Coding Cheat Sheet

This reading provides a reference list of code that you'll encounter as you work with object-oriented coding in Java. Understanding these concepts will help you write and debug object-oriented Java programs. Let's explore the following Java coding concepts:

- Working with classes and objects
- Working with access and non-access modifiers
- Using encapsulation
- Using constructors

Keep this summary reading available as a reference as you progress through your course, and refer to it when you begin coding object-oriented Java programming after this course!

Working with classes and objects

Creating a class

Description	Example
Create a `Car` class, which serves as a blueprint for creating Car objects.	`public class Car {`

Define attributes of the `Car` class. The variables `color`, `model`, and `year` store the car's color, model, and year, respectively.	`String color;`
	`String model;`
	`int year;`

Include the method `displayInfo()` to print car objects.	`void displayInfo() {`

Print the car details to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model);`
	`System.out.println("Car Color: " + color);`
	`System.out.println(>System.out.println("Car Year: " + year);`

Close curly braces to end the `Car` class definition.	`}` `}`

Explanation: This example creates a class named `Car` and defines three attributes for the `Car` class: `model`, `color`, and `year`. The `displayInfo()` method prints the car details.

Creating an object

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`

Description	Example
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an object of the `Car` class.	`Car myCar = new Car();`

Assign values to the object's attributes.	`myCar.color = "Red";`
	`myCar.model = "Toyota";`
	`myCar.year = 2020;`

Call the `displayInfo()` method to print the object details.	`myCar.displayInfo();`
Close curly braces to end the `main` method and class definition.	`}` `}`

Explanation: This example declares a reference variable named `myCar` of type `Car`. `new Car()` creates a new object of the `Car` class and assigns values to the object's attributes: `color`, `model` and `year`. The `displayInfo()` method prints the car details.

Working with access and non-access modifiers

Public access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning it can be accessed directly from outside the class.	`public class Car {`

Description	Example
A Java statement to declare a `String` variable named `model` to store the car's model name.	`public String model;`
Close curly braces to end the class definition.	`}`

Private access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a private `String` variable named `color` to store the car's color. The private modifier ensures the `color` variable can be accessed and modified only within the `Car` class.	`private String color;`
Call the `displayColor()` method with the private access modifier. This ensures the method can be called only within the `Car` class and not from other classes.	`private void displayColor() {`
Print the car's color to the console using the `System.out.println()` function.	`System.out.println("Car Color: " + color);`
Close curly braces to end the class definition.	`}` `}`

Protected access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a protected `int` variable named `year` to store the car's year. The protected modifier ensures the `year` variable is accessible within the same package (default package access) and by subclasses, even if they are in different packages.	`private String year;`
Call the `displayYear()` method with the protected access modifier. This ensures the method can be called within the same package and by subclasses, even if they are in different packages.	`private void displayYear() {`
Print the car's year to the console using the `System.out.println()` function.	`System.out.println("Car Year: " + year);`
Close curly braces to end the class definition.	`}` `}`

Default access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects.	`class Car {`
A Java statement to declare a `String` variable named `model` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String model;`
Call the `displayModel()` method without any access modifier.	`void displayModel() {`

Description	Example
Print the car's model to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model);`
Close curly braces to end the class definition.	`}` `}`

Static non-access modifier

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects. The variable model is declared as `public`, meaning that it can be accessed directly from outside the class.	`public class Car {`
A Java statement to declare a static `int` variable named `numberOfCars` to keep track of the total number of `Car` objects created. Since it's static, its value is shared among all instances of `Car`.	`static int numberOfCars = 0;`
A Java statement to declare a constructor. Every time a new `Car` object is created, this constructor runs.	`public Car() {`
A Java statement to increment the `numberOfCars` variable that keeps track of how many cars have been instantiated.	`numberOfCars++;`
Close curly braces to end the class definition.	`}`

Description	Example
Call the `displayCount()` method without creating an instance of the `Car` class. This method can only access static variables like `numberOfCars`, not instance variables.	`private void displayCount() {`
Print the total number of cars to the console using the `System.out.println()` function.	`System.out.println("Total Cars: " + numberOfCars);`
Close curly braces to end the class definition.	`}` `}`

Final non-access modifier

Description	Example
A Java statement used to define a final class named `Vehicle`, which acts as a blueprint for creating `Car` objects. The final class cannot be extended (inherited) by any other class. This means no subclasses can be created from `Vehicle`.	`public final class Vehicle {`
A Java statement to declare a final `int` variable named `maxSpeed` with the value 120. The final variable is a constant, meaning that its value cannot be changed once it is assigned. Trying to modify `maxSpeed` later in the code will cause a compilation error.	`final int maxSpeed = 120;`
A Java statement to declare a final method named `displayMaxSpeed()`. The final method cannot be overridden by subclasses. This ensures the behavior of `displayMaxSpeed` remains the same in all instances.	`final void displayMaxSpeed() {`
Print the maximum car speed to the console using the `System.out.println()` function.	`System.out.println("Max Speed: " + maxSpeed);`
Close curly braces to end the class definition.	`}` `}`

Description	Example

Abstract non-access modifier

Description	Example
A Java statement used to define an abstract class named `Shape`. This is an abstract class, meaning that it cannot be instantiated (you cannot create `Shape` objects directly). It works as a blueprint from which other classes can inherit.	`public abstract class Shape {`
A Java statement used to define a final class named `Vehicle`, which acts as a blueprint for creating `Car` objects. The final class cannot be extended (inherited) by any other class. This means no subclasses can be created from `Vehicle`.	`abstract void draw();`
Close curly braces to end the class definition.	`}`
A Java statement to describe `Circle` that extends the `Shape` class and provides an implementation of the `draw()` method.	`public class Circle extends Shape {`
A Java annotation to tell the compiler the draw() method in `Circle` is an override of the abstract method in `Shape`.	`@Override`
A Java statement saying the `draw` method is now fully implemented.	`void draw()`
Print the string `Drawing Circle` to the console using the `System.out.println()` function.	`System.out.println("Drawing Circle");`

Description	Example
Close curly braces to end the class definition.	} } }

Using encapsulation

Creating an encapsulated class

Description	Example
Create the `Person` class, which serves as a blueprint for creating Person objects.	class Person {
Create private attributes `name` and `age` to store the person's name and age. The `name` and `age` attributes cannot be accesse diretly from outside the class.	private String name; private int age;
Use the Java constructor to initialize the `name` and `age` variables when a `Person` object is created.	public Person(String name, int age) {
The keyword `this` refers to the current object's instance variables. It differentiates instance variables from method parameters.	this.name = name; this.age = age;
Close curly braces to end the class definition.	}
Use the Java public method (Getter) to obtain read access to private variables.	public String getName() {

Description	Example
`getName()` returns the value of `name`.	`return name;`
Close curly braces to end the class definition.	`}`
Use the Java public method (Setter) to obtain write access to private variables.	`public void setName(String name) {`
`setName()` updates `name`.	`this.name = name;`
Use the Java public method (Getter) to obtain read access to private variables.	`public int getAge() {`
`getAge()` returns the value of `age`.	`return age;`
Close curly braces to end the class definition.	`}`
Use the Java public method (Setter) to obtain write access to private variables.	`public void setAge(int age) {`
Use the Java `if` statement to ensure `age` is not negative before assigning.	`if (age >= 0) {`

Description	Example
Update the `age` variable.	`this.age = age;`
Use the Java `else` statement to specify what to do when the age is negative.	`} else {`
Print the string `Age cannot be negative` to the console using the `System.out.println()` function.	`System.out.println("Age cannot be negative.");`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates a `Person` class in which the `name` and `age` attributes are declared as private, meaning they cannot be accessed directly from outside the `Person` class. The constructor `Person(String name, int age)` initializes the attributes when a new object of the class is created. `getName()` and `getAge()` are getter methods that allow other classes to read the values of `name` and `age`. `setName(String name)` and `setAge(int age)` are setter methods that allow other classes to modify the values of `name` and `age`. The setter for `age` includes validation to ensure `age` cannot be set to a negative number.

Using an encapsulated class

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create a new instance of the `Person` class. Assign the value "Alice" to the `name` attribute and the value "30" to the `age` attribute.	`Person person = new Person("Alice", 30);`

Description	Example
Use the `getName()` getter to obtain and print the value of the `name` attribute.	`System.out.println("Name: " + person.getName());`
Use the `getAge()` getter to obtain and print the value of the `age` attribute.	`System.out.println("Age: " + person.getAge());`
Use the `setName()` setter to assign the value of `name` attribute to "Bob" and `age` attribute to "25".	`person.setName("Bob");` `person.setAge(25);`
Use the `getName()` getter to obtain and print the updated value of the `name` attribute.	`System.out.println("Updated Name: " + person.getName());`
The `setAge(-5)` call attempts to set an invalid age. Since `setAge()` has validation logic, it will print "Age cannot be negative."	`System.out.println("Updated Age: " + person.getAge());`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Person` class with the `name` "Alice" and `age` "30". We call the `getName()` and `getAge()` getter methods to print the values. We then update the `name` and `age` attributes usint the `setName()` and `setAge()` setter methods. When we attempt to set a negative age with setAge(-5), it prints an error message because of validation included in the setter method.

Using constructors

Creating a default constructor

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;`
This is the default constructor. It takes no arguments.	`Dog() {`
The default constructor initializes the `name` variable with the value "Unknown". This ensures every new `Dog` object always hasa name, even if the user doesn't provide one.	`name = "Unknown";`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's name to the console using the `System.out.println()` function. Since `name` was initialized in the constructor, it always has a value.	`System.out.println("Dog's name: " + name);`
Close curly braces to end the class definition.	`}` `}`

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `Dog` class using the default constructor. The `name` variable is automatically set to "Unknown".	`Dog myDog = new Dog();`
Call the `display()` method to print the dog's name.	`myDog.display();`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Dog` class with a default constructor that initializes the `name` attribute to "Unknown". When we create the instance, the default constructor is invoked automatically.

Creating a parameterized constructor

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;`
This is the parameterized constructor that takes one argument `dogName`.	`Dog(String dogName) {`

Description	Example
When the `Dog` object is created, the provided dogName value is assigned to the `name` variable. Parameterized constructors let you assign a unique name to each `Dog` object when it is created.	`name = dogName;`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's name to the console using the `System.out.println()` function. Since `name` was initialized in the constructor, it always has a value.	`System.out.println("Dog's name: " + name);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `Dog` class. "Buddy" is passed as an argument to the constructor, setting name to "Buddy".	`Dog myDog = new Dog("Buddy");`

Description	Example
Call the `display()` method to print the dog's name.	`myDog.display();`
Close curly braces to end the class definition.	`}` `}`

Explanation: This example creates an instance of the `Dog` class with a parameterized constructor that takes a `string` parameter `dogName`. When we create a `Dog` instance with the name "Buddy", the constructor initializes the name attribute with that value.

Creating a no-arg constructor

Description	Example
A Java statement used to define a class named `Car`, which acts as a blueprint for creating `Car` objects.	`class Car {`
A Java statement to declare a `String` variable named `model` and an `int` variable named `year` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String model;` `int year;`
This is a no-argument constructor that takes no parameters.	`Car() {`
When the `Car` object is created, it automatically assigns the value "Default Model" to `model` and 2020 to `year`.	`model = "Default Model";` `year = 2020;`
Close curly braces to end the class definition.	`}`

Description	Example
Call the `display()` method without any access modifier.	`void display() {`
Print the car's model and year to the console using the `System.out.println()` function.	`System.out.println("Car Model: " + model + ", Year: " + year);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create an instance of the `Car` class. The no-argument constructor is called, setting `model` = "Default Model" and `year` = 2020.	`Car myCar = new Car();`
Call the `display()` method to print the model and year of the car.	`myCar.display();`
Close curly braces to end the class definition.	`}` `}`

Description	Example

Explanation: This example creates an instance of the `Car` class with two attributes `model` and `year`. The Car() constructor initializes the model to "Default Model" and year to 2020. When we create an instance of the `Car` class with `new Car()`, the no-arg constructor is called automatically, and the default values are assigned to the attributes. The `display()` method prints the `model` and `year` of the car.

Constructor overloading

Description	Example
A Java statement used to define a class named `Dog`, which acts as a blueprint for creating `Dog` objects.	`class Dog {`
A Java statement to declare a `String` variable named `name` and an `int` variable named `age` without any access modifier. If no access modifier is used, the variable is considered default. Default variables are accessible only within their own package.	`String name;` `int age;`
This is the default constructor. It takes no arguments.	`Dog() {`
The default constructor initializes the `name` variable with the value "Unknown" and `age` variable with the value 0. This ensures every new `Dog` object always has a name and age, even if the user doesn't provide one.	`name = "Unknown";` `age = 0;`
Close curly braces to end the class definition.	`}`
This is the parameterized constructor that takes one argument `dogName`.	`Dog(String dogName) {`
When the `Dog` object is created, the provided dogName value is assigned to `name` while keeping the `age` as 0 by default. Parameterized constructors let you assign a unique name to each `Dog` object when it is created.	`name = dogName;` `age = 0;`

Description	Example
Close curly braces to end the class definition.	`}`
This is the parameterized constructor that takes two arguments `dogName` and `dogAge`.	`Dog(String dogName, int dogAge) {`
When the `Dog` object is created, the constructor allows the user to specify both `name` and `age`.	`name = dogName;` `age = dogAge;`
Close curly braces to end the class definition.	`}`
Call the `display()` method without any access modifier.	`void display() {`
Print the dog's `name` and `age` to the console using the `System.out.println()` function. Since `name` and `age` were initialized in the constructor, they always have a value.	`System.out.println("Dog's name: " + name + ", Age: " + age);`
Close curly braces to end the class definition.	`}` `}`
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`

Description	Example
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
Create the `dog1` object using the default constructor `Dog()`. So, name = "Unknown" and age = 0.	`Dog dog1 = new Dog();`
Create the `dog2` object using the one-parameter constructor `Dog("Charlie")`. So, name = "Charlie" and age = 0 (default).	`Dog dog2 = new Dog();`
Create the `dog3` object using the two-parameter constructor `Dog("Max", 5)`. So, name = "Max" and age = 5.	`Dog dog3 = new Dog();`

Call the `display()` method on each object to print their details.	`dog1.display();`
	`dog2.display();`
	`dog3.display();`

Close curly braces to end the class definition.	`}` `}`

Explanation: This example has three constructors of the `Dog` class. Depending on the number of parameters provided when creating an object, the corresponding constructor is called.

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan