Java Chapter 7: Object-Oriented Programming

Contents

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself

Read: Think Java, 2nd Ed.

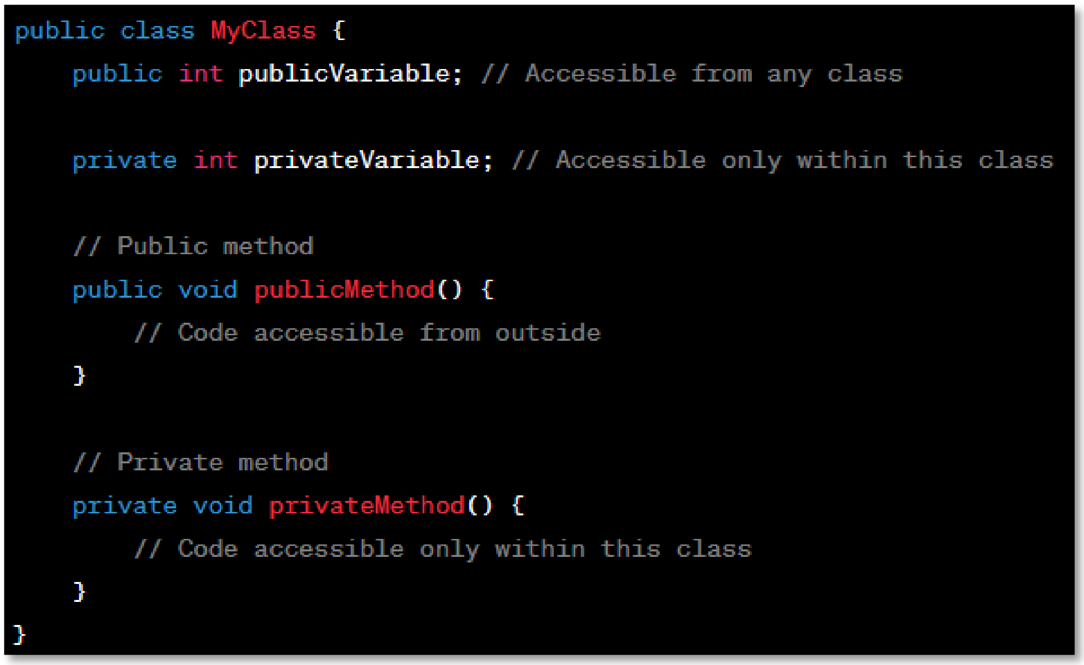
- Chapter 11 Designing Classes

Do: Online Tutorials

- LearnJavaOnline.org Objects
- Java OOP

- [Java Classes/Objects]
- [Java Class Attributes]
- [Java Class Methods]
- [Java Constructors]
- [Java Encapsulation]

Java Public and Private



```java
public class MyClass {
    public int publicVariable; // Accessible from any class

    private int privateVariable; // Accessible only within this class

    // Public method
    public void publicMethod() {
        // Code accessible from outside
    }

    // Private method
    private void privateMethod() {
        // Code accessible only within this class
    }
}
```

In Java, public and private are access modifiers used to control the visibility of class members.

Public: Allows members to be accessed from any other class. Use it when you want a member to be accessible from outside the class.

Private: Restricts access to the member within the same class. Use it when you want to encapsulate the implementation details and prevent direct external access.

Tutorial 7.1: Sammy the Shark

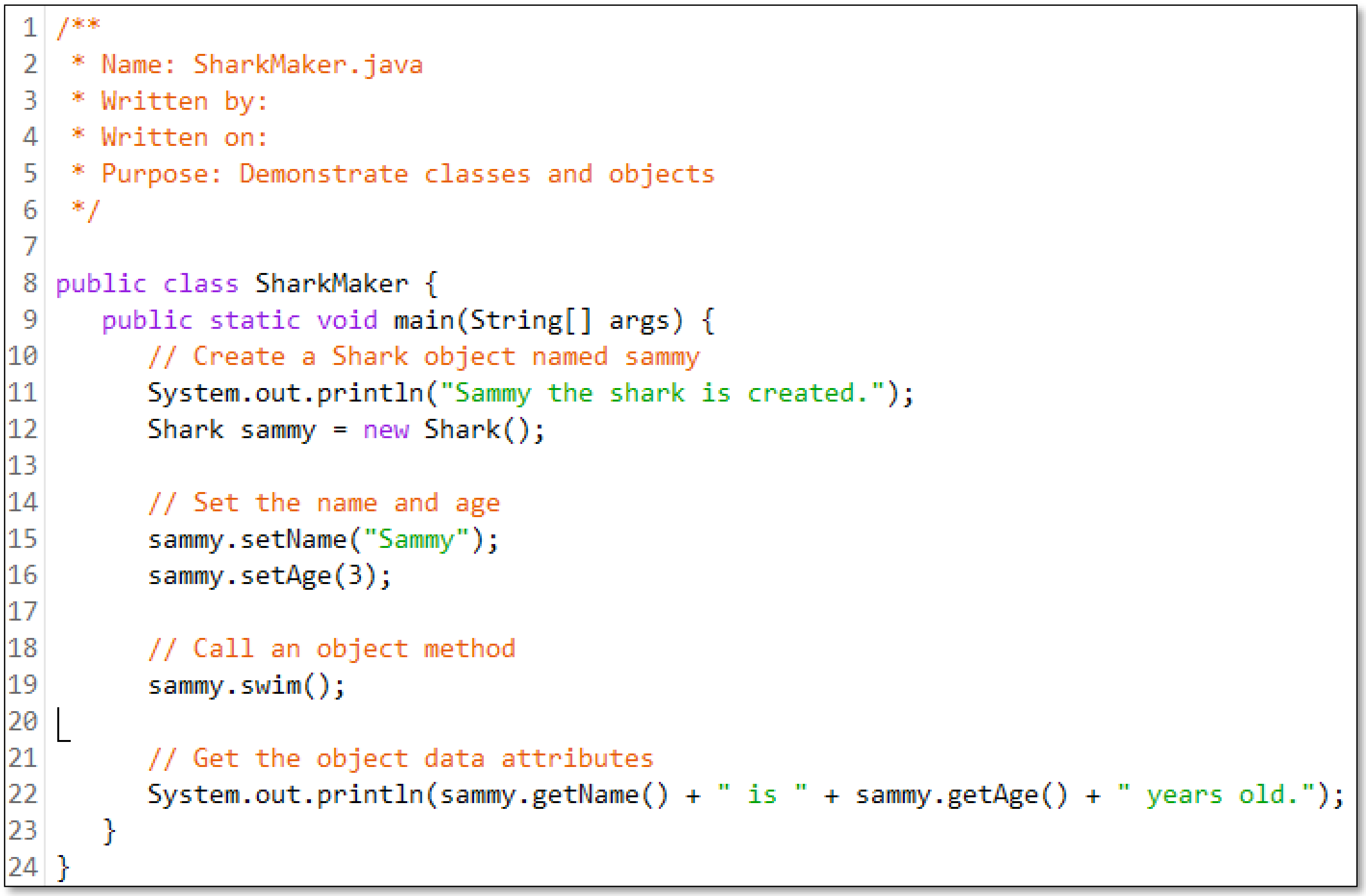
The following program demonstrates classes, objects, and methods. It uses a concept called **getters and setters** to set data and get data from an object.

An object is an instance of a class. We'll construct a **Shark** object called **sammy**.

Create a Java class named: **Shark.java**

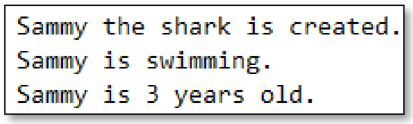
```java
/**
 * Name: Shark.java
 * Written by:
 * Written on:
 * Purpose: Class to create a shark
 */

public class Shark {
   // Define class data attributes
   private String name;
   private int age;

   // Define class getters and setters
   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   // Define class methods
   public void swim() {
      // this references the class data attribute
      System.out.println(this.name + " is swimming.");
   }
}
```

Create a Java program named: **SharkMaker.java**



```java
/**
 * Name: SharkMaker.java
 * Written by:
 * Written on:
 * Purpose: Demonstrate classes and objects
 */

public class SharkMaker {
   public static void main(String[] args) {
      // Create a Shark object named sammy
      System.out.println("Sammy the shark is created.");
      Shark sammy = new Shark();

      // Set the name and age
      sammy.setName("Sammy");
      sammy.setAge(3);

      // Call an object method
      sammy.swim();

      // Get the object data attributes
      System.out.println(sammy.getName() + " is " + sammy.getAge() + " years old.");
   }
}
```

Example run:



```
Sammy the shark is created.
Sammy is swimming.
Sammy is 3 years old.
```

Tutorial 7.2: Constructing Sharks

A constructor is a special method that is used to create an object from a class. It runs as soon as an object of a class is instantiated.

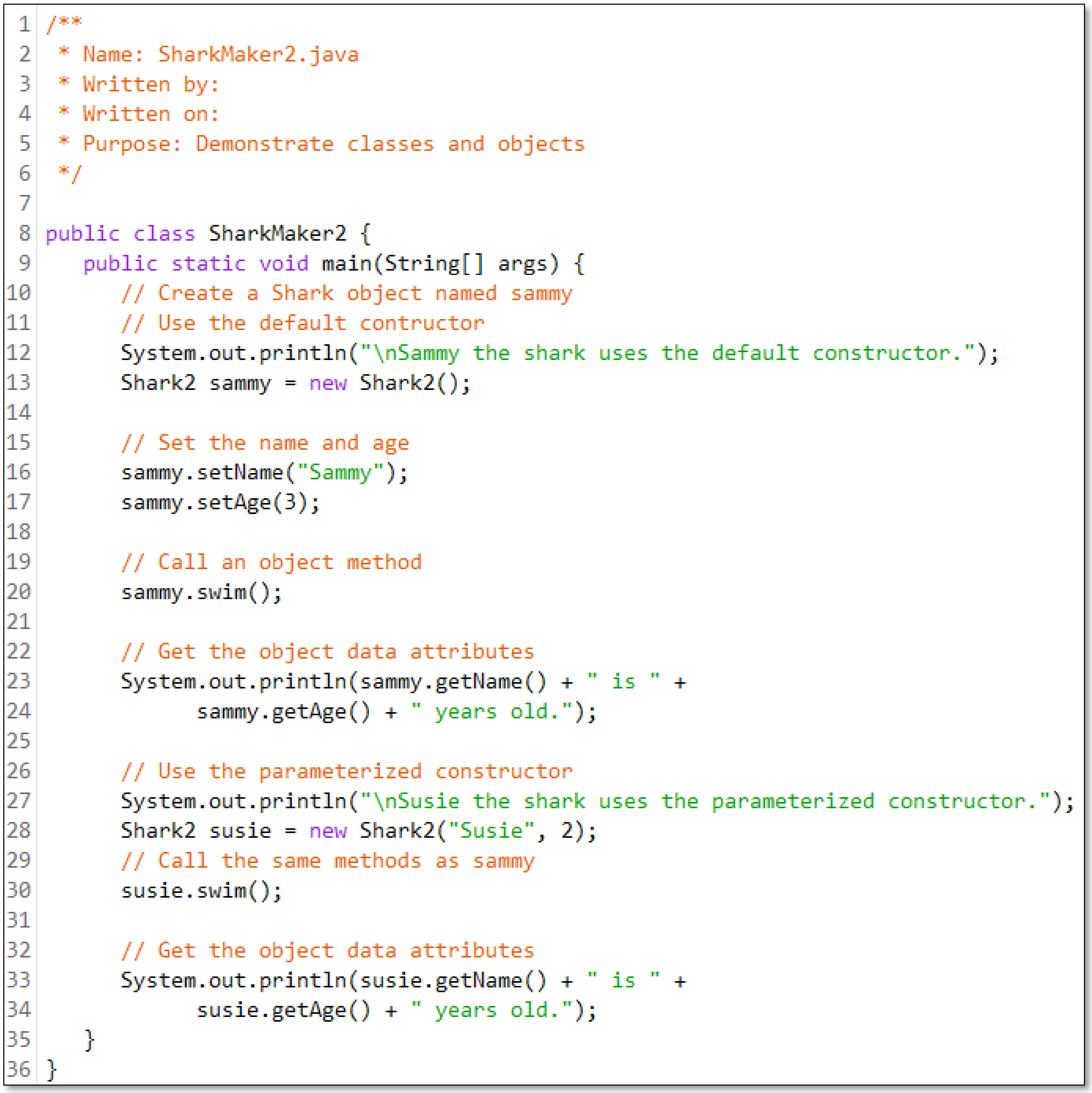
Classes are useful because they allow us to create many similar objects based on the same blueprint. This program creates two objects from the same class using the default constructor and a parameterized constructor.

1. Create a Java class named: **Shark2.java**

2. Create a Java program named: **SharkMaker2.java**

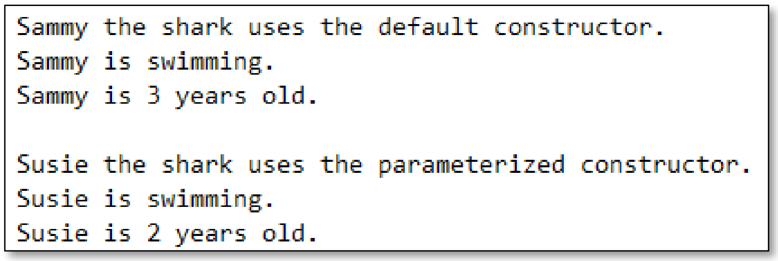
3. Copy the previous tutorial code into each file.

```java
/**
 * Name: Shark2.java
 * Written by:
 * Written on:
 * Purpose: Class with parameterized constructor to create a shark
 * If there is a parameterized constructor,
 * a default constructor must be put in manually
 */

public class Shark2 {
   // Define class data attributes
   private String name;
   private int age;

   // Default constructor
   public Shark2() {
   };

   // Constructor with two parameters
   public Shark2(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // Define class getters and setters
   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   // Define class methods
   public void swim() {
      // this references the class data attribute
      System.out.println(this.name + " is swimming.");
   }
}
```



```java
/**
 * Name: SharkMaker2.java
 * Written by:
 * Written on:
 * Purpose: Demonstrate classes and objects
 */

public class SharkMaker2 {
   public static void main(String[] args) {
      // Create a Shark object named sammy
      // Use the default contructor
      System.out.println("\nSammy the shark uses the default constructor.");
      Shark2 sammy = new Shark2();

      // Set the name and age
      sammy.setName("Sammy");
      sammy.setAge(3);

      // Call an object method
      sammy.swim();

      // Get the object data attributes
      System.out.println(sammy.getName() + " is " +
            sammy.getAge() + " years old.");

      // Use the parameterized constructor
      System.out.println("\nSusie the shark uses the parameterized constructor.");
      Shark2 susie = new Shark2("Susie", 2);
      // Call the same methods as sammy
      susie.swim();

      // Get the object data attributes
      System.out.println(susie.getName() + " is " +
            susie.getAge() + " years old.");
   }
}
```

Example run:



```
Sammy the shark uses the default constructor.
Sammy is swimming.
Sammy is 3 years old.

Susie the shark uses the parameterized constructor.
Susie is swimming.
Susie is 2 years old.
```

ThreadLocalRandom

ThreadLocalRandom.current().nextInt() in Java generates a random integer using the **ThreadLocalRandom** class, which is a variation of the Random class optimized for concurrent use.

ThreadLocalRandom: This class provides random number generation that is local to each thread. It helps avoid contention for a single random number generator instance among multiple threads.

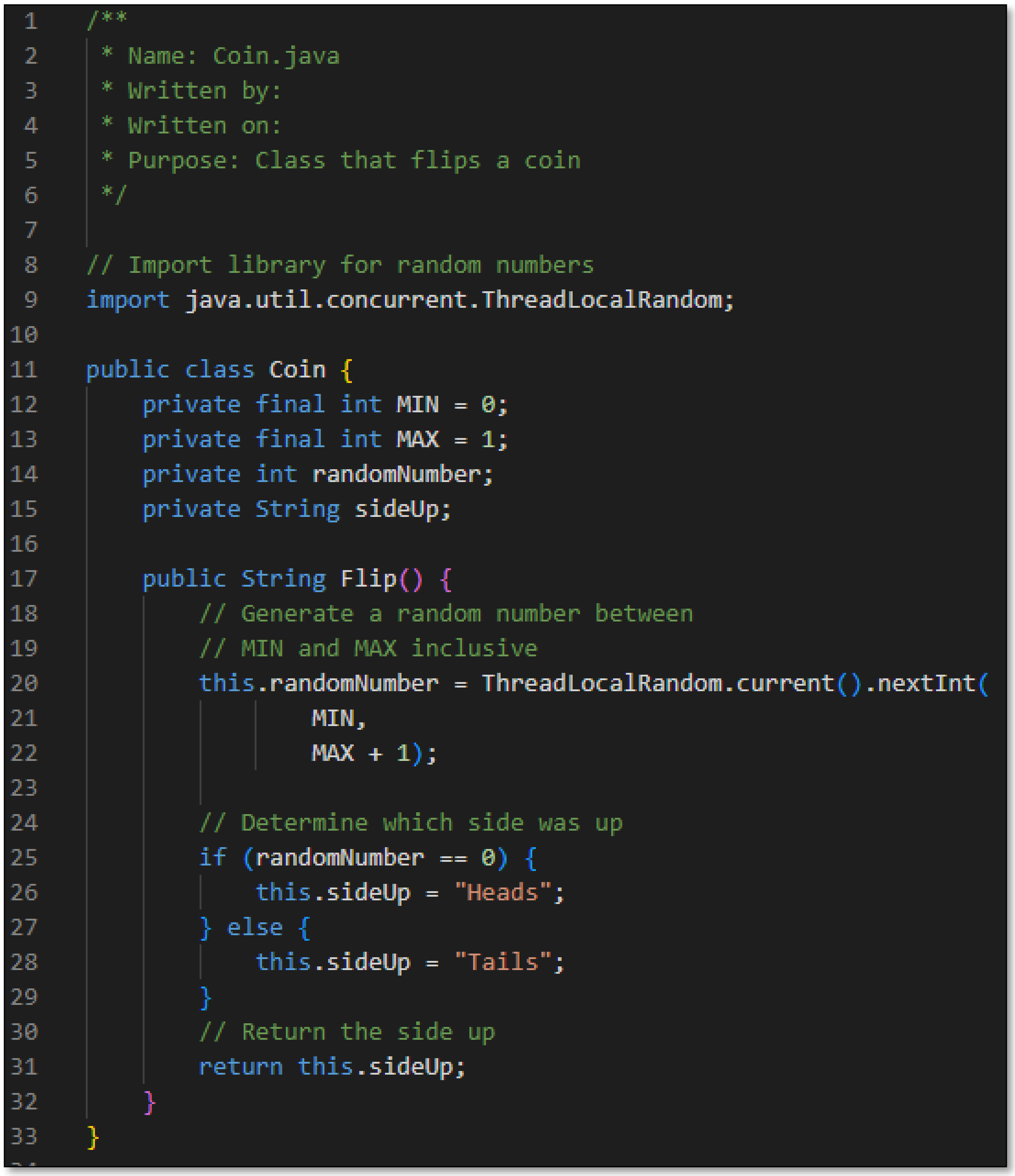
.current(): Returns the current thread's ThreadLocalRandom instance. Each thread has its own independent random number generator.

.nextInt(): Generates a random integer. This method is like **Random.nextInt()**, but it's designed to be more efficient in a concurrent environment.

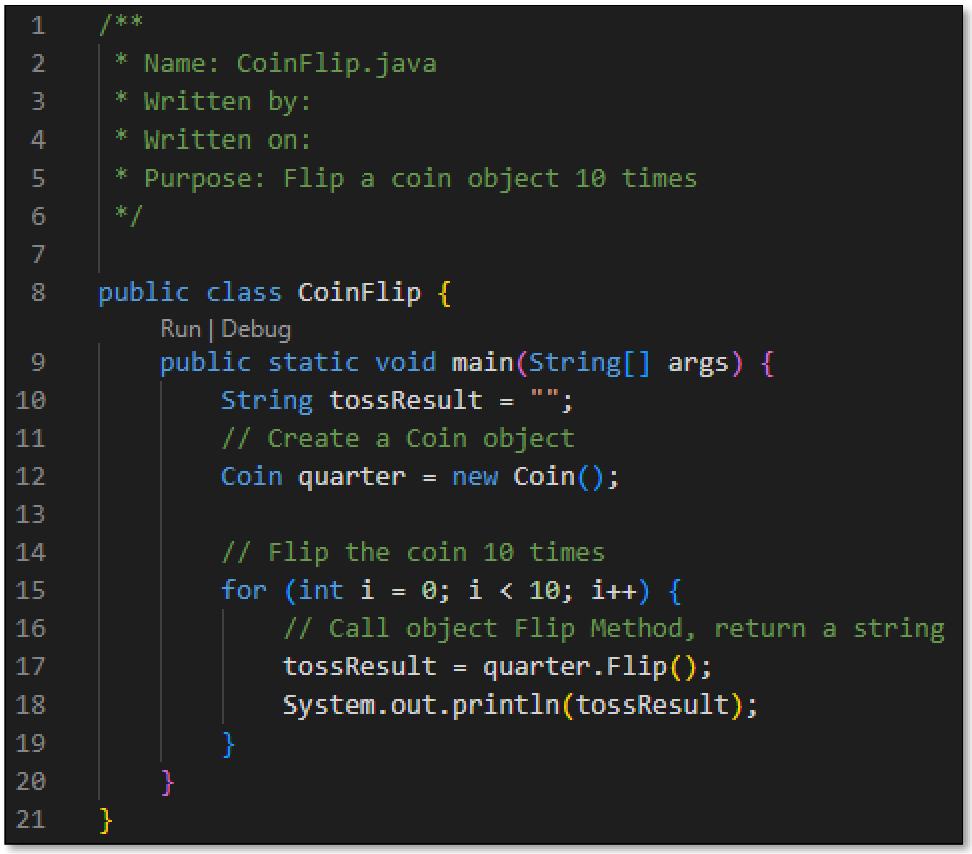
Tutorial 7.3: Coin Flip

The **Coin** class holds all the attributes and methods to flip a coin and return the value. The **CoinFlip** program creates a coin object and flips it 10 times.

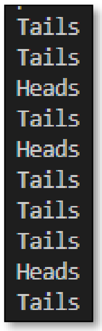
Create a Java class named **Coin.java**


```java
/**
 * Name: Coin.java
 * Written by:
 * Written on:
 * Purpose: Class that flips a coin
 */

// Import library for random numbers
import java.util.concurrent.ThreadLocalRandom;

public class Coin {
    private final int MIN = 0;
    private final int MAX = 1;
    private int randomNumber;
    private String sideUp;

    public String Flip() {
        // Generate a random number between
        // MIN and MAX inclusive
        this.randomNumber = ThreadLocalRandom.current().nextInt(
                MIN,
                MAX + 1);

        // Determine which side was up
        if (randomNumber == 0) {
            this.sideUp = "Heads";
        } else {
            this.sideUp = "Tails";
        }
        // Return the side up
        return this.sideUp;
    }
}
```

Create a Java program that uses the Coin class named **CoinFlip.py** to flip a coin 10 times.


```java
/**
 * Name: CoinFlip.java
 * Written by:
 * Written on:
 * Purpose: Flip a coin object 10 times
 */

public class CoinFlip {
    Run | Debug
    public static void main(String[] args) {
        String tossResult = "";
        // Create a Coin object
        Coin quarter = new Coin();

        // Flip the coin 10 times
        for (int i = 0; i < 10; i++) {
            // Call object Flip Method, return a string
            tossResult = quarter.Flip();
            System.out.println(tossResult);
        }
    }
}
```

Example run:

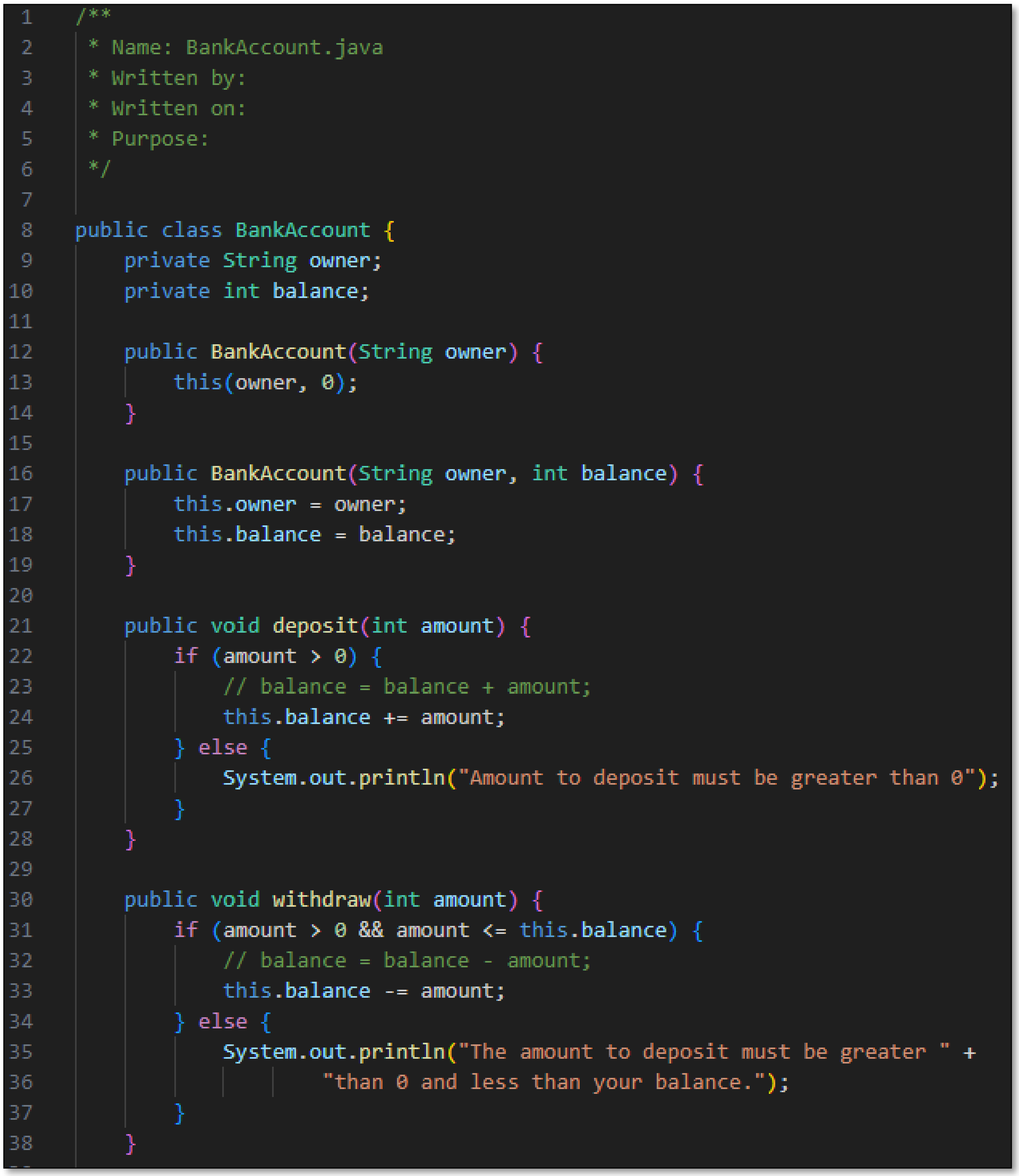

```
Tails
Tails
Heads
Tails
Heads
Tails
Tails
Tails
Heads
Tails
```

Tutorial 7.4: Bank Class Calculated Properties

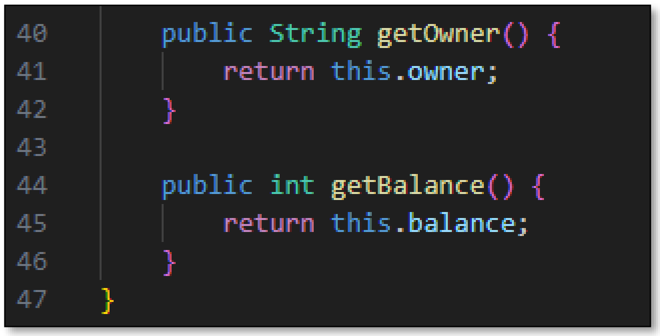
Not all properties have to be explicitly set, they can be calculated based on other properties.

- The balance is calculated based on the deviposits and withdrawals; it is never set explicitly.
- If the user doesn't have enough money, they can't make a withdrawal.
- There are two constructors.

Create a Java file named: **BankAccount.java**

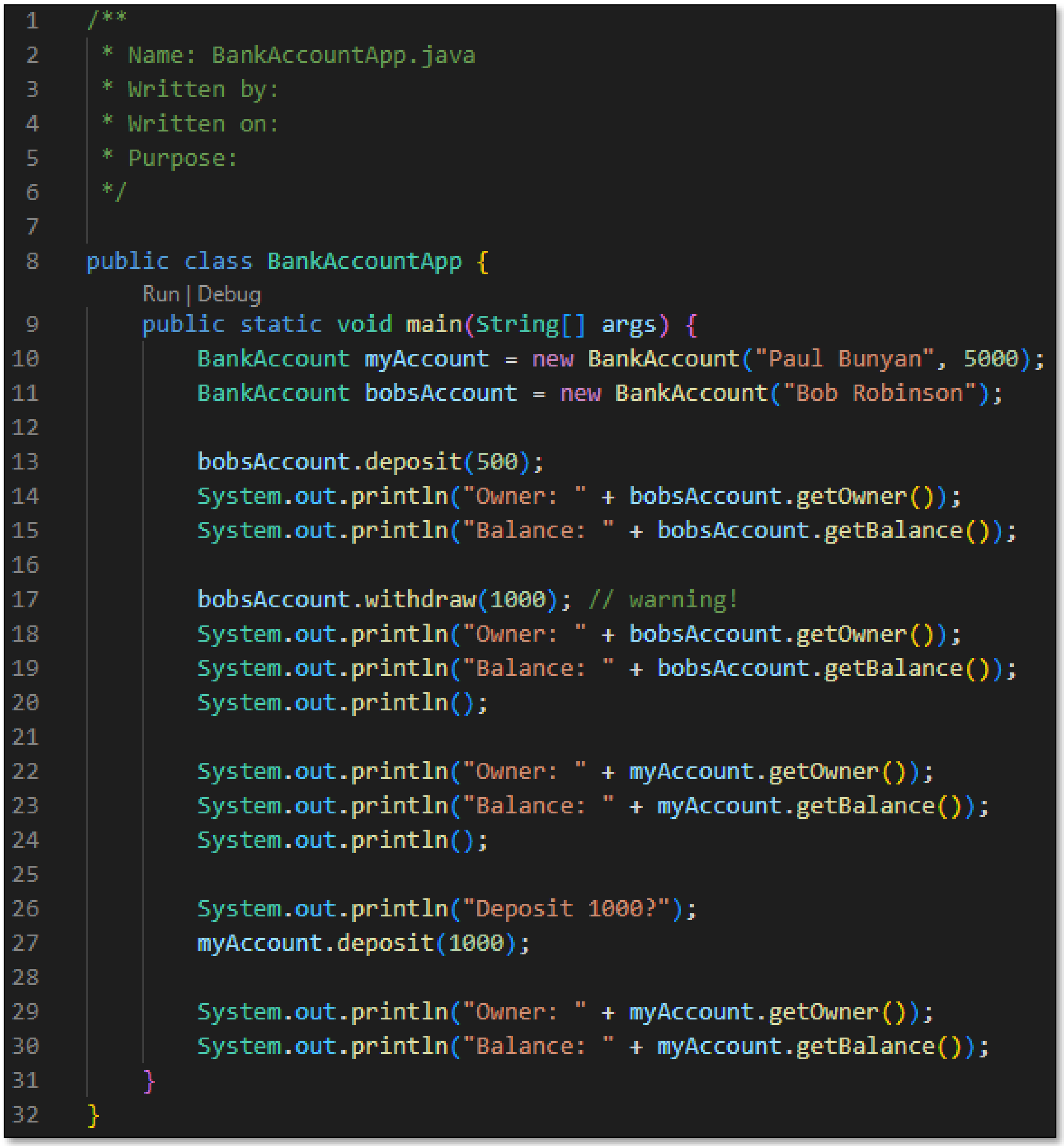


```java
/**
 * Name: BankAccount.java
 * Written by:
 * Written on:
 * Purpose:
 */

public class BankAccount {
    private String owner;
    private int balance;

    public BankAccount(String owner) {
        this(owner, 0);
    }

    public BankAccount(String owner, int balance) {
        this.owner = owner;
        this.balance = balance;
    }

    public void deposit(int amount) {
        if (amount > 0) {
            // balance = balance + amount;
            this.balance += amount;
        } else {
            System.out.println("Amount to deposit must be greater than 0");
        }
    }

    public void withdraw(int amount) {
        if (amount > 0 && amount <= this.balance) {
            // balance = balance - amount;
            this.balance -= amount;
        } else {
            System.out.println("The amount to deposit must be greater " +
                    "than 0 and less than your balance.");
        }
    }
```



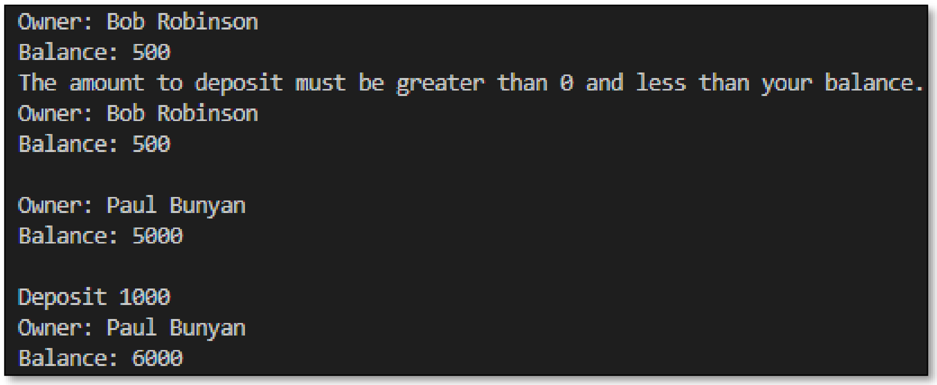
```
    public String getOwner() {
        return this.owner;
    }

    public int getBalance() {
        return this.balance;
    }
}
```

BankAccountApp.java



```java
/**
 * Name: BankAccountApp.java
 * Written by:
 * Written on:
 * Purpose:
 */

public class BankAccountApp {
    Run | Debug
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount("Paul Bunyan", 5000);
        BankAccount bobsAccount = new BankAccount("Bob Robinson");

        bobsAccount.deposit(500);
        System.out.println("Owner: " + bobsAccount.getOwner());
        System.out.println("Balance: " + bobsAccount.getBalance());

        bobsAccount.withdraw(1000); // warning!
        System.out.println("Owner: " + bobsAccount.getOwner());
        System.out.println("Balance: " + bobsAccount.getBalance());
        System.out.println();

        System.out.println("Owner: " + myAccount.getOwner());
        System.out.println("Balance: " + myAccount.getBalance());
        System.out.println();

        System.out.println("Deposit 1000?");
        myAccount.deposit(1000);

        System.out.println("Owner: " + myAccount.getOwner());
        System.out.println("Balance: " + myAccount.getBalance());
    }
}
```

Example run:



```
Owner: Bob Robinson
Balance: 500
The amount to deposit must be greater than 0 and less than your balance.
Owner: Bob Robinson
Balance: 500

Owner: Paul Bunyan
Balance: 5000

Deposit 1000
Owner: Paul Bunyan
Balance: 6000
```

Designing an Object-Oriented Program

The first step is to identify the classes the program will need.

1. Identify the real-world objects, the nouns.

   Customer
   Address
   Car
   labor charges
   Toyota

   Some nouns contain other nouns. A Car can also be a Toyota. A customer would have an address. We need a Car and a Customer class.

2. Determine what describes or makes up the class, the data attributes, adjectives.

   Customer: Name, Address, Phone

3. Determine the actions, the verbs, the methods.

   moveForward()
   makePurchase()
   createInvoice()

   Customer: setName(), setAddress(), getAddress()

Keep everything that is relevant to the class in the class.

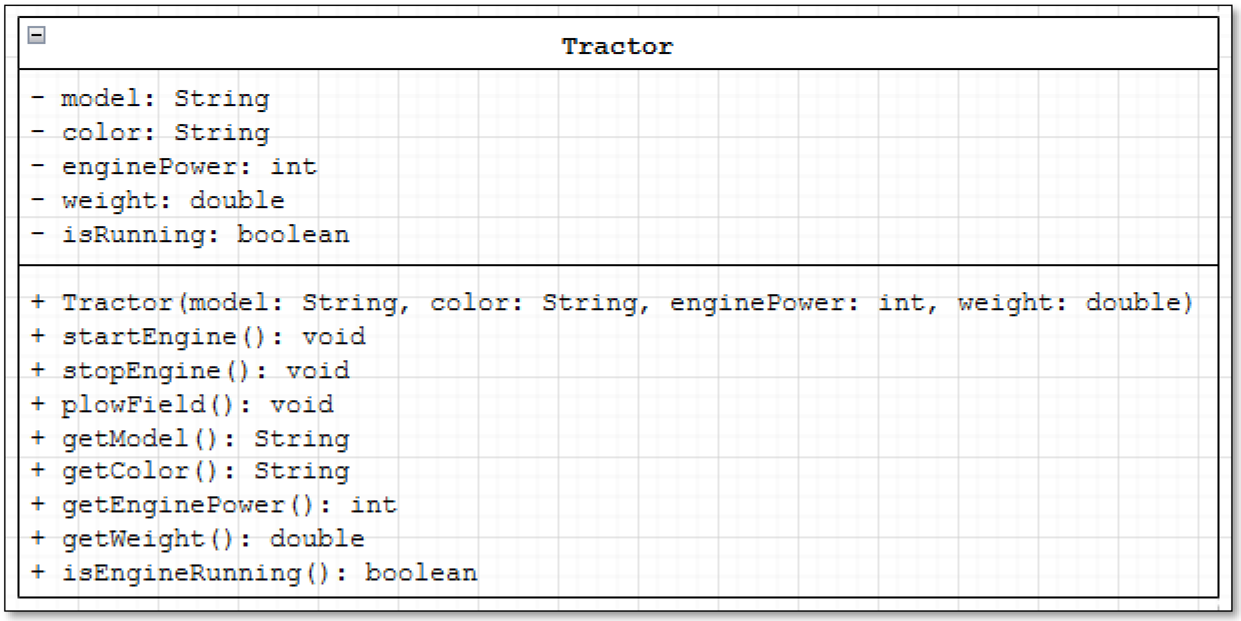
UML Diagram

A UML (Unified Modeling Language) diagram is a visual representation of a software system's structure and behavior. It's a way to show how different parts of a system interact with each other. Here is a brief explanation suitable for a beginner:

Key Elements of a UML Class Diagram

1. Class Name:
    1. The name of the class is written at the top of the diagram inside a rectangle.
2. Attributes:
    1. These are the properties or data members of the class.
    2. Listed below the class name in a rectangle.
    3. Represented with their data types, e.g., model: String, color: String.
3. Methods:
    1. These are the functions or behaviors of the class.
    2. Listed below the attributes in the rectangle.
    3. Represented with their return types and parameters, e.g., startEngine(): void.
4. Visibility:
    1. Indicates who can access the attribute or method.
    2. + for public (accessible by anyone).
    3. - for private (accessible only within the class).
    4. # for protected (accessible within the class and its subclasses).

This example was created using draw.io



Assignment 1: Design, Create, and Use a Class

Create a UML diagram for a vehicle class. Pick any type of vehicle you wish.

Create your own vehicle.

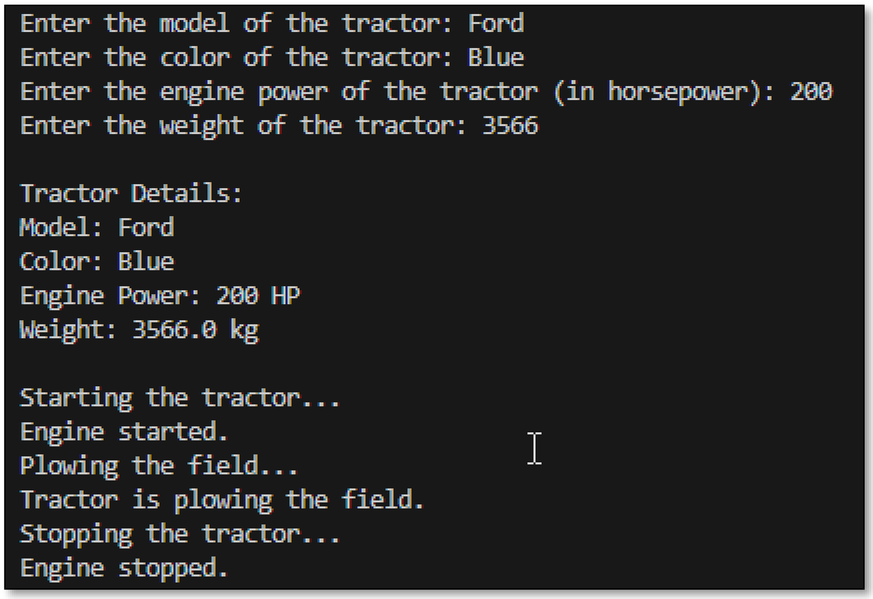
Use the UML diagram to create a small program that demonstrates this class.

Requirements

These are minimum requirements. You can go as far as you wish.

- 2 private data attributes
- Getters and setters for each
- 2 methods

Example run:



```
Enter the model of the tractor: Ford
Enter the color of the tractor: Blue
Enter the engine power of the tractor (in horsepower): 200
Enter the weight of the tractor: 3566

Tractor Details:
Model: Ford
Color: Blue
Engine Power: 200 HP
Weight: 3566.0 kg

Starting the tractor...
Engine started.
Plowing the field...
Tractor is plowing the field.
Stopping the tractor...
Engine stopped.
```

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.