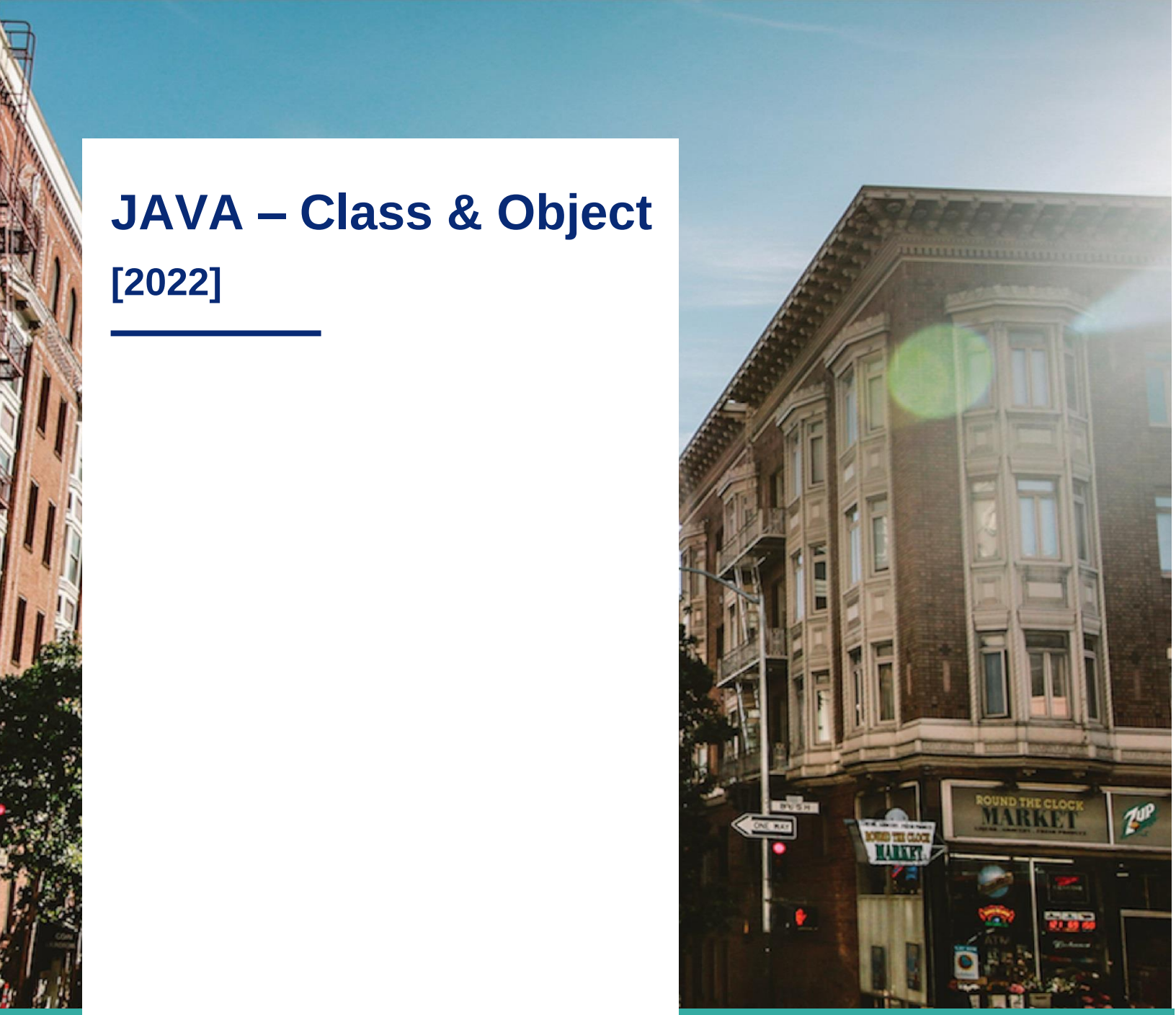


JAVA – Class & Object

[2022]

[29.08.2022]

Techno India College of Technology
Authored by: Abu Taha Md Farhad
Roll: 31401221052
Reg No: 213141001210016 OF 2021-22
Stream: BCA 2nd year
Subject: Object Oriented Programming [BCAC301]

Summary

Classes, objects, and methods are the basic components used in Java programming. The concept of classes is at the root of Java's design. We have discussed in detail the following in this chapter:

1. How to define a class
2. How to create objects
3. How to add methods to classes including constructors
4. How to extend or reuse a class
5. How to write application programs

We have also discussed various features that could be used to restrict the access to certain variables and methods from outside the class.

Introduction:

Java is a true object-oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a *class* that defines the state and behavior of the basic program components known as *objects*. Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

A class is essentially a description of how to make an object that contains fields and methods. It provides a sort of template for an object and behaves like a basic data type such as int. It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OOP concepts such as *encapsulation*, *inheritance*, *abstraction* and *polymorphism*.

Analysis:

Class

What is class?

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers:** A class can be public or has default access.
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body is surrounded by braces, {}.

Syntax:

```
public class Classname{
    // all methods and attributes
}
```

Example:

```
public class Rectangle {
    int length;
    int width;
    int area(){
        return (length * width);
    }
}
```

To initialize the class attributes, we have two ways to do that:

1. Using setData() method.
2. Using constructor method.

Initialize class attributes value using setData() method:

Example:

```
void setData(int l, int w) {
     length = l;
     width = w;
}
```

Initialize class attributes value using constructor method:

Constructor:

Java supports a special type of method, called a constructor, that enables an object to initialize itself when it is created. Constructor have the same name as the class itself. Secondly, they do not specify a return type, not even void. This is because they return the instance of the class itself.

Example:

```
Rectangle(int l, int w) {
     length = l;
     width = w;
}
```

Object:

What is object?

An object in java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as instantiating an object. Objects in java are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object.

Syntax:

```
Classname object_name; // declare the object
// instantiate the object
object_name = new ConstructorName();
```

Example:

```
Rectangle rect1; // declare the object
rect1 = new Rectangle (); // instantiate the object
```

Accessing class members:

Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. Remember, all variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the *dot operator*.

Syntax:

```
object_name.variableName = value;
object_name.methodName(parameter-list);
```

Example:

```
rect1.length = 15;
rect1.width = 10;
int area = rect1.area();
```

Difference between Class and Object:

Passing value while creating object using constructor:

Syntax:

```
Classname object_name = new ConstructorName(parameter-
list);
```

Example:

```
Rectangle rect2 = new Rectangle(15, 10);
```

Inheritance

How to extend or reuse a class?

Reusability is yet another aspect of OOP program. Java supports this concept. Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called *inheritance*. The old class is known as the base class or super class or parent class and the new one is called the subclass or derived class or child class. Inheritance may take different forms:

1. **Single inheritance** (only one super class).
2. **Multiple inheritance** (several super classes).
3. **Hierarchical inheritance** (one super class, many subclasses).
4. **Multilevel inheritance** (derived from a derived class).

Defining a Subclass:

If the super class is already existing then the subclasses can inherit all the attributes and methods. A subclass is defined as follows:

Syntax:

```
class subclassname extends superclassname {
     variables declaration;
     methods declaration;
}
```

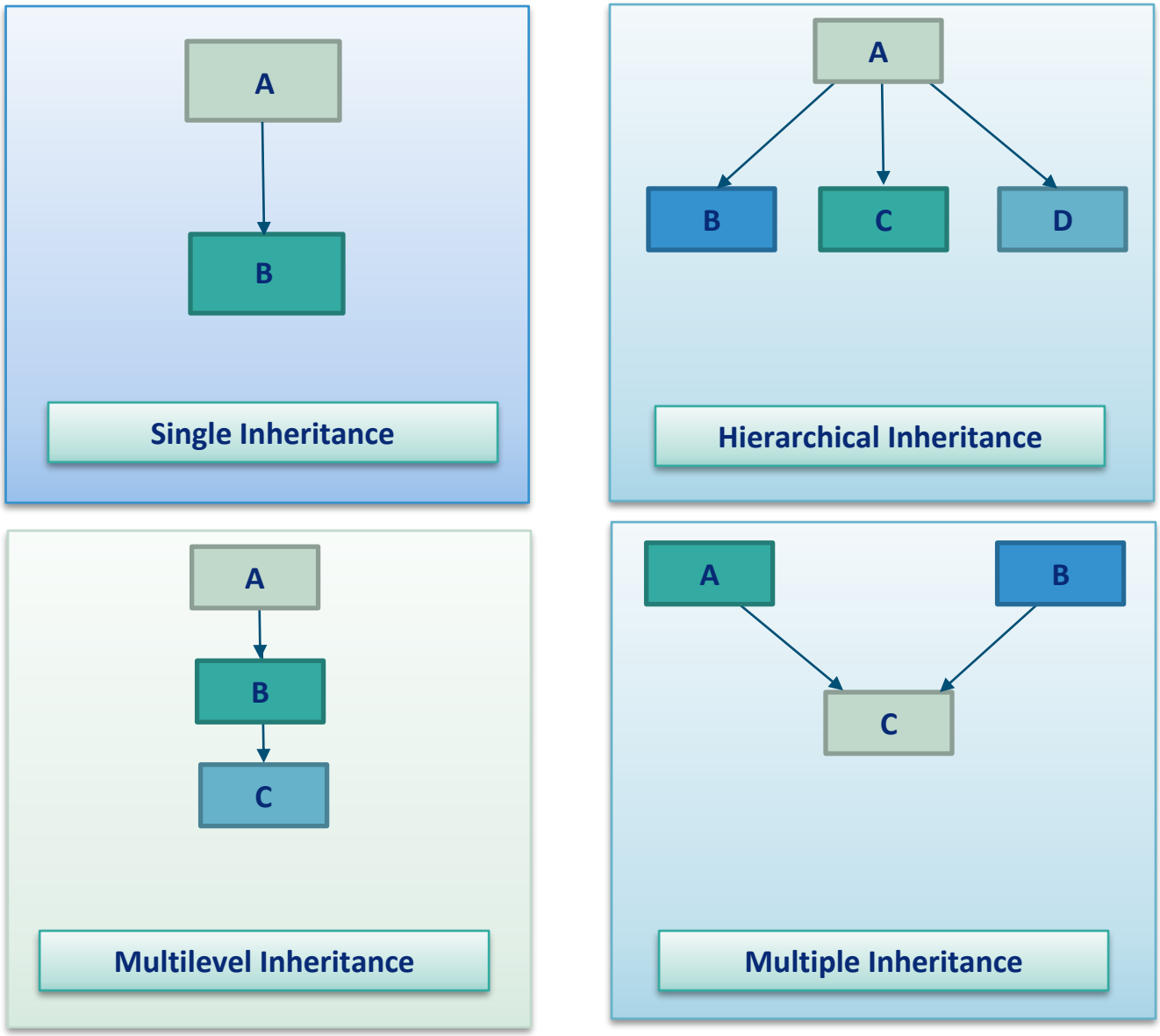
Example:

```
//super class
class Fruit {
    float sweetness;
    int[] color;
    void setColor(int r, int g, int b) {
        color = new int[]{r, g, b};
    }
    Fruit(float s, int[] c) {
        sweetness = s;
        setColor(c[0], c[1], c[2]);
    }
}

//subclass
class Mango extends Fruit {
    String name;
    Mango(float x, int[] y, String z) {
        super(x, y);
        name = z;
    }
    void display (){
        System.out.println("Name: " + name);
        System.out.println("Sweetness: " + sweetness);
        System.out.print("Color: R=" + color[0] + ", G="
                + color[1] + ", B=" + color[2]);
    }
}
```

The above program is the example of *Single Inheritance*. Here Fruit is the super class and inherited by a subclass named Mango. If we create an object of the Mango class then all the attributes and methods of the Fruit can be accessed from the Mango class.

Structures of different inheritance:



Final classes:

What are final classes?

Sometimes we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called a *final class*. This is achieved in Java using the keyword **final** as follows:

Example:

```
final class Aclass { . . . }
class Bclass extends Aclass { . . . } // error
```

Abstract classes:

What are abstract classes?

Abstract classes are the type of classes which contents abstract methods in it. Abstract methods are the methods whose definition is overridden by the derived classes. The abstract classes are exactly opposite to the Final classes. By the abstract methods we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory. This can be done by using the modifier keyword **abstract** in the method definition.

Syntax:

```
abstract class Classname {
     abstract return-type method_name();
}
```

Example:

```
abstract class Sayhello {
    abstract void greet();
}
class India extends Sayhello {
    void greet() {
        System.out.println("India: Namaste");
    }
}
class Spain extends Sayhello {
    void greet() {
        System.out.println("Spain: Hola");
    }
}
```

Visibility control:

How to accomplish data hiding?

It is possible to inherit all the members of a class by a subclass using the keyword extends. We have also seen that the variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. for example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in java by applying *visibility modifiers* to the instance variable and methods. the visibility modifiers are also known as *access modifiers*. Java provides three types of visibility modifiers:

1. **Public:**
   - The code is accessible for all classes.
2. **Private:**
   - The code is only accessible within the declared class.
3. **Protected:**
   - The code is accessible in the same package and subclasses.

Syntax:

```
access_modifier data-type variableName;
access_modifier data-type methodName() { . . . }
access_modifier class Classname { . . . }
```

Example:

Fathersproperty.java

```
package pca21;
public class Fathersproperty {
    public void letterBox() { . . . }
    private void locker(){ . . .}
    protected void bedroom() { . . . }
```

```
    public static void main(String[] args) {
        Fathersproperty father = new Fathersproperty();
        System.out.println("Father can access:");
        father.letterBox();
        father.bedroom();
        father.locker();
    }
}
```

Son.java

```
package pca21;
public class Son extends Fathersproperty{
    public static void main(String[] args) {
        Son son = new Son();

        System.out.println("Son can access:");
        son.letterBox();
        son.bedroom();
        // son.locker();
    }
}
```

Someone.java

```
package pca22;
import ca2.Fathersproperty;

public class Someone {
    public static void main(String[] args) {
        Fathersproperty f = new Fathersproperty();
        System.out.println("Others can access:");
        f.letterBox();
        // f.bedroom();
        // f.locker();
    }
}
```

If we try to access all properties of Fathersproperty class from the class itself, we can do that. Although the Son class inherits all the properties of the Fathersproperty class, still we cannot access the locker() method since it is private but we can access the bedroom() method as it is protected and protected properties can be accessed anywhere in the same package. From the Someone class except the letterBox() method we cannot access either of the methods bedroom() or locker() of the Fathersproperty class. Because, letterbox() is public, that means it can be accessed throughout the program but locker() is private, which can only be accessed in the class itself where it is declared and since Someone class belongs from another package it cannot access the protected bedroom() method.

Implementation program:

The following program is to implement the concept of inheritance:

```
package ca2;

//super class
class Fruit {
    float sweetness;
    int[] color;
    void setColor(int r, int g, int b) {
        color = new int[]{r, g, b};
    }
    Fruit(float s, int[] c) {
        sweetness = s;
        setColor(c[0], c[1], c[2]);
    }
}

//subclass
class Mango extends Fruit {
    String name;
    Mango(float x, int[] y, String z) {
        super(x, y);
        name = z;
    }
    void display(){
        System.out.println("Name: " + name);
        System.out.println("Sweetness: " + sweetness);
        System.out.print("Color: R=" + color[0] + ", G="
                + color[1] + ", B=" + color[2]);

    }
}

public class Fruittest {
    public static void main(String[] args) {
```

```
        Mango m1 = new Mango (9.8f, new int[]{255, 140, 0}, "Amrapali Mango");
        m1.display();
    }
}
```

Output:

```
Name: Amrapali Mango
Sweetness: 9.8
Color: R=255, G=140, B=0
```

The following program is to implement the concept of abstraction:

```
package ca2;

abstract class Sayhello {
    abstract void greet();
}

class India extends Sayhello {
    @Override
    void greet() {
        System.out.println("India: Namaste");
    }
}
class Spain extends Sayhello {
    @Override
    void greet() {
        System.out.println("Spain: Hola");
    }
}

public class Sayhellotest {
    public static void main(String[] args) {
        India i = new India();
        Spain s = new Spain();
        i.greet();
        s.greet();
    }
}
```

Output:

```
India: Namaste
Spain: Hola
```

The following program is to implement the concept of encapsulation:

Fathersproperty.java

```
package pca21;

public class Fathersproperty {
    public void letterBox() {
        System.out.println("Letter box: " +
                "Accessible to all.");
    }
    private void locker(){
        System.out.println("Locker: " +
                "Only father can access.");
    }
    protected void bedroom() {
        System.out.println("Bedroom: Only accessible to " +
                "father's family members.");
    }

    public static void main(String[] args) {
        Fathersproperty father = new Fathersproperty();

        System.out.println("Father can access:");
        father.letterBox();
        father.bedroom();
        father.locker();
    }

}
```

Output:

```
Father can access:
Letter box: Accessible to all.
Bedroom: Only accessible to father's family members.
Locker: Only father can access.
```

Son.java

```
package pca21;

public class Son extends Fathersproperty{
    public static void main(String[] args) {
        Son son = new Son();

        System.out.println("Son can access:");
        son.letterBox();
        son.bedroom();
        // son.locker();
    }
}
```

Output:

```
Son can access:
Letter box: Accessible to all.
Bedroom: Only accessible to father's family members.
```

Someone.java

```
package pca22;

import ca2.Fathersproperty;

public class Someone {
    public static void main(String[] args) {
        Fathersproperty f = new Fathersproperty();

        System.out.println("Others can access:");
        f.letterBox();
        // f.bedroom();
        // f.locker();
    }
}
```

Output:

```
Others can access:
Letter box: Accessible to all.
```

Conclusions:

Benefits:

Why we use object-oriented programming?

1. It is easy to partition the work in a project based on objects.
2. Object-oriented systems can be easily upgraded from small to large systems.
3. Software complexity can be easily managed.
4. The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
5. Through inheritance, we can eliminate redundant code and extend the use of existing classes.

Applications:

Where we can use object-oriented programming?

1. Real-time systems
2. Simulation and modelling
3. Object-oriented databases
4. Hypertext, hypermedia
5. Al and expert systems

References:

Reference list:

Programming with Java: A primer, 5e | E Balagurusamy, former Vice Chancellor, Anna University

Refsnes Data, “Java Tutorial”, w3schools, https://www.w3schools.com/java/default.asp, Retrieved: 23-08-2022

“Java OOP”, W3schools, https://www.w3schools.com/java/java_oop.asp, Retrieved: 23-08-2022

“Java Classes and Objects”, W3schools, https://www.w3schools.com/java/java_classes.asp, Retrieved: 23-08-2022

Thank You