Coding Cheat Sheet

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

- Inheritance in Java
- Polymorphism in Java
- Interfaces and abstract classes in Java
- Inner classes in Java

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin coding with Java after this course!

Inheritance in Java

Creating a superclass

Description	**Example**
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Define a String variable `name` to store the name of the animal.	`String name;`
Include a method `eat()` to print the message that the animal is eating.	`void eat() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`System.out.println(name + " is eating.");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	**Example**

Creating a subclass

Description	**Example**
The `Dog` class inherits from the `Animal` class, meaning it automatically gets all properties and methods from `Animal`.	`class Dog extends Animal {`
Include a method `bark()` to print the message that the dog is barking.	`    void bark() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`        System.out.println(name + " says woof!");`

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

Using inheritance

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) {`
Creates an instance of the `Dog` class. The `Dog` class inherits from the `Animal` class.	`Dog myDog = new Dog();`
Assigns "Buddy" to the `name` variable inherited from `Animal`.	`myDog.name = "Buddy";`
Calls the `eat()` method from the `Animal` class, which prints "Buddy is eating.".	`myDog.eat();`
Calls the `bark()` method from the `Dog` class, which prints "Buddy says woof!".	`myDog.bark();`

Description	**Example**

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

Using multilevel inheritance

Description	**Example**
The `Puppy` class inherits from the `Dog` class. Since `Dog` already inherits from `Animal`, `Puppy` indirectly inherits all properties and methods from `Animal` as well.	`class Puppy extends Dog {`
This method adds a new behavior specific to the `Puppy` class.	`    void weep() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`        System.out.println(name + " is weeping.");`

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

Explanation: This is an example of multilevel inheritance. Animal (Superclass) → Dog (Subclass) → Puppy (Subclass of Dog). The `Animal` class has attribute `name` and method `eat()`. The `Dog` class inherits from `Animal` and adds the `bark()` method. `Puppy` inherits from `Dog` and adds the `weep()` method.

Using hierarchical inheritance

Description	**Example**
The `Cat` class inherits from the `Animal` class. Since `Animal` contains the `name` variable and `eat()` method, `Cat` inherits those properties.	`class Cat extends Animal {`
This method adds a new behavior specific to the `Cat` class.	`void meow() {`
Print the message to the console using the `System.out.println()` function. The animal `name` is displayed dynamically.	`System.out.println(name + " says meow!");`

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

Explanation: This is an example of hierarchical inheritance because multiple subclasses (`Dog` and `Cat`) inherit from the same superclass (`Animal`). `Animal` has attribute `name` and method `eat()`. `Dog` and `Cat` inherit from `Animal`, but each adds unique behaviors. `Dog` adds the `bark()` method and `Cat` adds the `meow()` method.

Method overriding

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behavors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The @Override annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`

Description	Example
Include a `sound()` method to print the message "Dog barks".	`void sound() {`
Print the message to the console using the `System.out.println()` function.	`System.out.println("Dog barks");`
Close curly braces to end the `Dog` class definition.	`}` `}`

Explanation: In this example, `Dog` provides its own implementation of `sound()`, replacing the one in `Animal`. Method overriding occurs when a subclass provides a specific implementation of a method already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

Using overridden methods

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) {`

Description	Example
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Animal();`
The `Dog` object is stored in an `Animal` reference. Since Dog overrides the sound() method, Java uses dynamic method dispatch to call the overridden method in `Dog`, not in `Animal`.	`Animal myDog = new Dog();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
Since `myDog` refers to a `Dog` object (even though it's declared as Animal), it calls the overridden `sound()` method in `Dog` due to polymorphism.	`myDog.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: The `Dog` class inherits from `Animal`, meaning it gets all non-private properties and methods of `Animal`. `Dog` overrides the `sound()` method from `Animal`, providing a more specific implementation. Even though `myDog` is declared as an `Animal`, Java determines the method to call at runtime, not compile time. When calling `myDog.sound()`, Java looks at the actual object type (Dog) and calls `sound()` from `Dog`, not `Animal`.

Polymorphism in Java

Compile-time polymorphism

Description	**Example**
Create a class `MathOperations` that contains multiple methods for performing addition.	`class MathOperations {`
Include an `add` method that accepts two `int` values (a and b).	`int add(int a, int b) {`
Add the values of `a` and `b` and return the sum to the calling method as an `int`.	`return a + b;`
Close curly braces to end the method.	`}`
Include an `add` method that accepts three `int` values (a, b, and c).	`int add(int a, int b, int c) {`
Add the values of `a`, `b`, and `c` and return the sum to the calling method as an `int`. This method overloads the first `add()` method because it has different number of parameters.	`return a + b + c;`

Description	**Example**
Close curly braces to end the method.	`}`
Include an `add` method that accepts two `double` values (a and b).	`int add(double a, double b) {`
Add the values of `a` and `b` and return the sum to the calling method as a `double`. This method overloads both of the previous `add()` methods, but it works with double values instead of int.	`return a + b;`

Close curly braces to end the method and the `MathOperations` 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) {`

Description	**Example**
Create an instance of the `MathOperations` class and assign it to the `math` object.	`MathOperations math = new MathOperations();`
Calls the method `add(int a, int b)` to add two integers (2 + 3) and print the result to the console.	`System.out.println("Sum of 2 and 3: " + math.add(2, 3));`
Calls the method `add(int a, int b, int c)` to add three integers (2 + 3 + 4) and print the result to the console.	`System.out.println("Sum of 2, 3 and 4: " + math.add(2, 3, 4));`
Calls the method `add(double a, double b)` to add two double values (2.5 + 3.5) and print the result to the console.	`System.out.println("Sum of 2.5 and 3.5: " + math.add(2.5, 3.5));`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: The `add()` method is overloaded three times in the `MathOperations` class. Different number of parameters (int a, int b) versus (int a, int b, int c) and different types of parameters (int versus double). In Java, overloading is based on the method signature, which includes the number and types of parameters. It does not depend on the return type. The correct method is selected at compile time based on the arguments passed to the `add()` method. This is an example of compile-time polymorphism (or static polymorphism).

Using compile-time polymorphism

Description	**Example**
Create a class `MathOperations` that contains multiple methods for performing addition.	`class MathOperations {`
Include an `add` method that accepts two `int` values (a and b).	`    int add(int a, int b) {`
Add the values of `a` and `b` and return the sum to the calling method as an `int`.	`        return a + b;`
Close curly braces to end the method.	`    }`
Include an `add` method that accepts two `double` values (a and b).	`    int add(double a, double b) {`
Add the values of `a` and `b` and return the sum to the calling method as a `double`. This method overloads both of the previous `add()` methods, but it works with double values instead of int.	`        return a + b;`

Description	**Example**
Close curly braces to end the method.	`}`
Include an `add` method that accepts three `int` values (a, b, and c).	`int add(int a, int b, int c) {`
Add the values of `a`, `b`, and `c` and return the sum to the calling method as an `int`. This method overloads the first `add()` method because it has different number of parameters.	`return a + b + c;`
Close curly braces to end the method and the `MathOperations` 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) {`

Description	**Example**
Create an instance of the `MathOperations` class and assign it to the `math` object.	`MathOperations math = new MathOperations();`
Calls the method `add(int a, int b)` to add two integers (2 + 3) and print the result to the console.	`System.out.println("Sum of 2 and 3: " + math.add(2, 3));`
Calls the method `add(double a, double b)` to add two double values (2.5 + 3.5) and print the result to the console.	`System.out.println("Sum of 2.5 and 3.5: " + math.add(2.5, 3.5));`
Calls the method `add(int a, int b, int c)` to add three integers (2 + 3 + 4) and print the result to the console.	`System.out.println("Sum of 1, 2 and 3: " + math.add(2, 3, 4));`

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

Explanation: In this example, the `MathOperations` class has three overloaded `add` methods. Depending on the number and type of arguments passed to `add`, Java determines which method to invoke at compile time. This makes our code more flexible and easier to read.

Using runtime polymorphism

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behavors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The @Override annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`

Description	Example
Include a `sound()` method to print the message "Dog barks".	`    void sound() {`
Print the message to the console using the `System.out.println()` function.	`        System.out.println("Dog barks");`

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

Description	Example
The `Cat` class inherits from the `Animal` class.	`class Cat extends Animal {`
`Cat` overrides the `sound()` method to provide a specific implementation: "Cat meows". The @Override annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`    @Override`
Include a `sound()` method to print the message "Cat meows".	`    void sound() {`

Description	Example
Print the message to the console using the `System.out.println()` function.	`System.out.println("Cat meows");`
Close curly braces to end the `Cat` 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) {`
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Animal();`
The `Dog` object is stored in an `Animal` reference. Since Dog overrides the sound() method, Java uses dynamic method dispatch to call the overridden method in `Dog`, not in `Animal`.	`myAnimal = new Dog();`

Description	Example
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
The `Cat` object is stored in an `Animal` reference. Since Cat overrides the sound() method, Java uses dynamic method dispatch to call the overridden method in `Cat`, not in `Animal`.	`myAnimal = new Cat();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the `Animal` class.	`myAnimal.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, `Animal` is a superclass with a method called `sound()`. Both `Dog` and `Cat` classes extend `Animal`, providing their own implementation of the `sound()` method. When we create an `Animal` reference and assign it to different subclasses (Dog and Cat), the appropriate `sound()` method is called at runtime based on the object type. This allows for more dynamic and flexible code.

Creating virtual methods

Description	Example
Create a superclass named `Animal`, which serves as a base class for other classes that might inherit from it.	`class Animal {`

Description	Example
Include a `sound()` method. This method is meant to be overridden by subclasses that define more specific behavors.	`void sound() {`
Print the message "Animal makes a sound" to the console using the `System.out.println()` function.	`System.out.println("Animal makes a sound");`
Close curly braces to end the `Animal` class definition.	`}` `}`

Description	Example
The `Dog` class inherits from the `Animal` class.	`class Dog extends Animal {`
`Dog` overrides the `sound()` method to provide a specific implementation: "Dog barks". The @Override annotation tells the compiler that this method replaces the `sound()` method from `Animal`.	`@Override`
Include a `sound()` method to print the message "Dog barks".	`void sound() {`

Description	Example
Print the message to the console using the `System.out.println()` function.	`System.out.println("Dog barks");`
Close curly braces to end the `Dog` 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) {`
Creates an instance of `Animal` and stores it in a variable `myAnimal`.	`Animal myAnimal = new Dog();`
Since `myAnimal` is a regular `Animal` object, calling `myAnimal.sound()` executes the `sound()` method from the	`myAnimal.sound();`

Description	Example
`Animal` class.	

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

Explanation: In this example, even though `myAnimal` is an `Animal`, the `sound()` method from the `Dog` class is called, demonstrating virtual method behavior.

Designing interfaces and abstract classes

Creating an interface

Description	Example
Declare an `Animal` interface.	`interface Animal {`
Include a method `sound()`. Any class that implements this interface must provide an implementation of `sound()`.	`    void sound();`
Close curly braces to end the interface definition.	`}`

Description	**Example**
Create a `Dog` class that implements the `Animal` interface.	`class Dog implements Animal {`
Include a `sound()` method for the class.	`    public void sound() {`
Calling `sound()` prints "Bark" to the console using the `System.out.println()` function.	`        System.out.println("Bark");`

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

Description	**Example**
Create a `Cat` class that implements the `Animal` interface.	`class Cat implements Animal {`
Include a `sound()` method for the class.	`    public void sound() {`

Description	**Example**
Calling `sound()` prints "Meow" to the console using the `System.out.println()` function.	`System.out.println("Meow");`
Close curly braces to end the `Cat` 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 the `Dog` object and assign it to the variable `dog`.	`Animal dog = new Dog();`
Create the `Cat` object and assign it to the variable `cat`.	`Animal cat = new Cat();`

Description	Example
Call `sound()` on the `dog` object. This prints the message "Bark".	`dog.sound();`
Call `sound()` on the `cat` object. This prints the message "Meow".	`cat.sound();`
Close curly braces to end the `Main` class definition.	`}` `}`

Explanation: In this example, we define an interface `Animal` with a method `sound()`. The `Dog` and `Cat` classes implement the `Animal` interface and provide their own versions of the `sound()` method. In the `Main` class, we create instances of `Dog` and `Cat`, calling the `sound()` method on each to demonstrate polymorphism.

Creating an abstract class

Description	Example
Create an abstract class `Shape` that cannot be instantiated directly.	`abstract class Shape {`
Include an abstract method `draw()` that must be implemented by any subclass.	`abstract void draw();`

Description	**Example**
Include a concrete method `display()` that has a default implementation.	`void display() {`
Calling the `display()` method prints "This is a shape." to the console using the `System.out.println()` function.	`System.out.println("This is a shape.");`
Close curly braces to end the `Dog` class definition.	`}` `}`

Description	**Example**
Create a `Circle` class that extends the `Shape` class.	`class Circle extends Shape {`
Include a `draw()` method for the class.	`public void draw() {`
Calling the `draw()` method prints "Drawing Circle" to the console using the `System.out.println()` function.	`System.out.println("Drawing Circle");`

Description	Example
Close curly braces to end the `Dog` 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) {`
The `shape` object is instantiated from the `Shape` class but it refers to a `Circle` object.	`Shape shape = new Circle();`
Calling `draw()` on the `shape` object prints "Drawing Circle".	`shape.draw();`
Calling `display()` on the `shape` object prints "This is a shape."	`shape.display();`

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

Explanation: In this example, we define an abstract class `Shape` with an abstract method `draw()` and a concrete method `display()`. The `Circle` class extends the `Shape` class and provides an implementation for the `draw()` method. In the `Main` class, we create an instance of `Circle` using the `Shape` reference type to show how it works. The `draw()` method executes the overridden version from `Circle`. The `display()` method is inherited from `Shape` and is called as is.

Inner classes in Java

Creating inner classes

Description	**Example**
Create an `OuterClass` that works as a container for the inner class.	`class OuterClass {`
Set the value of the `int outerVariable` to 10.	`int outerVariable = 10;`
Create a classs `InnerClass` inside the `OuterClass`.	`class InnerClass {`
Include a method `display()` that accesses `OuterVariable`	`void display();`

Description	Example
from the outer class. Inner classes have direct access to the outer class's members (including private ones).	
Calling the `display()` method prints the `outerVariable` value to the console using the `System.out.println()` function. The `outerVariable` value is generated dynamically.	`System.out.println("Outer variable value: " + outerVariable);`
Close curly braces to end the `OuterClass` class definition.	`}` `}` `}`

Explanation: In this example, `OuterClass` contains a variable `outerVariable`. `InnerClass` is defined inside `OuterClass` and has a method `display()`. This method can access `outerVariable` directly.

Using inner classes

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 `OuterClass`. This is necessary because non-static inner	`OuterClass outer = new OuterClass();`

Description	Example
classes require an instance of the outer class to be created first.	
Create a classs `InnerClass` inside the `OuterClass`. Since `InnerClass` is a non-static inner class, it must be created using an instance of `OuterClass`.	`OuterClass.InnerClass inner = outer.new InnerClass();`
Call the `display()` method inside `InnerClass`.	`inner.display();`

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

Explanation: In this example, `InnerClass` is nested inside `OuterClass` and has access to all outer class's members. The `display()` method will print the value of the `outerVariable`. The code demonstrates encapsulation in Java.

Creating a static nested classes

Description	Example
Create an `OuterClass` that works as a container for the inner class.	`class OuterClass {`
Set the value of the `int outerVariable` to 20.	`static int staticVariable = 20;`

Description	Example
Create a classs `InnerClass` inside the `OuterClass`.	`static class StaticNestedClass {`
Include a method `show()` that accesses `OuterVariable` from the outer class. Inner classes have direct access to the outer class's members (including private ones).	`void show();`
Calling the `show()` method prints the `outerVariable` value to the console using the `System.out.println()` function. The `outerVariable` value is generated dynamically.	`System.out.println("Static variable value: " + staticVariable);`
Close curly braces to end the `OuterClass` class definition.	`}` `}` `}`

Explanation: In this example, `OuterClass` contains a static variable named `staticVariable` with a value of 20. Since the variable is static, it belongs to the class itself rather than an instance. Static nested classes do not require an instance of the outer class. It can access `staticVariable` without an instance of `OuterClass`. The nested class keeps related logic inside `OuterClass`, improving organization.

Using a static nested classes

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 `StaticNestedClass` inside the `OuterClass`.	`OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();`
Include a method `nested.show()` that prints the value of the `staticVariable` from `OuterClass`.	`nested.show();`
Close curly braces to end the `OuterClass` class definition.	`}` `}` `}`

Creating a method-local inner class

Description	Example

Create an `OuterClass` with a method myMethod() that will define and use a method-local inner class.	`class OuterClass {`
	`void myMethod() {`

Description	Example

Define a class `MethodLocalInner` inside `myMethod()`. `MethodLocalInner` is local to the method, meaning that it cannot be accessed outside of `myMethod()`. Calling `MethodLocalInner` prints the message "Inside Method Local Inner Class" to the console using the `System.out.println()` function.	`class MethodLocalInner {`
	`    void display() {`
	`        System.out.println("Inside Method Local Inner Class");`
	`    }`
	`}`

The inner class is instantiated within the method where it is defined.	`MethodLocalInner inner = new MethodLocalInner();`
`inner.display()` calls the `display()` method, printing "Inside Method Local Inner Class".	`inner.display();`

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

Creating an anonymous inner class

Description	Example

The `Greeting` interface defines a single method `greet()`, which must be implemented by any class that uses this interface.	`interface Greeting {`
	`void greet();`
	`}`

Description	Example

This creates an anonymous inner class that implements the `Greeting` interface. The anonymous class provides an implementation for the `greet()` method at the moment of object creation.	`public class Main {`
	`    public static void main(String[] args) {`
	`        Greeting greeting = new Greeting() {`
	`            public void greet() {`
	`                System.out.println("Hello from Anonymous Inner Class!");`
	`            }`
	`        };`

This calls the overridden `greet()` method in the anonymous inner class, printing "Hello from Anonymous Inner Class!".	`        greeting.greet();`

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

Using inner classes in the real world

Description	Example

The `Library` class represents a library and has a private variable `libraryName` to store its name. A constructor initializes `libraryName`.	`class Library {`
	`    private String libraryName;`
	`    public Library(String name) {`
	`        this.libraryName = name;`
	`    }`

Description	**Example**

Nested inside `Library`, this class represents a book. It has two private attributes: `title` and `author`. The `Book` class has a constructor to initialize these attributes. The `displayBookInfo()` method prints the book's title and author. It also accesses `libraryName` from `Library`, demonstrating how inner classes can access private members of the outer class.	`class Book {`
	`    private String title;`
	`    private String author;`
	`    public Book(String title, String author) {`
	`        this.title = title;`
	`        this.author = author;`
	`    }`
	`    public void displayBookInfo() {`
	`        System.out.println("Library: " + libraryName);`
	`        System.out.println("Book Title: " + title);`
	`        System.out.println("Author: " + author);`
	`    }`
	`}`

This creates a `Library` instance named "City Library" and creates a `Book` instance associated with that library. Since `Book` is a non-static inner class, it must be created using an instance of Library. The `displayBookInfo()` method in the `Book` inner class prints out the name of the library along with the book's title and author.	`public class Main {`
	`    public static void main(String[] args) {`
	`        Library myLibrary = new Library("City Library");`
	`        Library.Book myBook = myLibrary.new Book("1984", "George Orwell");`
	`        myBook.displayBookInfo();`

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

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan