sign, fares and operating hours.

This class has methods such as run which runs main loop that asks the user to enter, exit, or view the admin. Handle Entry that controls ID reading, fare type handling, number plate validation, and vehicle entry. Handle exit clears space, processes vehicle exits, and computes payments if necessary. Simulate payment models how a user would interact with a contactless or credit card payment. Also get validated plate, get fare payment, get vehicle type, and get ID reader methods that gather and verify data.

This class Interacts with all major application components: payment, vehicle types, readers, display signs, sensor, and barrier.

```
private int capacity = 1;
private int freeSpaces = 1;
```

```
private Sensor entranceSensor = new Sensor( location: "Entrance"); 1 u
private Sensor exitSensor = new Sensor( location: "Exit"); 1 usage
private Barrier entranceBarrier = new Barrier( location: "Entrance");
private Barrier exitBarrier = new Barrier( location: "Exit"); 2 usages
private FullSign fullSign = new FullSign(); 10 usages
private FareDisplay fareDisplay = new FareDisplay(); 1 usage
private OpeningHours openingHours = new OpeningHours(); 1 usage
```

```
public void run() 1 usage
{
    boolean running = true;
    while (running)
    {
        System.out.println("\n--- IQ Building Park System ---");
        System.out.println("1. Enter Car Park");
        System.out.println("2. Exit Car Park");
        System.out.println("3. Admin View");
        System.out.println("4. Exit Program");
        System.out.print("Select an option: ");
```

```
public void incrementSpaces() 1 usage
{
    if (freeSpaces < capacity) freeSpaces++;
}
```

```
public void decrementSpaces() 2 usages
{
    if (freeSpaces > 0) freeSpaces--;
}
```

```
private void handlePayment() 3 usages
{
    System.out.println("Choose payment method: 1. Contactless 2. Card");
    String method = scanner.nextLine();
    if (method.equals("2"))
    {
        System.out.print("Enter card number: ");
        scanner.nextLine();
        System.out.print("Enter PIN: ");
        scanner.nextLine();
        System.out.println("Payment successful by card.");
    } else
    {
        System.out.println("Contactless payment accepted.");
    }
}
```

```
private void handleEntry() 1 usage
{
    openingHours.show();
    fareDisplay.show();
    System.out.println("Available Spaces: " + spacesLeft() + " | FullSign");

    if (spacesLeft() == 0)
    {
        fullSign.switchOn();
        logs.add("Entry denied - car park full");
        System.out.println("Car park is full.");
        return;
    }
}
```

```
private void showAdmin() 1 usage
{
    System.out.println("\n--- Admin View ---");
    System.out.println("Capacity: " + capacity);
    System.out.println("Free Spaces: " + freeSpaces);
    System.out.println("Occupied: " + activeVehicles.size());
}
```

4. Static Memory Allocation and Dynamic Memory Allocation

In Java, memory allocation is a critical factor that determines application speed as well as adaptability. Knowing the difference between dynamic and static memory allocation is essential to creating effective data structures, particularly in systems such as a parking management system.

Static Memory Allocation

Static memory allocation happens during compile time and ensures that the memory size remains unchanged during the execution of the program. Arrays of data in Java serve as an excellent illustration once an array is declared with a fixed size, it cannot hold more elements than its initially declared size at the beginning. Another example is the usage of primitive variables or constants and static variable, which memory is assigned at compile time and remains constant during the program's execution. consider this constant static variable below that holds the car park max numbers of vehicles allowed in the parking area. Because it is marked as final, the value it holds cannot be changed once assigned, and memory is allocated statically when the class is loaded. This predictability can lead to improved performance due to lower overhead. nevertheless, it lacks adaptability, leaving it unsuitable for applications with data that has variable sizes. [Reference](#)

```
final int carParkMaxCapacity = 100;
```

Dynamic Memory Allocation

In contrast with static memory allocation, dynamic memory allocation happens during runtime. Java accomplishes dynamic memory allocation using the new keyword, which generates an object on the heap of memory and provides the reference held on the stack. This makes it possible for an even more adaptable usage of memory, allowing the program to react to modifications in the amount of data or system state.

The heap memory area manages Java's dynamic allocation, and the garbage collector constantly cleans it up, reclaiming memory used by no longer referenced objects. This helps developers to concentrate on logic rather than manual memory management, while nevertheless making optimal use of resources.

In Java, dynamic memory allocation is demonstrated by the use of adjustable data structures like ArrayList, HashMap, LinkedList, HashSet, TreeMap and many more, which can allocate memory on the heap at runtime to handle variable data sizes. Dynamic memory is very beneficial for coping with user generated input with unknown data size, collections that vary over time and object instantiation according to program flow.

Below code is a demonstration of dynamic memory allocation with Java's integrated data structures. With this example, the list of cities in the array list can expand or contract dynamically throughout the execution of the application. Unlike fixed-size arrays as shown earlier in the car park example, it responds to modifications in the amount of data without needing previous understanding of the total amount of elements in list of cities.

```
ArrayList<String> city = new ArrayList<>();
city.add("Swansea");
city.add("Cardiff");
```

Comparison table

feature	Static allocation	Dynamic Allocation
Timing	Compile time	Run time
Memory allocation	Method area, stack	Heap
Sizing	Fixed	Flexible allows resizing
Allocation method	With static final	Using the new key word
Best for	Constants	Collections, object instantiation
Examples	int [numbers = new int[2]	List<Integer> numbers = new ArrayList<> ();

Java memory management architecture depends significantly upon both static and dynamic memory allocation. Static allocation is predictable and efficient for fixed size and unchangeable data, whereas dynamic allocation gives the adaptability required for working with variable sized or changing sets of data. The decision among the two is largely dictated by the demands of the application and the type of the data structures concerned. Understanding these distinctions is crucial in determining the best data format for a particular issue.

5. Best data structures approach for the Car Park

The choice of suitable structures for data is an important design consideration for developing a reliable and effective Car Park Management Solution. The application's dynamic nature managing vehicles entry and exits, tracking subscriptions, recording activity, and managing parking capacity needs adaptable, memory efficient structures that can expand at runtime. This section assesses numerous key Java data structures and explains the decision to choose ones according to functionality, time complexity, and design fit.

These key requirements behind the selection of data structure, the Car Park application must track vehicles coming and exiting in real time, prevent repeated vehicles entries, enable quick searches by the license plate or ticket ID etc. Track the sequence of vehicle entry for possible audit, allow adjustable scaling as the vehicle count changes, following these circumstances, these structures were chosen.

Hash map

A HashMap is currently used to keep track of vehicles that are currently parked in the parking area and vehicles that holds an active subscription weekly or monthly so that these vehicles enter and exit the car park seamlessly without having to go through the payment process upon entries and exits.

This use of HashMap provides a quick search by license plate, which is a vital component for verifying vehicle presence or re-entry. **According to Sedgewick and Wayne (2011)**, hash-based structures are ideal when quick key-based access is required, particularly in applications where the key field (for example, license plates) is non-numeric and unique.

```
private Map<String, Vehicle> activeVehicles = new HashMap<>();  
private Map<String, Vehicle> subscriptionRecords = new HashMap<
```

Array List

The application utilises an ArrayList for keeping the order of when vehicles enter the car park. This is essential for traceability admin views can view full logs easily in a sequence order.

An ArrayList is preferred over a LinkedList because it has better cache performance and faster indexed access. Although insertions at random locations are more expensive, in this situation, vehicles are added at the end, making it ideal. **Bengtsson and Karlsson (2015)** showed that ArrayList outperformed LinkedList in cases based on sequential additions and retrieves, which is compatible with Car park application's requirements of entry recording functionality.

```
private List<Vehicle> entryOrder = new ArrayList<>()  
private List<String> logs = new ArrayList<>(); 5 usa
```

Other Data Structures Considered but not use:

LinkedList intended for vehicle log storage but eliminated because of inadequate random-access effectiveness and increased memory overhead. Tree Map to provides sorted order, although it is irrelevant in this case because neither license plate nor ID ordering is not necessary. HashSet not utilised because duplication avoidance has been handled by HashMap keys.

6. Unit Testing

A comprehensive set of unit tests was created to guarantee the accuracy, dependability, and resilience of the Car Park system. Junit was used to create the unit tests. Every class has a specific test file that verifies error handling, boundary cases, and expected behaviour.

Vehicle type Enum

- Tested that only “CAR” and “VAN” are accepted vehicle types.
- Attempted to parse an invalid type (e.g., “LORRY”) and confirmed that an exception is thrown.

Payment classes

- confirmed that the fare calculations for different hours and durations were accurate.
- claimed that the minimum charge was applied correctly.
- Verified that invalid scenarios either produce the right results or throw exceptions.

Barrier

- Barrier raising and lowering techniques were tested to guarantee proper output and state.
- Lock, unlock and manual override functionality were confirmed.

Sensor

- used simulated input to test the detection of cars.
- Methods for trigger counting and sensitivity adjustment were tested.

Sign classes

- Confirmed show, display methods provide accurate data.
- tested state transitions, including cleaning, deactivation, and activation.

Vehicle

- verified that the fields were valid, and the instantiation was correct.
- confirmed the weekly/monthly subscription logic.
- Fare calculation and string representation were tested.

ID Readers

- confirmed proper format validation and input handling.
- Additional logic for error scenarios was tested.

7. Screenshots of working application

Main men:

```
--- IQ Building Park System
1. Enter Car Park
2. Exit Car Park
3. Admin View
4. Exit Program
Select an option: |
```

Exit application:

```
--- IQ Building Park System ---
1. Enter Car Park
2. Exit Car Park
3. Admin View
4. Exit Program
Select an option: 4

Process finished with exit code 0
```

Enter vehicle:

```
Select an option: 1
Opening hours:
- Monday to Friday: 24 hours
- Saturday: 7:00 AM - 8:00 PM
- Sunday: 9:00 AM - 6:00 PM
Fares:
- £1 per hour
- £60 per week
- £220 per month
Available Spaces: 1 | FullSign: OFF
Sensor triggered is there a vehicle at Entrance? (y/n): |
```

Vehicle details:

```
Sensor triggered is there a vehicle at Entrance? (y/n): y
Enter number plate: YP58PZJ
Choose vehicle type car or van: CAR
Enter vehicle make: Ford
Enter vehicle model: Fiesta
Enter vehicle color: Green
```

Weekly payment:

Test using contactless payment.

```
Choose fare type: 1. Hourly 2. Weekly 3. Monthly
2
Weekly pass selected. Payment due: £60.
Choose payment method: 1. Contactless 2. Card & PIN
1
Contactless payment accepted.
```

Monthly payment:

Test using card & PIN.

```
Choose fare type: 1. Hourly 2. Weekly 3. Monthly
3
Monthly pass selected. Payment due: £220.
Choose payment method: 1. Contactless 2. Card & PIN
2
Enter card number: 1234567891111111
Enter PIN: 1234
Payment successful by card.
Subscription valid for 30 days.
```

Car entry:

Hourly payment due upon vehicle exit.

```
Choose fare type: 1. Hourly 2. Weekly 3. Monthly
1
Select ID method: 1. Number Plate 2. Ticket 3. Barcode 4. QR Code
1
Reading based on Number Plate...
Enter your number plate: YP58PZJ
IDReader [NPR-001] is connected.
Entrance barrier raised.
Vehicle Entered: Vehicle [Plate: yp58pzj, Type: CAR, Make: Ford , Model: Fiesta, Color: Green, Entry
Entrance barrier lowered.
Remaining Spaces: 0 | FullSign: ON
```

Entering another vehicle when car park is full.

```
--- IQ Building Park System ---
1. Enter Car Park
2. Exit Car Park
3. Admin View
4. Exit Program
Select an option: 1
Opening hours:
- Monday to Friday: 24 hours
- Saturday: 7:00 AM - 8:00 PM
- Sunday: 9:00 AM - 6:00 PM
Fares:
- £1 per hour
- £60 per week
- £220 per month
Available Spaces: 0 | FullSign: ON
Car park is full.
```

Admin View:

```
--- IQ Building Park System
1. Enter Car Park
2. Exit Car Park
3. Admin View
4. Exit Program
Select an option: 3
```

```
Currently Parked Vehicles:
Vehicle [Plate: yp58pzj, Type: CAR, Make: Ford, Model: Fiesta, Color: Green, Entry: 2025-04-25T10:59

Entry Order:
Vehicle [Plate: yp58pzj, Type: CAR, Make: Ford, Model: Fiesta, Color: Green, Entry: 2025-04-25T10:59

Recent Logs:
[ENTRY LOG] Vehicle [Plate: yp58pzj, Type: CAR, Make: Ford, Model: Fiesta, Color: Green, Entry: 2025
```

Exit:

```
--- IQ Building Park System ---
1. Enter Car Park
2. Exit Car Park
3. Admin View
4. Exit Program
Select an option: 2
Sensor triggered is there a vehicle at Exit? (y/n): y
Select ID method: 1. Number Plate 2. Ticket 3. Barcode 4.
1
Reading based on Number Plate...
Enter your number plate: YP58PZJ
IDReader [NPR-001] is connected.
Enter your number plate or code: YP58PZJ
Enter hours parked: 1
Payment due: £1.0
Choose payment method: 1. Contactless 2. Card & PIN
1
Contactless payment accepted.
```

```
Choose payment method: 1. Contactless 2. Card & PIN
1
Contactless payment accepted.
Exit barrier raised.
Vehicle Exited: Vehicle [Plate: yp58pzj, Type: CAR, Make: Ford , Model: Fiesta, Color: Green, Entry
Exit barrier lowered.
Remaining Spaces: 1 | FullSign: OFF
```

User Guide

Launch the application:

- Launch the application by running main it will display the main menu.

Choices on the main menu:

- Enter the car park: Enter the vehicle's information, pick a payment option, and select an identification method by following the instructions provided.
- Exit the car park: Enter the hours parked, if necessary, provide proof of vehicle identification such as a code or number plate, and process payment if necessary.
- Admin View: See details about available spots, activity logs, parked cars, and entry order.
- Exit Program: Closes the program.

When entering:

- Enter your vehicle number plate, make, model, colour, and type of vehicle.
- Choose a payment option hourly, weekly, or monthly and make the required payment.
- Enter the appropriate code after selecting an identification method such as a ticket, barcode, QR code, or license plate.

While Leaving:

- Choose the method of identification and enter your code or license plate.
- Enter the number of hours parked and make the calculated payment if you are not a weekly or monthly payment.

General:

- Type answers precisely as instructed for example, car or van for vehicle type.
- To guarantee a seamless entry and exit, adhere to the directions for each step.

1. Bibliography

- [1] A. Taivalsaari, "On the notion of inheritance," *ACM Computing Surveys*, vol. 28, no. 3, pp. 438–479, Sep. 1996, doi: <https://doi.org/10.1145/243439.243441>
- [2] G. B. Singh, "Single versus multiple inheritance in object oriented programming," *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, Jan. 1995, doi: <https://doi.org/10.1145/209866.209871>.
- [3] E. Tempero, J. Noble, and H. Melton, "How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software," *ECOOP 2008 – Object-Oriented Programming*, pp. 667–691, doi: https://doi.org/10.1007/978-3-540-70592-5_28.
- [4] Brahma Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*. Springer, 2015.
- [5] M. Lavand and R. Pawar, "A Study On Abstraction in Object Oriented Programming with Java and Its Implementation in Developing Programs," *International Journal on Science and Technology (IJSAT)* IJSAT25023438, vol. 16, no. 2, 2025, Available: <https://www.ijسات.org/papers/2025/2/3438.pdf>
- [6] N. Milojkovic, A. Caracciolo, M. F. Lungu, O. Nierstrasz, D. Rothlisberger, and R. Robbes, "Polymorphism in the Spotlight: Studying Its Prevalence in Java and Smalltalk," *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, doi: <https://doi.org/10.1109/icpc.2015.29>.
- [7] W. Al-Ahmad, "A framework for conceptual modeling in OOP," *Journal of the Franklin Institute*, vol. 343, no. 4–5, pp. 532–544, Jul. 2006, doi: <https://doi.org/10.1016/j.jfranklin.2006.02.035>.
- [8] S. Pheng and C. Verbrugge, "Dynamic Data Structure Analysis for Java Programs," *IEEE Xplore*, Jun. 01, 2006. <https://ieeexplore.ieee.org/abstract/document/1631121>.
- [9] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, Dec. 2012, doi: <https://doi.org/10.1007/s10664-012-9236-6>.