Java OOP Cheatsheet

(Tested with Java 11)

The Java OOP Cheatsheet is designed in an easy to understand flow with examples to learn the OOP (Object-Oriented Programming) concepts of Java. It can also be considered as a guide to Getting Started with OOP in Java. It explains the basic usage of abstraction, encapsulation, inheritance, and polymorphism with examples. At the end, it provides the link to download the PDF having all the sections of the Java OOP Cheat Sheet.

0 - Installers

Java 11 On macOS, Java 11 On Windows, Java 11 On Ubuntu, Java 8 On macOS, Java 8 On Windows, Java 8 On Ubuntu, IntelliJ IDEA on Windows, IntelliJ IDEA on Ubuntu, Visual Studio Code on Windows, Visual Studio Code on Ubuntu, Eclipse on Windows, Eclipse on Ubuntu, Eclipse on Mac, and NetBeans on Windows

1 - Basics

You may follow the Java Basics Cheatsheet to learn the basics of Java.

2 - Modifiers

```
# Access Modifiers

# There are three access modifiers in Java including public, private, and protected.
# Apart from these three modifiers, a class or member can have default visibility without specifying any access modifier.
# public Visible within the same or other packages.
# protected Visible within the same package or child classes in same or other packages.
# private Visible within the same class.
# none Visible within the same package.

# Non-Access Modifiers

# Apart from the three access modifiers, there are non-access modifiers including abstract, final, strictfp, default, static, native, synchronized, transient, and volatile.
# abstract Applicable on classes, interfaces, and methods.
# final Applicable on classes, methods, instance variables, local variables, class variables, and interface variables.
# strictfp Applicable on classes, and methods. The floating points in classes or methods marked as strictfp must adhere to IEEE 754 standard.
# default Applicable on interface methods with body since Java 8.
```

static Applicable on nested classes, methods, class variables, and interface variables. Also, applicable on interface methods since Java 8.
native Applicable on methods to call platform dependent method.
synchronized Applicable on methods and method blocks. The synchronized methods and blocks can be accessed by one thread at a time.
transient The instance variables marked as transient won't be serialized.
volatile The instance variables marked as volatile will force threads to occupy their own copy of the variable with the master copy in memory.

Access Modifiers Scope

Scope	Private	Default	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package	No	Yes	Yes	Yes
Different Package Sub Class	No	No	Yes	Yes
Different Package Non-Sub Class	No	No	No	Yes

3 - Classes

A Class can be declared using the keyword class.
A class can be considered as blueprint or prototype to create the instances of the class.
Instances of a class are also known as objects.
We can create multiple objects of the same class using the keyword new.
Every Object has it's own state and behavior in the form of instances fields and methods respectively.
The classes marked as public are visible within other packages.
The classes without any access modifier are visible within the same package.
The class declaration cannot use the private or protected access modifier.
The nested class declaration can use the public, private or protected access modifier.
The nested class declaration can use the static modifier.
The file name of the file having public class must be the same as that of the class.
The source code file can have any name in absence of public class. The name must not match any class name of the classes declared within the file.
A file can have only one public class.
A file can have multiple non-public classes.
If the class is part of a package, the package statement must be the first line.
If there are import statements, they must go between the package and first class in the same source code file.

```
# In absence of package statement, import statement must be the first line.
# In absence of package and import statements, class declaration must be the
first line.
# It's preferred to follow Camel Case naming convention while naming the
method. The first letter can be capital and capital letter can be used for
the inner words.

# Class Examples

// Package Level
class Vehicle { }

public class Vehicle { }

public class Car { }

public class Truck { }

// Create Objects
Car car = new Car();
Truck truck = new Truck();
```

4 - Constructors

```
# The Java Compiler creates a default no-args constructor in case no
constructor is defined for a class.
# The default no-args constructor has the same access modifier as that of the
class.
# The default no-args constructor includes call to the superclass no-args
constructor using super().
# We can explicitly declare and define the no-args constructor.
# The constructor name must be the same as that of the class.
# The constructor must not be associated with return type.
# A class can have multiple overloaded constructors.
# The class objects can be created using the appropriate constructor.
# Every constructor, as it's first statement, implicitly invokes no-args
constructor of the superclass by calling super().
# We can explicitly invokes no-args constructor or overloaded constructor
with arguments of the same class by calling this() or superclass by calling
super() as the first statement of the constructor.
# In absence of no-args constructor and presence of constructor with
arguments, the compiler won't create the default no-args constructor.
# In absence of no-args constructor and presence of constructor with
arguments in superclass, the constructor must explicitly invokes no-args or
overloaded constructor of same class by calling this() or overloaded
constructor of superclass by calling super().
```

```
# In absence of no-args constructor and presence of constructor with
arguments, we cannot create an object without passing appropriate constructor
arguments.
# It's preferred to define a no-args constructor in presence of constructors
with arguments. Though, in several scenarios we might not be required to
define the no-args constructor.
# Constructors can use any access modifiers including public, protected,
private, or none.
# A constructor cannot be invoked like a method. A constructor can be invoked
within another constructor by calling this() or super() as the first
statement.
```

 # Constructor Examples

```
public class Car {

        private String color;

        // Constructor without any argument and without return type
        public Car() {

                this.color = "White";
        }

        // Constructor with argument and without return type
        public Car( String color ) {

                this.color = color;
        }
}

// Create Objects
Car whiteCar = new Car();
Car redCar = new Car( "Red" );
```

5 - Methods

```
# A Method is a block of code defining the behavior of the class.
# A method must always specify the return type.
# A class can have multiple methods with different names.
# A class can have multiple overloaded methods having same name, but
different parameters.
# Though it's not preferred, the method name can be the same as that of the
class with return type.
# A method can be either instance method or class method.
# The class methods must use the keyword static.
```

```
# The static methods can be called without creating the object of the class.
# The static methods can't be overridden.
# The static methods can't call non-static method or use instance variables.
# The static methods can call static methods or use static variables.
# The methods without any access modifiers remains visible within the same package.
# The methods with public access modifier remains visible within the same and other packages.
# The methods with protected access modifier remains visible within the same package and child classes in the same package or other packages.
# It's preferred to follow Camel Case naming convention while naming the method. The first letter can be small and capital letter can be used for the inner words.
```

 # Method Examples

```
public class Car {

        private String color;

        public Car() {

                this.color = "White";
        }

        public String getColor() {

                return color;
        }

        public void setColor( String color ) {

                this.color = color;
        }
}

// Create Objects
Car myCar = new Car();

// Print Color
System.out.println( myCar.getColor() );
```

6 - Initialization Blocks

```
# Apart from constructors and methods, a class can perform operations using initialization blocks.
```

```
# The static initialization block runs only once when the class is loaded.
# The instance initialization block runs when the instance is created.
# The instance initialization block runs after the call to super() is over.
# A class can have multiple initialization blocks.
# The initialization blocks runs in the same order as they appear in the
source code file.

# Initialization Block Examples

public class Car {

        static int count;

        static {

                count = 1000; // Start counting from 1001
        }

        public Car() {

                count++;
        }
}

// Print Count
System.out.println( Car.count );

// Output
1001
```

7 - Inheritance

```
# A class can be sub-classed using the keyword extends.
# A class cannot extend more than one class.
# A class marked as final cannot be sub-classed.
# We can test the object type using the keyword instanceof.
# We can perform Is-A relationship test to identify the type of object.
# A constructor cannot be inherited.
# The child class inherits all the public and protected members of parent
classes.
# The child class can override parent class methods.
# The child class can overload parent class methods.
# Inheritance makes it possible to re-use the code by extending existing
classes without re-implementing the same logic.
# We can take advantage of polymorphism by using inheritance.
```

```
# Inheritance Examples

// Package Level
public class Vehicle { }

public class Car extends Vehicle { }

public class Truck extends Vehicle { }

// Create Objects
Car car = new Car();
Truck truck = new Truck();
Vehicle myCar = new Car();

# IS-A relationship tests

// car Is-A Car
// car Is-A Vehicle
// truck Is-A Vehicle
// myCar Is-A Car
// myCar Is-A Vehicle
```

8 - Abstraction

Abstraction is the process to hide the implementation details and showing only the essential information to the user. Abstraction in Java can be achieved by using abstract classes and interfaces.

9 - Abstract Classes

```
# A class can be declared as abstract by using the abstract keyword.
# Abstract classes cannot be marked as final.
# Abstract classes cannot be instantiated i.e. objects cannot be created
using the new keyword.
# An abstract class must be sub-classed to create the objects.
# Methods without the implementation must be marked as abstract method using
the abstract keyword.
# Abstract methods must end with semi-colon without the method body.
# The class must be abstract if it contains even a single abstract method.
# An abstract class can have zero to several abstract methods.
# An abstract class may implement the abstract methods of parent classes.
# The concrete class must implement all the abstract methods of parent
classes.
# An abstract class may implement the abstract methods of the interfaces
implemented by it.
```

```
# The concrete class must implement all the abstract methods of the
interfaces implemented by it.
# The concrete class may leave the implementation of abstract methods if
already implemented by the parent classes.
```

 # Abstract Class Examples

```
public abstract class Vehicle { }

public abstract class Vehicle {

        public abstract String getName();
}
```

10 - Interfaces

```
# An Interface can be declared using the keyword interface.
# An interface is implicitly abstract.
# An interface can be either package level with default visibility or marked
as public.
# An interface can be implemented by any class.
# The class implementing the interface must be marked as abstract if it does
not implement all the abstract methods of the interface.
# An interface can extend multiple interfaces.
# An interface cannot implement another interface.
# An interface cannot extend class.
# All the interface methods are implicitly public and abstract unless
declared as static or default.
# All the variables defined in the interface are implicitly public, static
and final.
# Interface methods cannot be marked as final, strictfp, or native.
# An interface cannot have constructor.
```

 # Interface Examples

```
public abstract interface Bounceable { }

public interface Bounceable { }

// Package Level
interface Bounceable { }

public interface Bounceable {

        public void bounce();
```

```
}
```

11 - Encapsulation

```
# Hide implementation details behind a public interface or methods so that
the implementation can be changed without breaking the code.
# Variables of a class remains hidden from other classes using the private or
protected access modifier.
# The private or protected variables can be accessed by other classes using
the getter and setter methods.
# The getter and setter can implement logic before returning the variable or
updating the variable.
# The class controls the variable values.
# The variables of a class can be made either read-only or write-only by
omitting setter or getter.
# A full-encapsulated class can declare all it's variables private.
# Encapsulation is a way to hide data by denying direct access to the
variables.
# To use encapsulation for reference variables, the getter methods must
return reference to the copy of the object.
```

 # Encapsulation Examples

```
public class Vehicle {

        private String name;
        private String color;

        public String getName() {

                return this.name;
        }

        public void setName( String name ) {

                this.name = name;
        }

        public String getColor() {

                return this.color;
        }

        public void setColor( String color ) {

                this.color = color;
```

```
        }
}

Vehicle myVehicle = new Vehicle();

myVehicle.setName( "Car" );
myVehicle.setColor( "Red" );

System.out.println( "Name:" + myVehicle.getName() );
System.out.println( "Color:" + myVehicle.getColor() );
```

12 - Polymorphism

```
# Polymorphism means many forms.
# Same action can be performed in different ways.
# Any Java object which can pass more than one IS-A test can be considered polymorphic. All Java objects are polymorphic except the objects of class Object.
# Upcasting - Reference variable of parent class refers to object of child class. Upcasting is implicit.
# Downcasting - Reference variable of child class explicitly refers to reference variable of parent class which further refers to object of child class.
# Types of polymorphism are compile-time polymorphism and runtime polymorphism.
# The Compile-Time Polymorphism can be achieved by method overloading. It is also know as Static Polymorphism.
# The Runtime Polymorphism can be achieved by method overriding. It is also know as Dynamic Method Dispatch.

# Polymorphism Examples

public interface Injectable {

        public boolean inject( int quantity );

        public boolean inject( int quantity, int depth );
}

public class Animal implements Injectable {

        private String name;

        protected int maxQuantity;
        protected int minQuantity;
```

```
        protected int maxDepth;
        protected int minDepth;

        protected boolean injected;

        public Animal( String name, int maxQuantity, int minQuantity ) {

                this.name = name;
                this.maxQuantity = maxQuantity;
                this.minQuantity = minQuantity;
        }

        public Animal( String name, int maxQuantity, int minQuantity, int
maxDepth, int minDepth ) {

                this.name = name;
                this.maxQuantity = maxQuantity;
                this.minQuantity = minQuantity;
                this.maxDepth = maxDepth;
                this.minDepth = minDepth;
        }

        public String getName() {

                return this.name;
        }

        public void setName( String name ) {

                this.name = name;
        }

        public boolean isInjected() {

                return this.injected;
        }

        public boolean inject( int quantity ) {

                return this.injected;
        }

        public boolean inject( int quantity, int depth ) {

                return this.injected;
        }
}
```

```
public class Cow extends Animal {

        public Cow( String name, int maxQuantity, int minQuantity ) {

                super( name, maxQuantity, minQuantity );
        }

        public Cow( String name, int maxQuantity, int minQuantity, int
maxDepth, int minDepth ) {

                super( name, maxQuantity, minQuantity, maxDepth, minDepth );
        }

        public boolean inject( int quantity ) {

                if( !this.injected && quantity >= minQuantity && quantity <=
maxQuantity ) {

                        this.injected = true;
                }

                return this.injected;
        }

        public boolean inject( int quantity, int depth ) {

                if( !this.injected && quantity >= minQuantity && quantity <=
maxQuantity && depth >= minDepth && depth <= maxDepth ) {

                        this.injected = true;
                }

                return this.injected;
        }
}

Animal cow1 = new Cow( "Jersey", 10, 50 );
Cow cow2 = new Cow( "Sahiwal", 10, 40, 5, 10 );

// Runtime Polymorphism - Overriden Method - Decided dynamically based on the
Object type
// Inject method of the Cow class will be called, since the cow1 reference of
Animal type refers to Cow object
cow1.inject( 25 );

// Compile-Time Polymorphism - Overloaded Method - Decided based on the
```

```
method signature
cow2.inject( 15, 6 );

// Upcasting
Animal cow3 = cow1;
Injectable cow4 = cow1;

// Downcasting
Cow cow5 = (Cow) cow3;
```

The Java OOP Cheatsheet by https://www.tutorials24x7.com. Download PDF.

Notes

This cheatsheet does not cover all the OOP concepts of Java.
It can be considered as a reference to quickly revise or learn the OOP concepts of Java.
Submit the Contact Form in case you face any issues in using it or finds any discrepancy.