Overloading, Java API, Advanced structures , Nesting, Wrapping

Lecture 4, 2023

University American College Skopje
School of Computer Science and Information Technology
Course: Programming Languages
Prepared by: Dejan Mitov

Method Signature Concept

Method Signature Concept

- In Java, a method signature is a unique identifier for a method. It is composed of the method name and the parameter list. The parameter list includes the number and types of the parameters that the method takes.
- The method signature does not include the return type of the method. This is because the return type is not used to distinguish between overloaded methods.

```
public void add(int a, int b) {}
public double multiply(double a, double b) {}
public String greet(String name) {}
```

Method Overloading

- Method overloading is a feature in Java that allows you to have multiple methods with the same name, but with different parameter lists. This can be useful when you need to perform the same basic operation on different types of data.
- For example, you could have two add() methods, one that takes two integers as parameters and the other that takes two doubles as parameters.
- Now, you can call the add() method with either two integers or two doubles, depending on the type of data you need to add.

```
public class MathUtils {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

Role in Method Overloading

- Two or more methods are considered overloaded if they share the same name but have different parameter lists.

- In Java, overloaded methods differ from one another due to their distinct method signatures.

```java
public class Person {
    private String name;
    private Date dateOfBirth;

    public Person(String name, Date dateOfBirth) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }

    public Person(String name) {
        this.name = name;
    }
}
```

Constructor Overloading

- Constructor overloading is similar to method overloading, but it applies to constructors instead of methods. A constructor is a special method that is used to create a new object.
- Constructor overloading allows you to have multiple constructors for the same class, each with a different parameter list. This can be useful when you need to create objects in different ways.
- For example, you could have two constructors for the Person class, one that takes a name and a date of birth as parameters and the other that takes just a name.

```java
// Method overloading
public class MathUtils {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }
}

// Example usage:
int sumOfInts = MathUtils.add(1, 2);
double sumOfDoubles = MathUtils.add(1.0, 2.0);
String sumOfStrings = MathUtils.add("Hello", "World!");
```

```java
// Constructor overloading
public class Person {
    private String name;
    private Date dateOfBirth;

    public Person(String name, Date dateOfBirth) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }

    public Person(String name) {
        this.name = name;
    }
}

// Example usage:
Person person1 = new Person("John Doe");
Person person2 = new Person("Jane Doe", new Date());
```

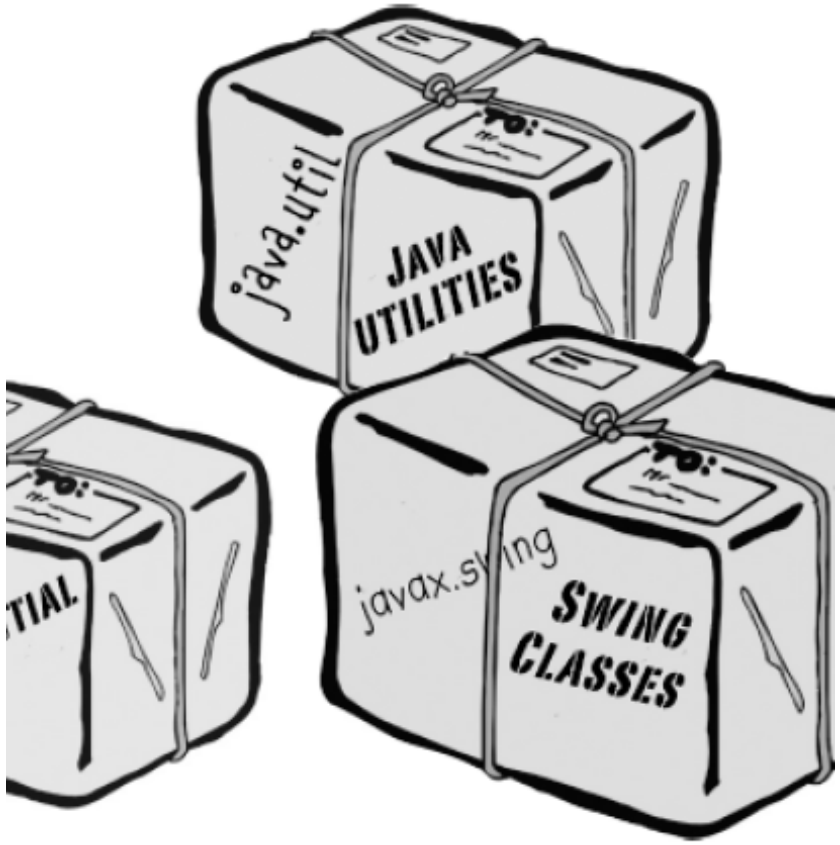
Key points

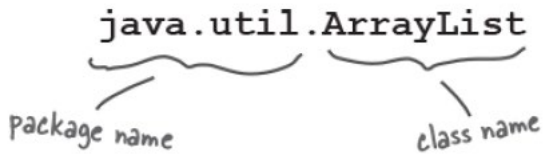
- What is a method signature?
  A method signature is composed of the method name and the parameter list. It is used to uniquely identify a method.
- What is the difference between method overloading and constructor overloading?
  Method overloading allows you to have multiple methods with the same name, but with different parameter lists. Constructor overloading allows you to have multiple constructors for the same class, each with a different parameter list.


Java API

Java API

- What is Java API?
  - A set of pre-packaged libraries and classes to facilitate common programming tasks.
- Java API is like a toolbox for Java developers. The API includes classes for everything from basic data types to complex graphical user interfaces.
- The Java API is divided into several packages, each of which contains classes and interfaces related to a specific area of functionality.
- *For example, the java.lang package contains classes for basic data types and object-oriented programming concepts, while the java.util package contains classes for common data structures and algorithms.*
- https://docs.oracle.com/en/java/javase/17/docs/api/






```java
//Working with Java Strings - java.lang.String
String text = "Hello, World!";
System.out.println(text.toUpperCase());

//Managing Collections - java.util
import java.util.ArrayList;
ArrayList<String> list = new ArrayList<>();
list.add("Java");

//File Operations - java.io
import java.io.File;
File myFile = new File("example.txt");

//Networking - java.net
import java.net.URL;
URL myURL = new URL("http://www.example.com/");

//Database Access - java.sql
import java.sql.Connection;
import java.sql.DriverManager;

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/db", "user", "pass");
```

Key point on Java API

Java API serves as an extensive toolbox that aids in a variety of tasks, ranging from basic operations like string manipulation to more complex functionalities like networking and database access.

Java lists, maps, and hashtables

Lists

- Java lists are ordered collections of objects. They are implemented using the java.util.List interface. The List interface provides a variety of methods for adding, removing, and searching for elements in a list.

- Some of the most common methods for working with lists include:
  - add(): Adds an element to the end of the list.
  - remove(): Removes an element from the list.
  - get(): Returns the element at a specific index in the list.
  - contains(): Checks if the list contains a specific element.
  - indexOf(): Returns the index of a specific element in the list, or -1 if the element is not in the list.

```java
// Create a new list
List<String> names = new ArrayList<>();

// Add some elements to the list
names.add("Alice");
names.add("Bob");
names.add("Carol");

// Remove an element from the list
names.remove("Carol");

// Get the element at a specific index in the list
String name = names.get(0);

// Check if the list contains a specific element
boolean containsAlice = names.contains("Alice");

// Print all of the elements in the list
for (String name : names) {
    System.out.println(name);
}
```

Types of lists

- Java provides several types of lists that can be used to store a collection of elements. Some of the commonly used lists in Java are:
  - ArrayList: It is a dynamic array that can grow and shrink as per the requirement. It is faster than LinkedList for random access and iteration, but slower for insertion and deletion operations. It allows duplicate elements and null values.
  - LinkedList: It is a doubly linked list that can grow and shrink as per the requirement. It is faster than ArrayList for insertion and deletion operations, but slower for random access and iteration. It allows duplicate elements and null values.
  - Vector: It is similar to ArrayList, but it is synchronized, which makes it thread-safe. It is slower than ArrayList because of the synchronization overhead.
  - Stack: It is a subclass of Vector that implements a stack data structure. It follows the LIFO (Last In First Out) principle, which means the last element added to the stack will be the first one to be removed.

```
ArrayList<String> list = new ArrayList<String>(5);
```

- The number inside the parentheses of `ArrayList<String> list = new ArrayList<String>(5)` specifies the initial capacity of the ArrayList, which is the number of elements that the ArrayList will allocate to begin with as the internal storage of items.
- The initial capacity is simply a hint to the JVM about how much space to allocate for the ArrayList when it is created. If you don't specify an initial capacity, the default value of 10 will be used
- Specifying an initial capacity can be useful when you know in advance how many elements you will be adding to the ArrayList. By providing an initial capacity that is close to the final size of the ArrayList, you can avoid unnecessary resizing operations when adding elements to the list, which can improve performance. However, if you don't know in advance how many elements you will be adding to the list, it's best to leave the initial capacity unspecified or use a value that is larger than what you expect to need.

Maps

- Java maps are unordered collections of key-value pairs. They are implemented using the java.util.Map interface. The Map interface provides a variety of methods for adding, removing, and searching for key-value pairs in a map.

- Some of the most common methods for working with maps include:
  - put(): Adds a key-value pair to the map.
  - get(): Returns the value associated with a specific key in the map.
  - remove(): Removes the key-value pair associated with a specific key from the map.
  - containsKey(): Checks if the map contains a specific key.
  - containsValue(): Checks if the map contains a specific value.

```java
// Create a new map
Map<String, Integer> ages = new HashMap<>();

// Add some key-value pairs to the map
ages.put("Alice", 25);
ages.put("Bob", 30);
ages.put("Carol", 35);

// Remove a key-value pair from the map
ages.remove("Carol");

// Get the value associated with a specific key
Integer age = ages.get("Alice");

// Check if the map contains a specific key
boolean containsAlice = ages.containsKey("Alice");

// Print all of the key-value pairs in the map
for (Map.Entry<String, Integer> entry : ages.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Hashtables

- Java hashtables are a type of map that use a hash function to efficiently store and retrieve key-value pairs. Hashtables are implemented using the java.util.Hashtable class.

- Hashtables are similar to maps in many ways, but there are a few key differences:
  - Hashtables are thread-safe, meaning that they can be safely accessed by multiple threads at the same time.
  - Hashtables do not allow null keys or values.
  - Hashtables are less efficient than maps for some operations, such as iteration.

```java
// Create a new hashtable
Hashtable<String, Integer> ages = new Hashtable<>();

// Add some key-value pairs to the hashtable
ages.put("Alice", 25);
ages.put("Bob", 30);
ages.put("Carol", 35);

// Remove a key-value pair from the hashtable
ages.remove("Carol");

// Get the value associated with a specific key
Integer age = ages.get("Alice");

// Check if the hashtable contains a specific key
boolean containsAlice = ages.containsKey("Alice");

// Print all of the key-value pairs in the hashtable
Enumeration<String> keys = ages.keys();
while (keys.hasMoreElements()) {
    String key = keys.nextElement();
    Integer value = ages.get(key);
    System.out.println(key + ": " + value);
}
```

Key points

- Lists and arrays are both data structures in Java that are used to store collections of data. However, there are some key differences between the two.

- Arrays are fixed in size, meaning that the number of elements that an array can store cannot be changed once it is created. Lists, on the other hand, are dynamic in size, meaning that they can grow and shrink as needed.

- Arrays are also more efficient than lists in terms of performance. This is because arrays are stored in contiguous memory, while lists are not.

key differences between lists and arrays

Feature	List	Array
Size	Dynamic	Fixed
Performance	Less efficient	More efficient
Traversing	Easier	More difficult
Searching	Easier	More difficult
Sorting	Easier	More difficult
Inserting and deleting elements	Easier	More difficult

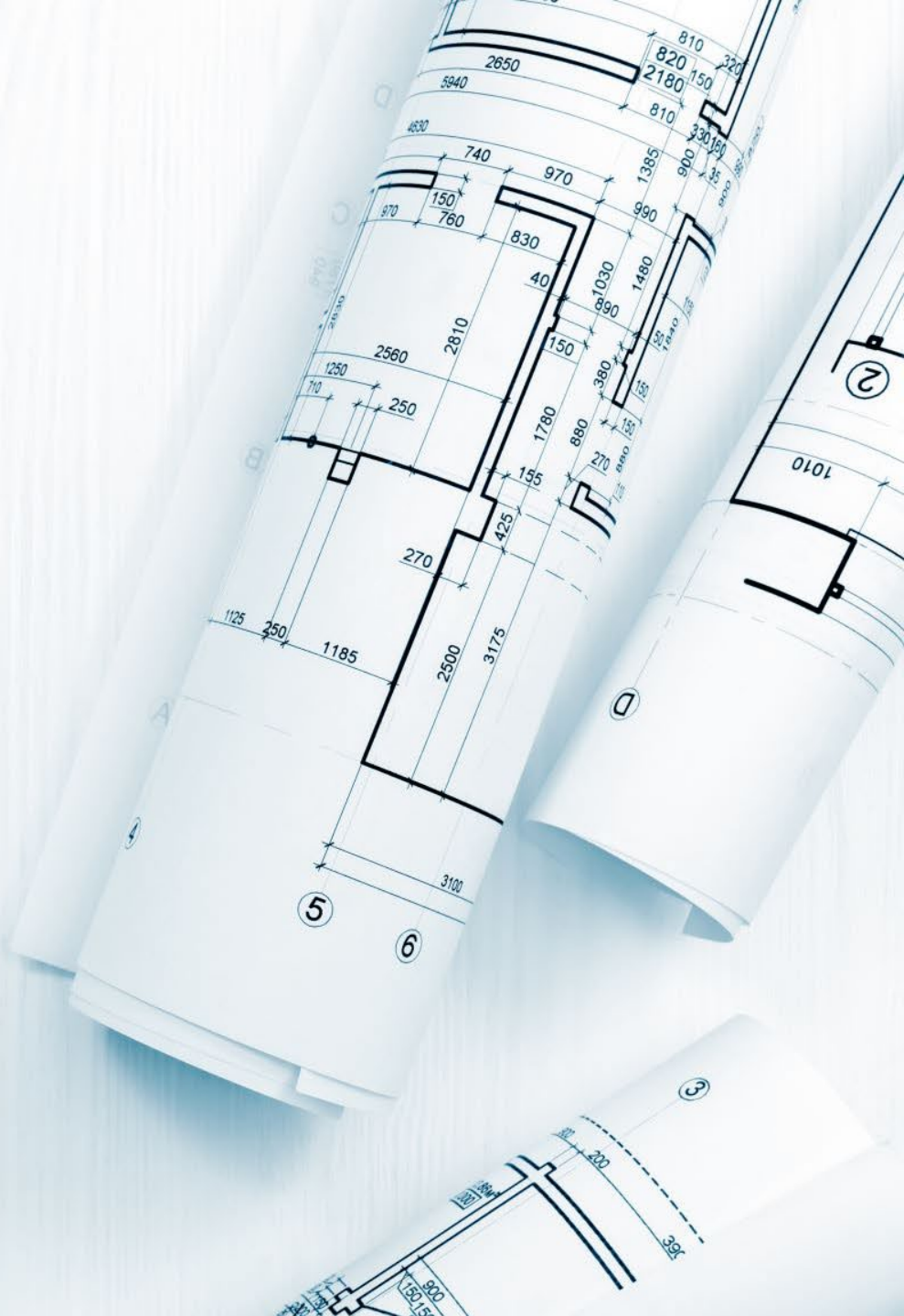
When to use lists and when arrays?

- When to use lists
  - Lists are a good choice when you need a data structure that is dynamic in size or when you need to be able to easily insert and delete elements. Lists are also a good choice when you need to be able to traverse or search the data structure frequently.
- When to use arrays
  - Arrays are a good choice when you need a data structure that is efficient in terms of performance or when you need to be able to access the elements of the data structure by index. Arrays are also a good choice when you know the exact number of elements that the data structure will need to store.

Examples of when you might use lists and arrays

- Lists:
  - To store a list of student names in a class.
  - To store a list of products in a shopping cart.
  - To store a list of items to be processed by a queue.
- Arrays:
  - To store a grid of pixels in an image.
  - To store a lookup table of values.
  - To store a list of mathematical constants.
- In general, lists are more flexible and easier to use than arrays. However, arrays are more efficient in terms of performance. Therefore, the best data structure to use depends on the specific needs of your application.

Java Scope Rules



Java Scope Rules

- In Java, the scope of a variable is the region of the program where it can be accessed.
- The scope of a variable is determined by where it is declared in the program.

Four types of scope in Java

- Class scope: Variables declared outside of any method or block in a class have class scope. This means that they can be accessed by any method in the class.
- Method scope: Variables declared inside a method have method scope. This means that they can only be accessed by the code within the method in which they are declared.
- Block scope: Variables declared inside a block of code, such as an if statement or a for loop, have block scope. This means that they can only be accessed by the code within the block in which they are declared.
- Local variable scope: Variables declared as method parameters or as part of a for loop header have local variable scope. This means that they can only be accessed by the code within the method or loop in which they are declared.

```
{
    int x = 10;
}

System.out.println(x); // This line will not compile because x is not declared in the scope of the `System.out.println()` statement.
```

Identifiers in Different Blocks of Statements

- If an identifier is declared inside a block of code, it can only be accessed by the code within that block.

Method Names and Signatures Within Different Classes

- Method names and signatures can be the same in different classes, as long as the classes are in different packages. For example, the following code is valid.
- If you try to create two methods with the same name and signature in the same class, you will get a compiler error.

```java
package com.example.package1;

public class Class1 {
    public void myMethod() {
        System.out.println("Hello from Class1!");
    }
}

package com.example.package2;

public class Class2 {
    public void myMethod() {
        System.out.println("Hello from Class2!");
    }
}
```

```java
package com.example.package1;

public class Class1 {
    public void myMethod() {
        System.out.println("Hello from Class1!");
    }
}

package com.example.package2;

public class Class1 {
    public void myMethod() {
        System.out.println("Hello from Class2!");
    }
}
```

Classes with Identical Names but in Different Packages

- Classes with identical names can be created in different packages. For example, the following code is valid.

- To distinguish between the two classes, you need to use the fully qualified class name, which includes the package name. For example, to call the myMethod() method in the Class1 class in the com.example.package2 package, you would use the following code:

```java
com.example.package2.Class1 class1 = new com.example.package2.Class1();
class1.myMethod();
```

```java
public class ScopeExample {
    public static void main(String[] args) {
        int x = 10; // This variable has class scope.

        {
            int y = 20; // This variable has block scope.
```

Key Points on Scope

- There are four types of scope in Java: class scope, method scope, block scope, and local variable scope.
- Identify the scope of a variable. To do this, look at where the variable is declared in the program.
- Understand how scope rules affect the accessibility of variables. For example, a variable declared inside a block of code can only be accessed by the code within that block.

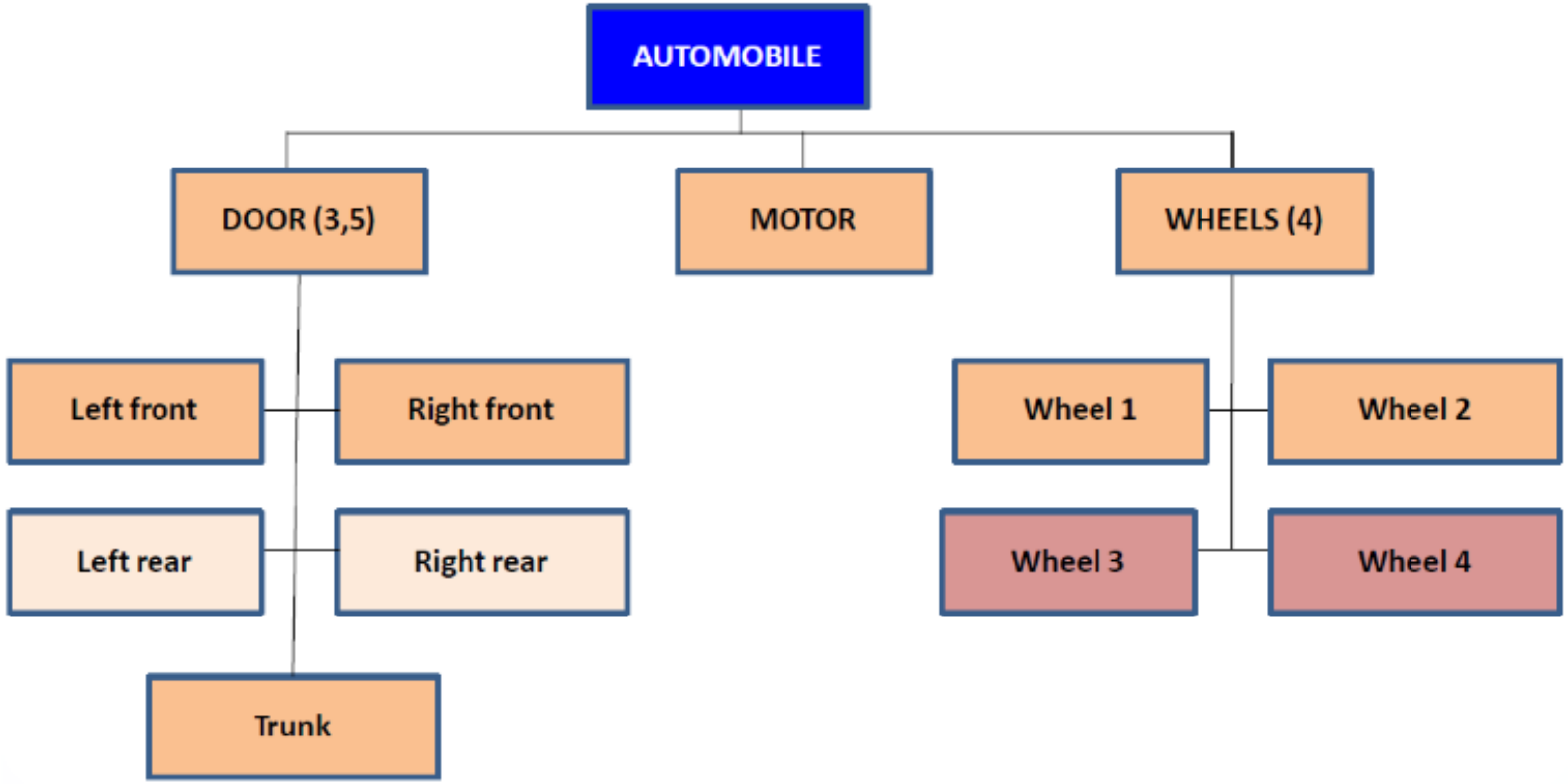
Data Nesting

Data Nesting in Java

- Data nesting in Java is a technique in which you store one or more data structures within another data structure. This can be useful for organizing and grouping related data.

- There are many different ways to nest data in Java. For example, you can nest arrays, lists, maps, and objects. You can also nest data structures within themselves.

DATA NESTING



```java
public class Automobile {
    private String make;
    private String model;
    private List<Door> doors;
    private Engine engine;
    private List<Wheel> wheels;

    // Constructor
    public Automobile(String make, String model, List<Door> doors, Engine engine, List<Wheel> wheels) {
        this.make = make;
        this.model = model;
        this.doors = doors;
        this.engine = engine;
        this.wheels = wheels;
    }

    // Getters and setters
    // ...

    // Nested classes
    public static class Door {
        private String position;
        private boolean open;

        // Constructor
        public Door(String position, boolean open) {
            this.position = position;
            this.open = open;
        }

        // Getters and setters
        // ...
    }

    public static class Engine {
        private int numCylinders;
        private String fuelType;

        // Constructor
        public Engine(int numCylinders, String fuelType) {
            this.numCylinders = numCylinders;
            this.fuelType = fuelType;
        }

        // Getters and setters
        // ...
    }

    public static class Wheel {
        private int size;
        private String type;

        // Constructor
        public Wheel(int size, String type) {
            this.size = size;
            this.type = type;
        }

        // Getters and setters
        // ...
    }
}
```

- As you can see, the Automobile class contains four nested classes: Door, Engine, and Wheel. This allows us to group all of the data related to these components together in one place.

- We can use this nested data structure to create new automobile objects. For example, the following code creates a new Automobile object with four doors, a four-cylinder engine, and four all-season tires:

```
List<Door> doors = new ArrayList<>();
doors.add(new Door("Left front", false));
doors.add(new Door("Right front", false));
doors.add(new Door("Left rear", false));
doors.add(new Door("Right rear", false));

Engine engine = new Engine(4, "Gasoline");

List<Wheel> wheels = new ArrayList<>();
wheels.add(new Wheel(16, "All-season"));
wheels.add(new Wheel(16, "All-season"));
wheels.add(new Wheel(16, "All-season"));
wheels.add(new Wheel(16, "All-season"));

Automobile automobile = new Automobile("Toyota", "Camry", doors, engine, wheels);
```

- Once we have created an Automobile object, we can access the data for each component using the getters and setters that we provided. For example, the following code prints the number of cylinders in the automobile's engine:

```
int numCylinders = automobile.getEngine().getNumCylinders();
System.out.println(numCylinders);
```

- Data nesting can be a very useful way to organize and manage complex data structures in Java. By nesting related data together, we can make our code more readable and maintainable.

Wrapper classes

Wrapper classes for primitive data types

- Wrapper classes in Java are used to convert primitive data types into objects. These objects come with methods that offer added functionalities.
- By using wrapper classes, you can go beyond the limitations of primitive data types, gaining access to a plethora of methods that make data manipulation more dynamic and efficient.

Primitive Data Type	Wrapper Class	Type of Parameter in the Constructor
byte	**Byte**	byte or String
short	**Short**	short or String
int	**Integer**	int or String
long	**Long**	long or String
float	**Float**	float, double or String
double	**Double**	double or String
char	**Character**	char
boolean	**Boolean**	boolean or String

Examples of Wrapper Classes

```
char character[]={'a','b','5','?','A',' '};

for(inti=0;i<character.length;i++) {
    if(Character.isDigit(character[i]))
        System.out.println(character[i]+"is a digit");
    if(Character.isLetter(character[i]))
        System.out.println(character[i]+"is a letter");
    if(Character.isWhitespace(character[i]))
        System.out.println(character[i]+"is a white space");
    if(Character.isLowerCase(character[i]))
        System.out.println(character[i]+"is a lower case letter");
    if(Character.isUpperCase(character[i]))
        System.out.println(character[i]+"is an upper case letter");
}

doubled=5.0;
Double aD = new Double(d); doubler=aD.doubleValue();
System.out.println(aD.toString()); System.out.println(r);
Integerage=newInteger(40);
System.out.println(age);
```

Casting Between Primitives and Wrapper Classes in Java

```java
//Boxing: Converting Primitive to Object. Java allows automatic conversion
//from a primitive to its corresponding object. This is termed as autoboxing.
Integer intObject;
int number = 42;
intObject = number;  // Automatic Boxing

//Unboxing: Converting Object to Primitive
Integer Number = Integer.valueOf("20");
int num1 = Number;  // Automatic Unboxing

//Explicit conversion using methods:
int num2 = intObject.intValue();  // Using intValue()

//Parsing a string to get its primitive value:
int num3 = Integer.parseInt("30");  // Parsing a string

//Retrieving maximum value for a data type:
float num4 = Float.MAX_VALUE;  // Maximum float value
```

Questions