OOP Pillers

- Encapsulation
  - Encapsulation is a method of **limiting** direct access to some components of an entity. Encapsulation can be used to conceal both data members and data functions or methods related to an instantiated class or object by using public, private and protected keywords

OOP Pillers

- Inheritance
  - It is the process in Java that allows one class to inherit the features (fields and methods) of another, resulting in the creation of new classes from existing ones. A class that inherits from another class may reuse its methods and fields. In addition, you can modify your existing class by adding additional fields and methods. Example Sedan extends Car, and Car extends Vehicle, hence Sedan inherits from the Vehicle class.
    - Vehicle ==> Car ==> Sedan ==> Accord

OOP Pillers

- Polymorphism
  - "Poly" means many and "morphs" means forms
  - Perform a single action in various ways.
  - Overloading – Different signature
    - public void sum(int x, int y);
    - public void sum(int x, int y, int z);
    - public void sum(int x, int y, int z, int a)
  - Overriding – Same Signature
    - Base Class = public void fun(int x);
    - Child Class = public void fun(int x);

What is OOP

- Object Oriented Programming is faster
- OOP establishes a clear framework for programming.
- OOP promotes DRY (Don't Repeat Yourself) principal.
- Code written using OOP is fully reusable

Different between Classes and Objects

Classes (Blueprint, Templates)	Objects (Instances)
Fruit	Apple, Mango, Grapes
University	Clark, Assumption, WPI
Car	Civic, Camery, Mustang
Person	Jack, John, Tom, Bill

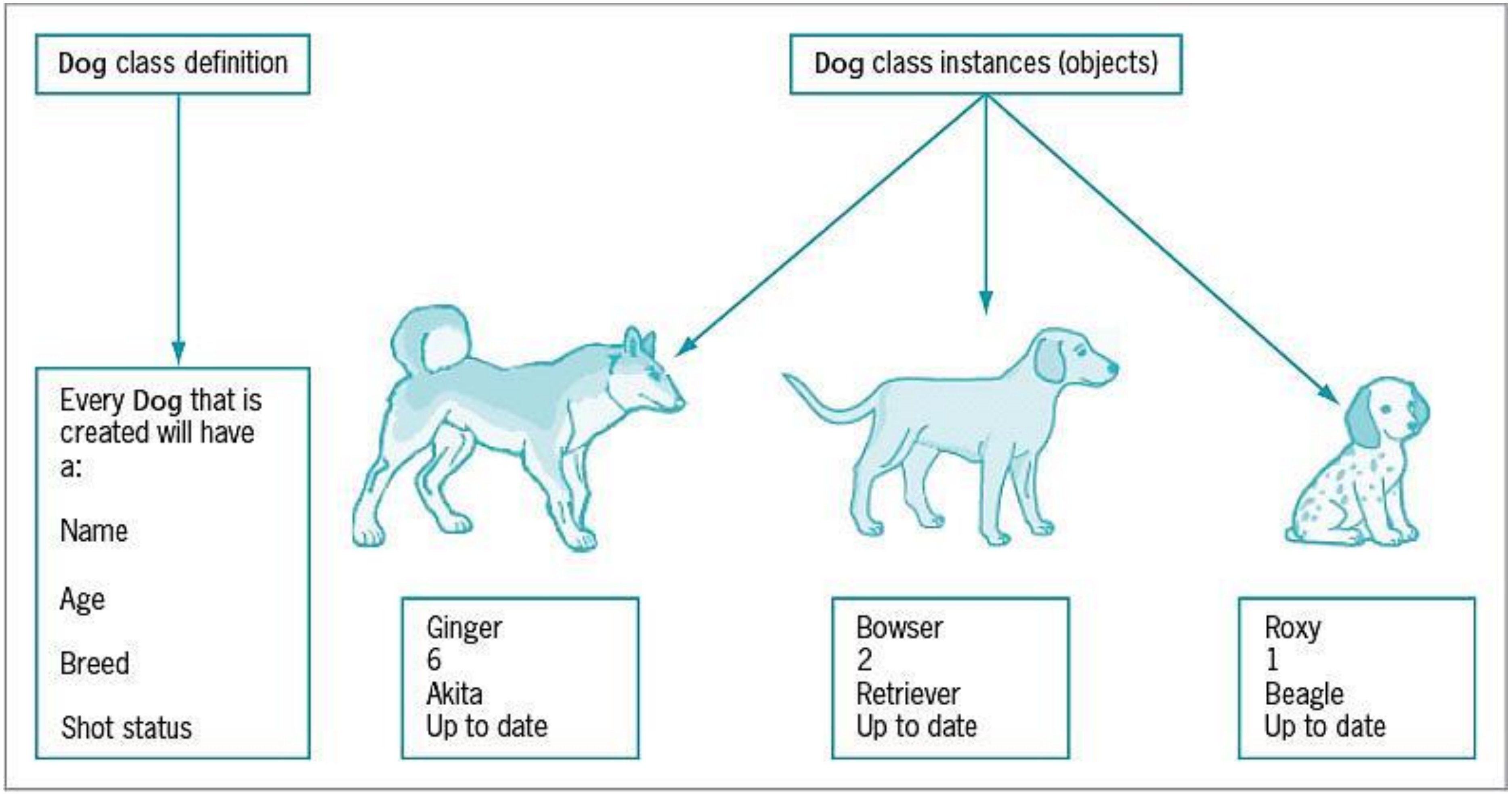
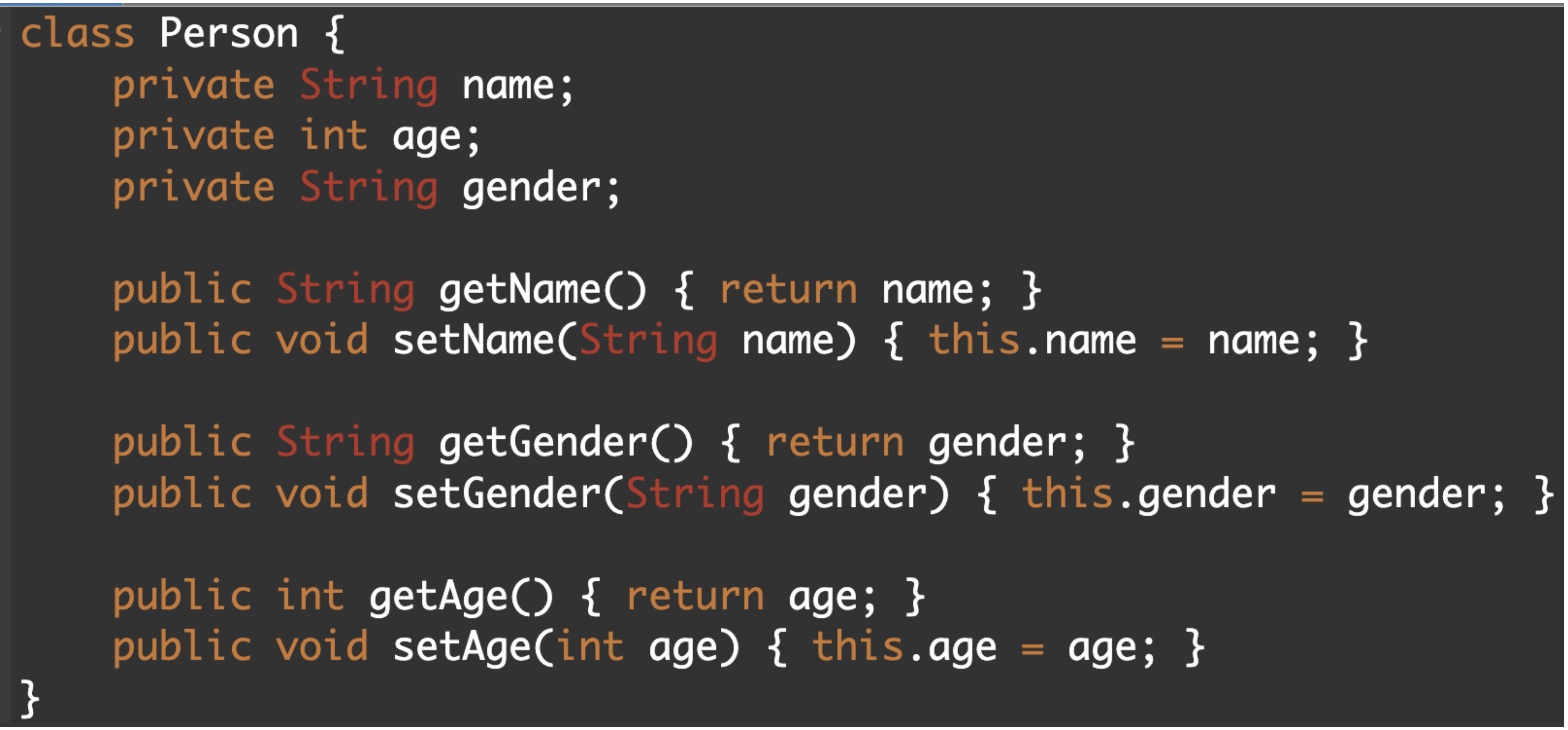


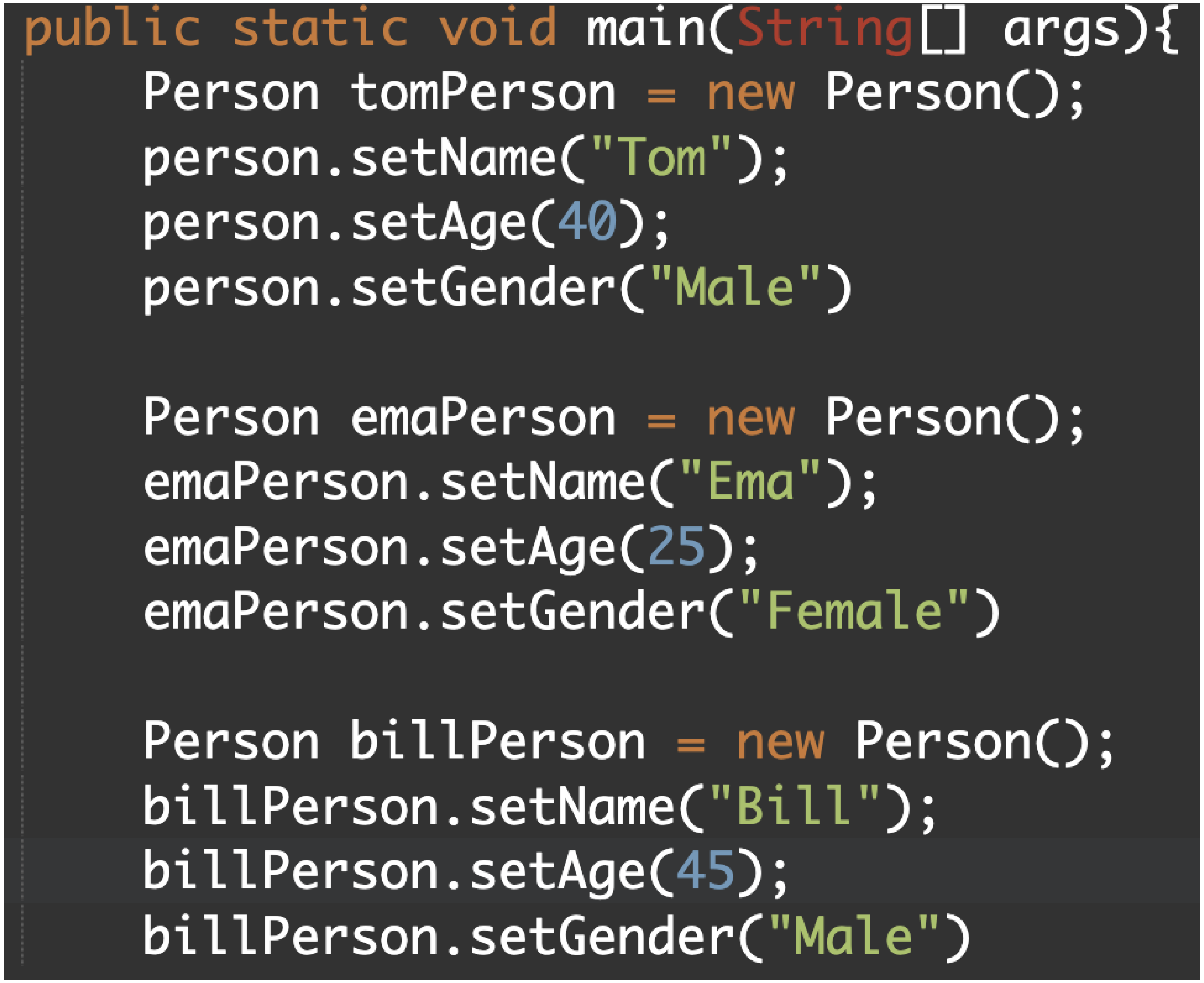
Figure 1-2 Dog class definition and some objects created from it

Class Blueprint



```java
class Person {
    private String name;
    private int age;
    private String gender;

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

    public String getGender() { return gender; }
    public void setGender(String gender) { this.gender = gender; }

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

Objects



```java
public static void main(String[] args){
    Person tomPerson = new Person();
    person.setName("Tom");
    person.setAge(40);
    person.setGender("Male")

    Person emaPerson = new Person();
    emaPerson.setName("Ema");
    emaPerson.setAge(25);
    emaPerson.setGender("Female")

    Person billPerson = new Person();
    billPerson.setName("Bill");
    billPerson.setAge(45);
    billPerson.setGender("Male")
```

Classes

A Class is a collection of fields and methods

Each class should be unique

Objects are created dynamically using **new** operator

State: Attributes of an object. Eg Dog Color, Age, Breed

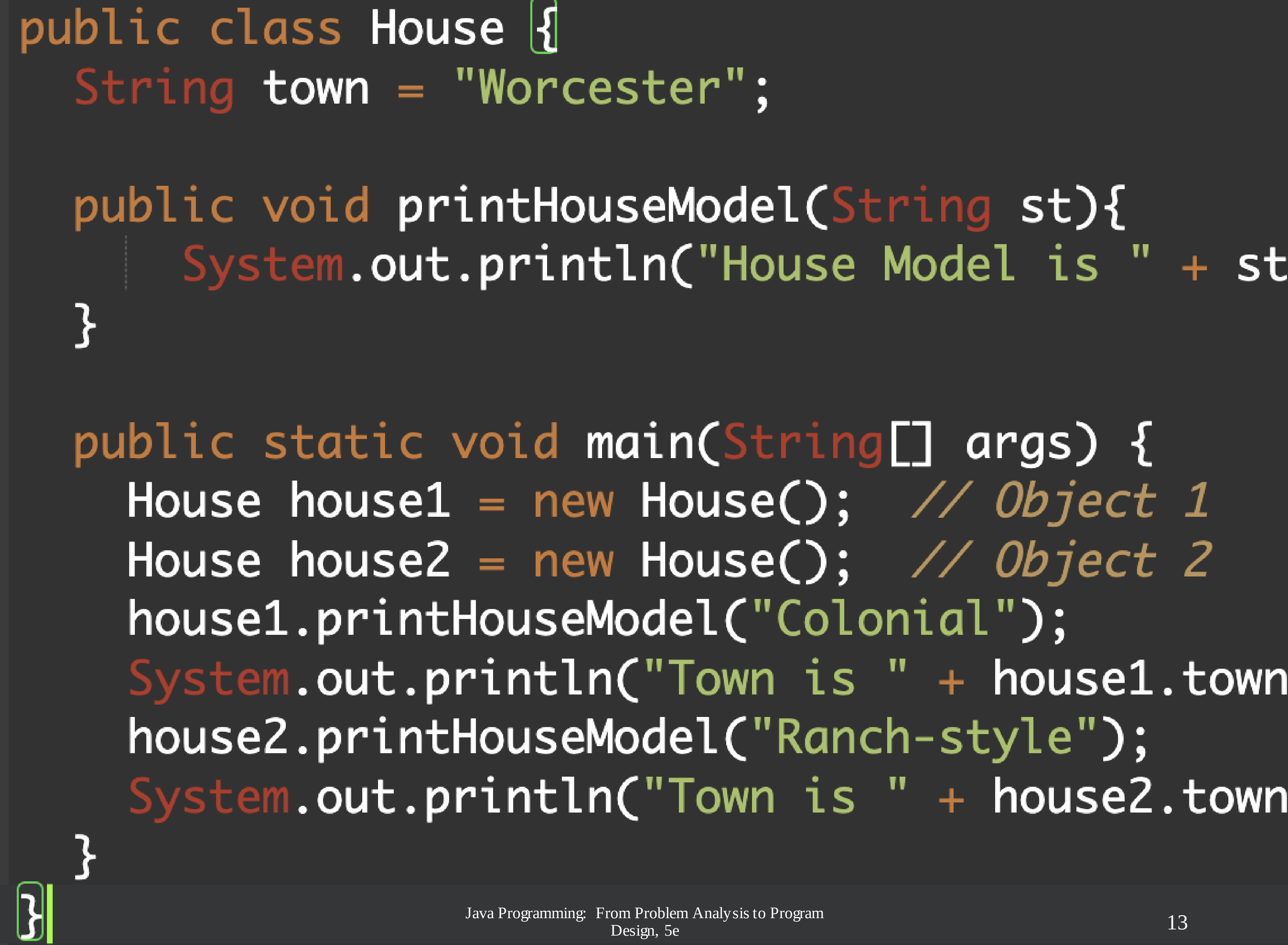
Behavior: methods of an object. Eg bark(), sleep(), Eat()

Identity: It gives a unique name to an object

When individual objects are formed, they inherit the class's variables and methods.

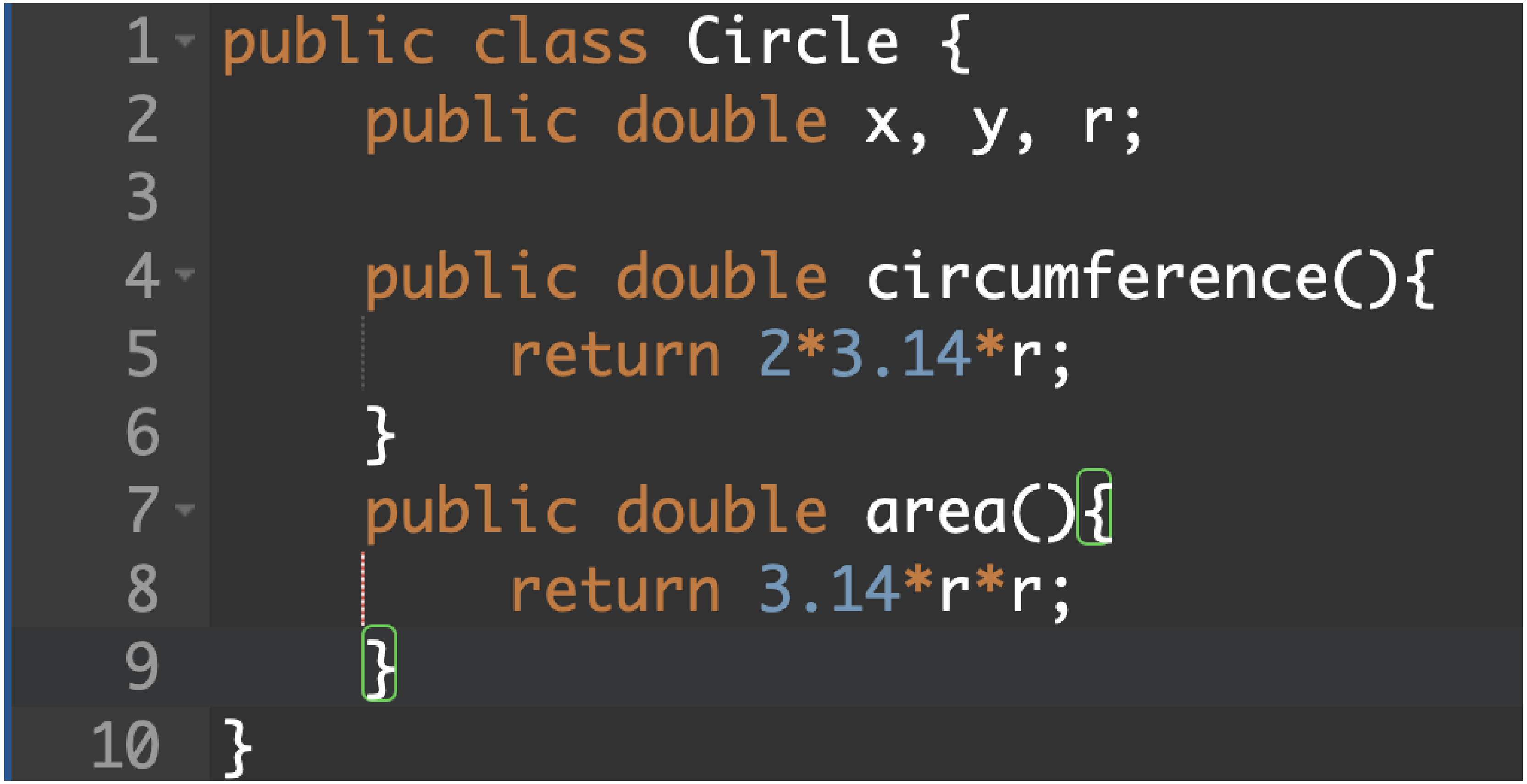
- Set or Change the values are called mutator methods
- Get the values are called accessor methods
- Methods inside the class (object) are not static
- If method occur once per class, should be static
- If method occur per object, should not be static

- public class MyClass{
-     public static pubStatMethod();
-     private static privStatMethod();
-     public pubNonStatMethod();
-     private privNonStatMethod();
- }



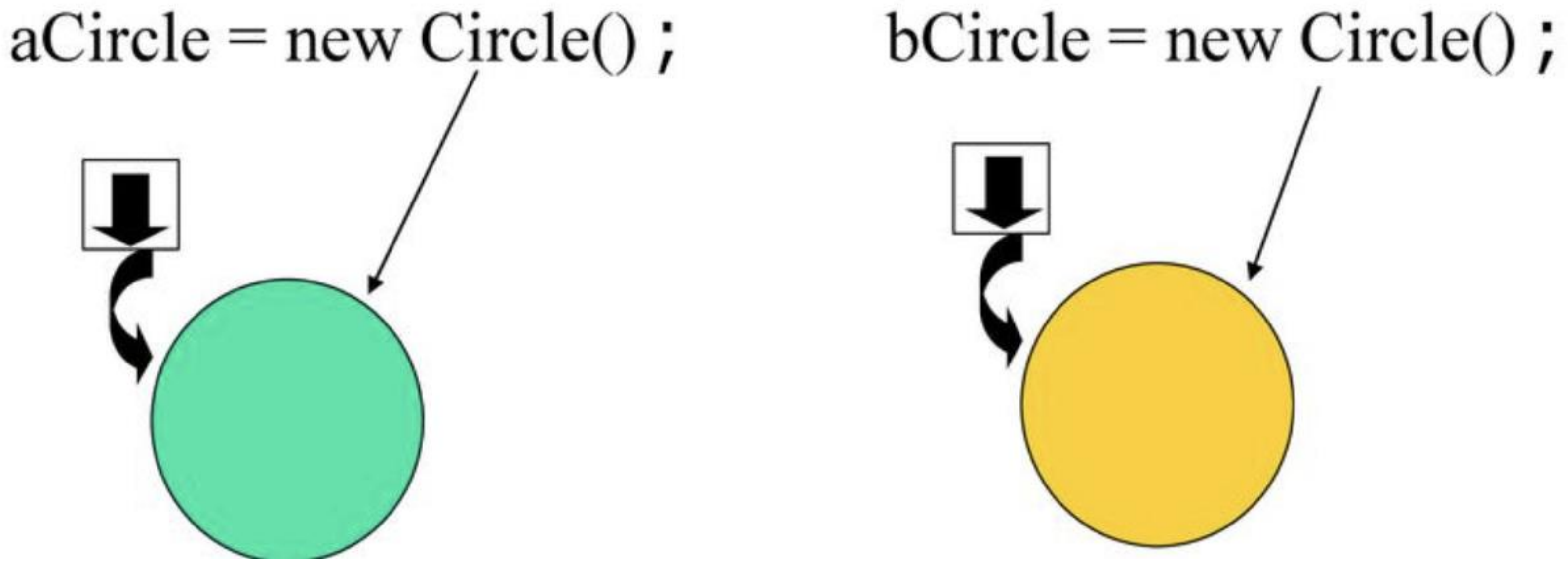
```
public class House {
  String town = "Worcester";

  public void printHouseModel(String st){
      System.out.println("House Model is " + st
  }

  public static void main(String[] args) {
    House house1 = new House();  // Object 1
    House house2 = new House();  // Object 2
    house1.printHouseModel("Colonial");
    System.out.println("Town is " + house1.town
    house2.printHouseModel("Ranch-style");
    System.out.println("Town is " + house2.town
  }
}
```

Example of a Circle Class

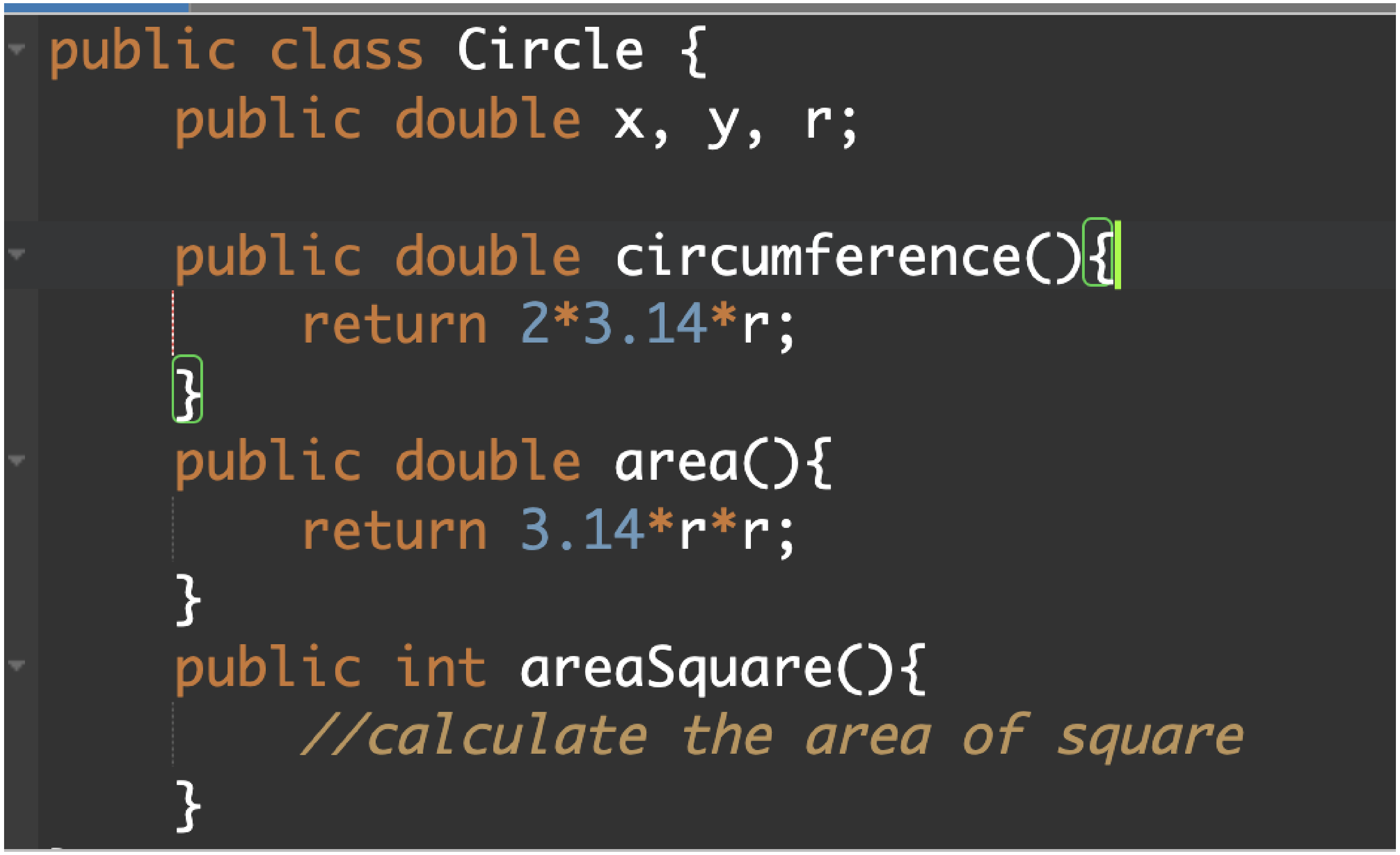


```
public class Circle {
        public double x, y, r;

        public double circumference(){
                return 2*3.14*r;
        }
        public double area(){
                return 3.14*r*r;
        }
}
```

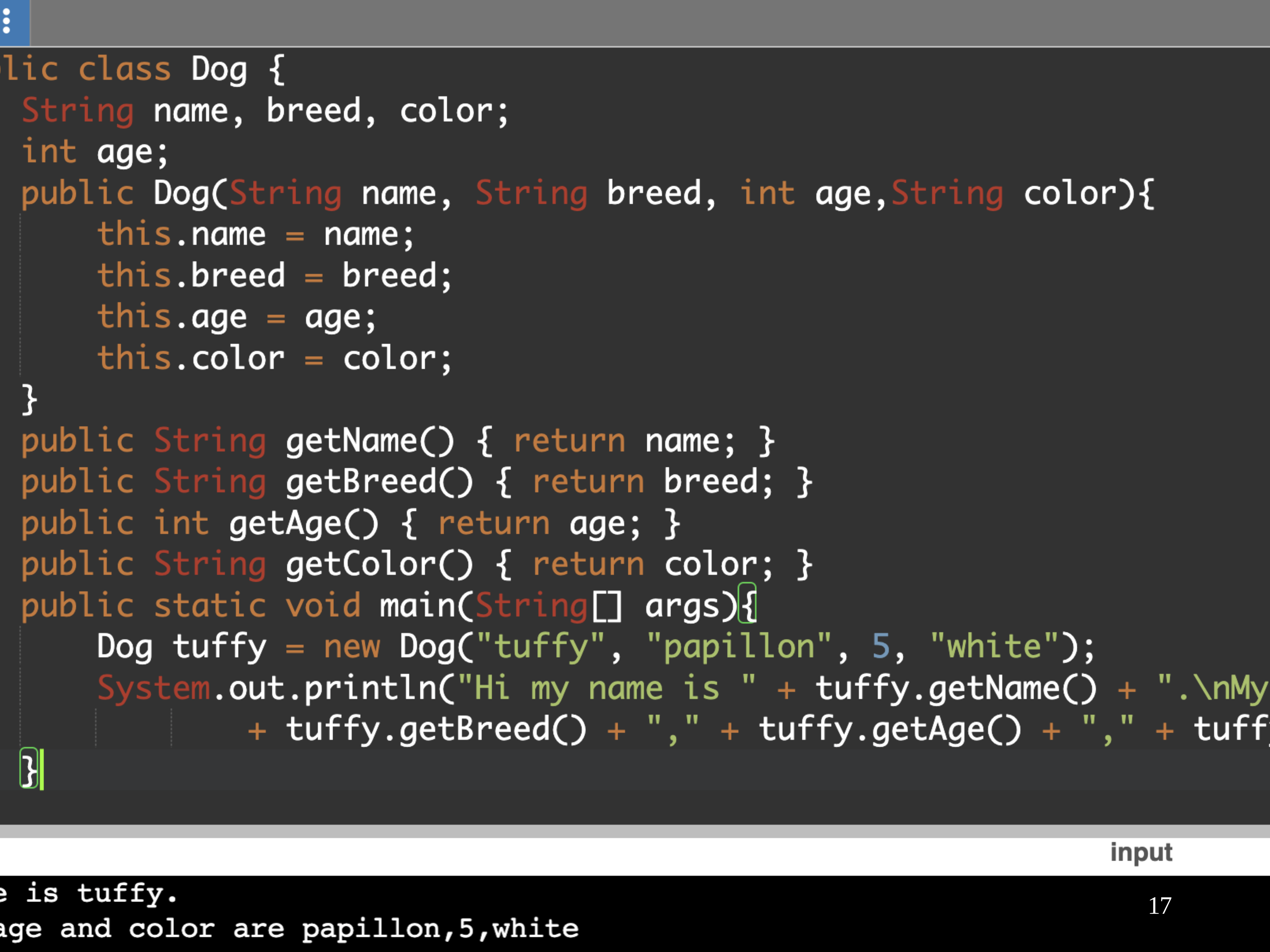
Circle Object



Invalid example : Square and Circle

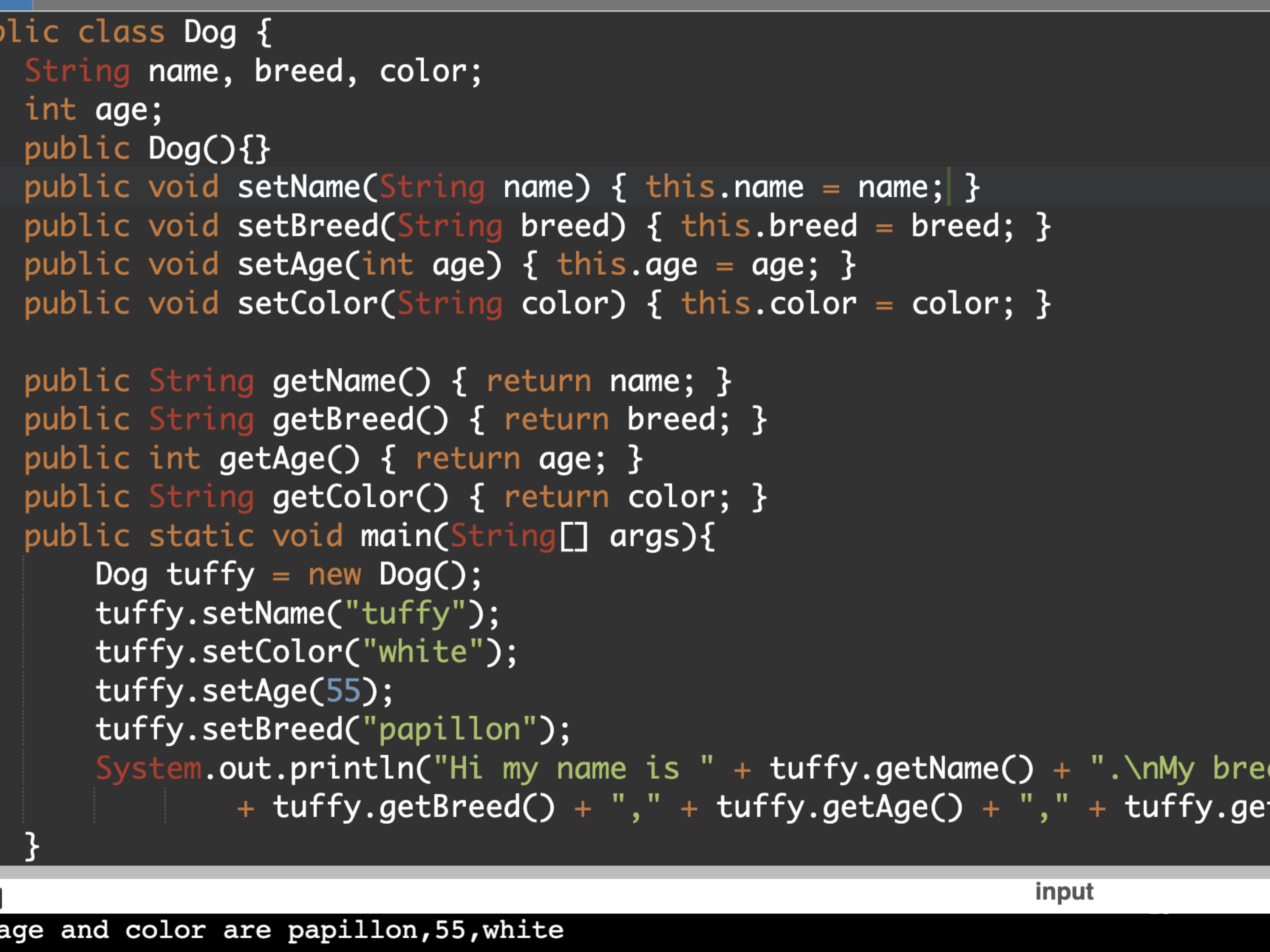


```
public class Circle {
    public double x, y, r;

    public double circumference(){
        return 2*3.14*r;
    }
    public double area(){
        return 3.14*r*r;
    }
    public int areaSquare(){
        //calculate the area of square
    }
```



```
lic class Dog {
 String name, breed, color;
 int age;
 public Dog(String name, String breed, int age,String color){
     this.name = name;
     this.breed = breed;
     this.age = age;
     this.color = color;
 }
 public String getName() { return name; }
 public String getBreed() { return breed; }
 public int getAge() { return age; }
 public String getColor() { return color; }
 public static void main(String[] args){
     Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
     System.out.println("Hi my name is " + tuffy.getName() + ".\nMy
             + tuffy.getBreed() + "," + tuffy.getAge() + "," + tuff
 }
```

```
e is tuffy.
age and color are papillon,5,white
```



```
lic class Dog {
  String name, breed, color;
  int age;
  public Dog(){}
  public void setName(String name) { this.name = name; }
  public void setBreed(String breed) { this.breed = breed; }
  public void setAge(int age) { this.age = age; }
  public void setColor(String color) { this.color = color; }

  public String getName() { return name; }
  public String getBreed() { return breed; }
  public int getAge() { return age; }
  public String getColor() { return color; }
  public static void main(String[] args){
      Dog tuffy = new Dog();
      tuffy.setName("tuffy");
      tuffy.setColor("white");
      tuffy.setAge(55);
      tuffy.setBreed("papillon");
      System.out.println("Hi my name is " + tuffy.getName() + ".\nMy bre
              + tuffy.getBreed() + "," + tuffy.getAge() + "," + tuffy.ge
  }
```

```
age and color are papillon,55,white
```

Encapsulations – Data Hiding

Data hiding restricts access by hiding implementation details. Ex. CalculateTax()

The user will have no notion how the class is implemented internally. The user will not be able to see how the class stores values in variables.

The user will simply understand that we are sending the values to a setter method and that variables are being initialized with that value.

Encapsulations - Flexibility

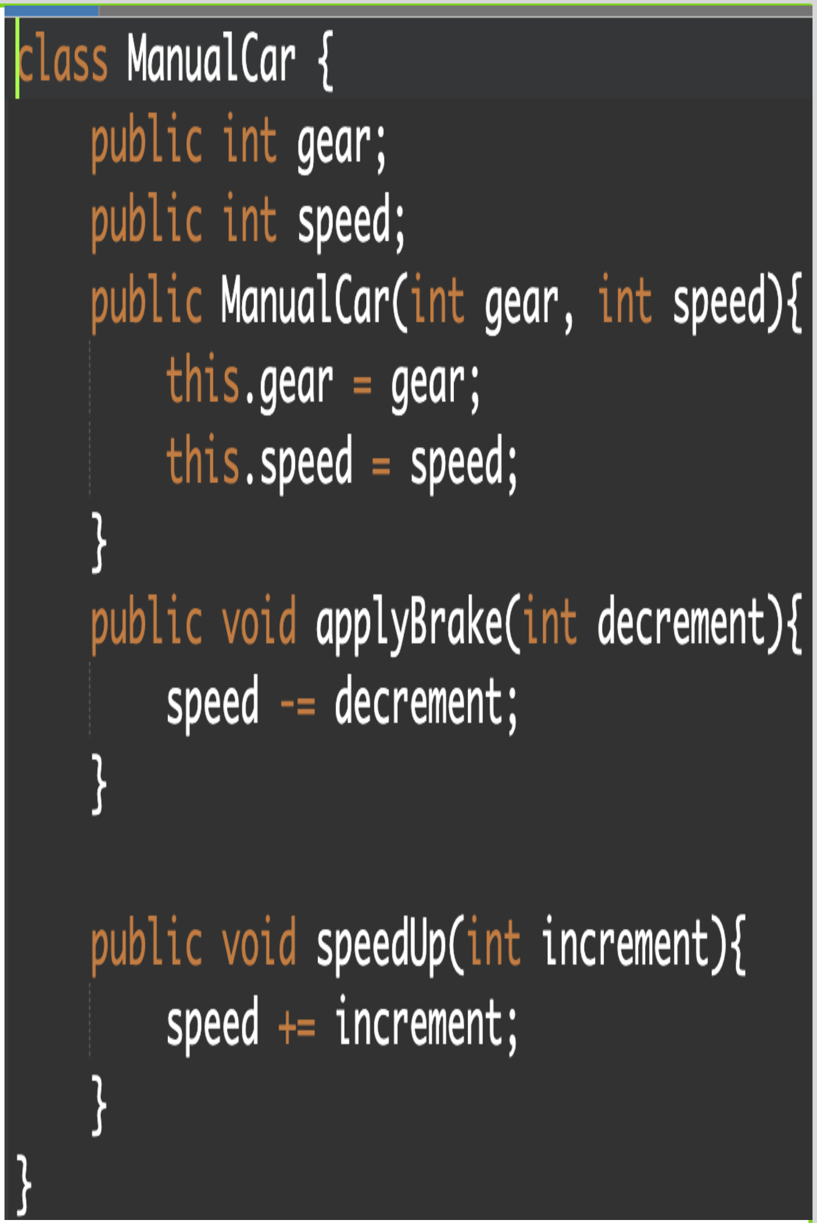
- Class variables can be read or written based on requirements. Read-only meaning there are no setter methods such as setName(), setAge(), etc. Only get methods like getName(), getAge()

Inheritance - Terminology

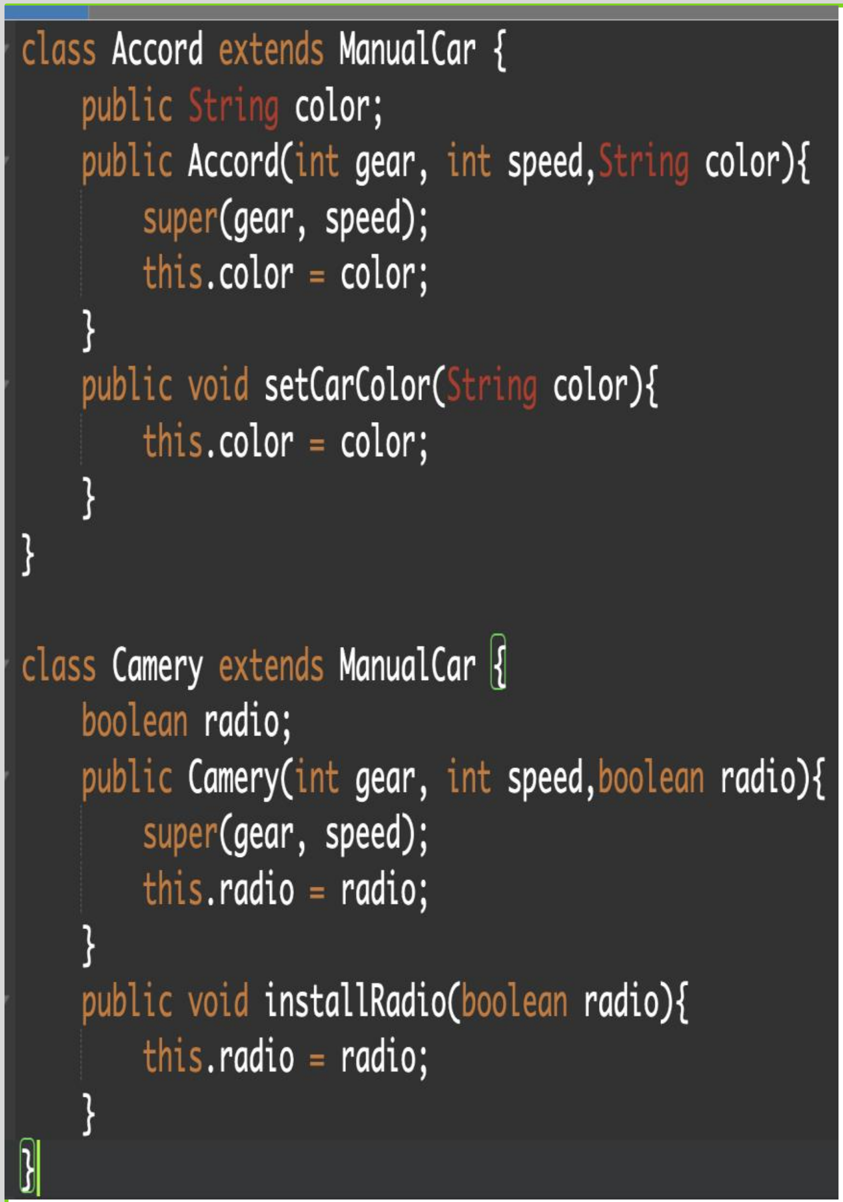
- A typical **class** has characteristics/behavior and shared properties/attributes. It is simply a template or blueprint.
- A **superclass**, often called a base class or parent class, is the class from which features are inherited.
- A **subclass**, also known as a derived class, extended class, or child class, is the class that inherits from the parent class. In addition to the fields and methods provided by the superclass, the subclass may include its own.
- In Java, the **extends** keyword is used to specify inheritance. Using the extends keyword shows that you are deriving from an existing class.

Inheritance Type

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance



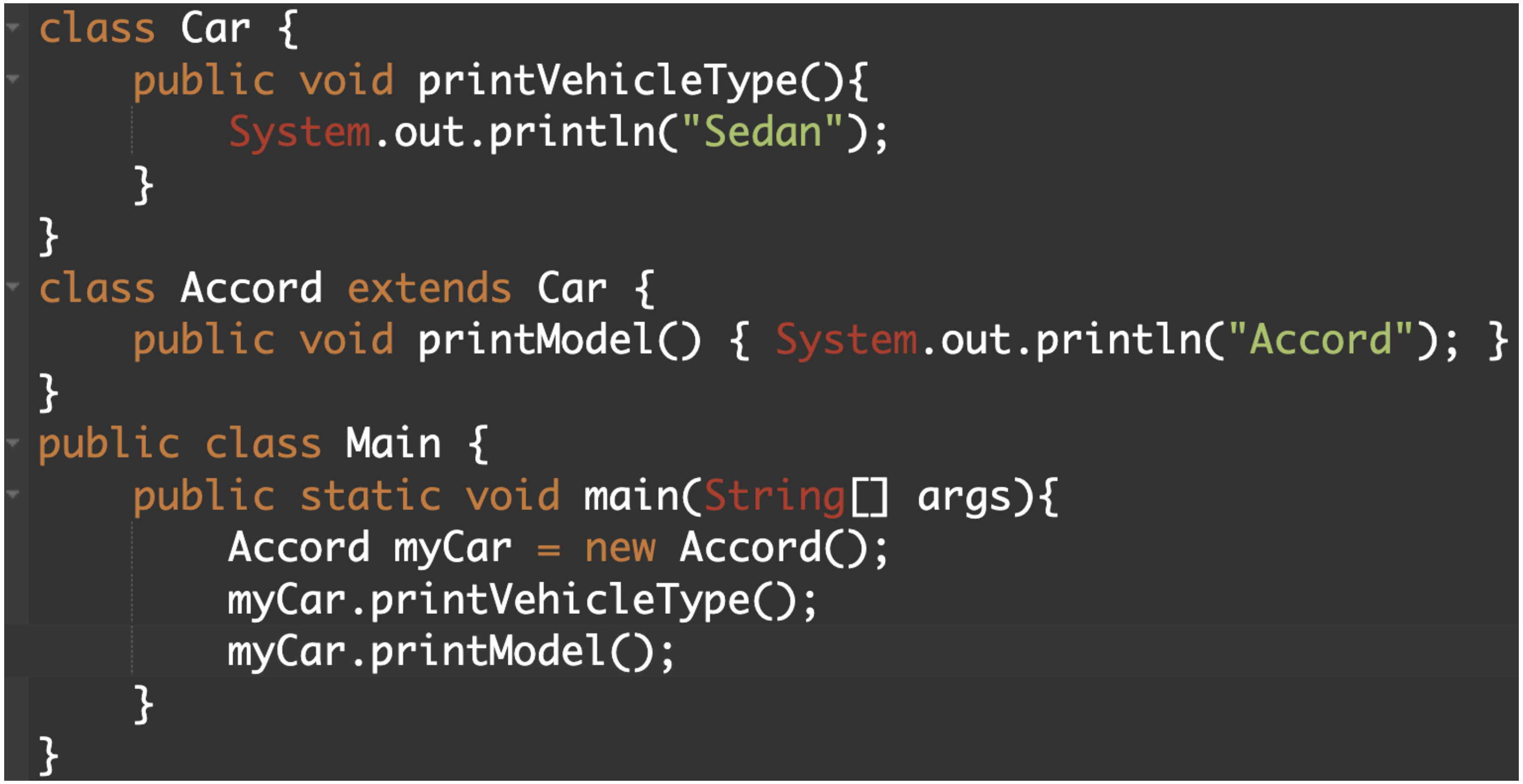
```
class ManualCar {
    public int gear;
    public int speed;
    public ManualCar(int gear, int speed){
        this.gear = gear;
        this.speed = speed;
    }
    public void applyBrake(int decrement){
        speed -= decrement;
    }

    public void speedUp(int increment){
        speed += increment;
    }
}
```



```
class Accord extends ManualCar {
    public String color;
    public Accord(int gear, int speed,String color){
        super(gear, speed);
        this.color = color;
    }
    public void setCarColor(String color){
        this.color = color;
    }
}

class Camery extends ManualCar {
    boolean radio;
    public Camery(int gear, int speed,boolean radio){
        super(gear, speed);
        this.radio = radio;
    }
    public void installRadio(boolean radio){
        this.radio = radio;
    }
}
```

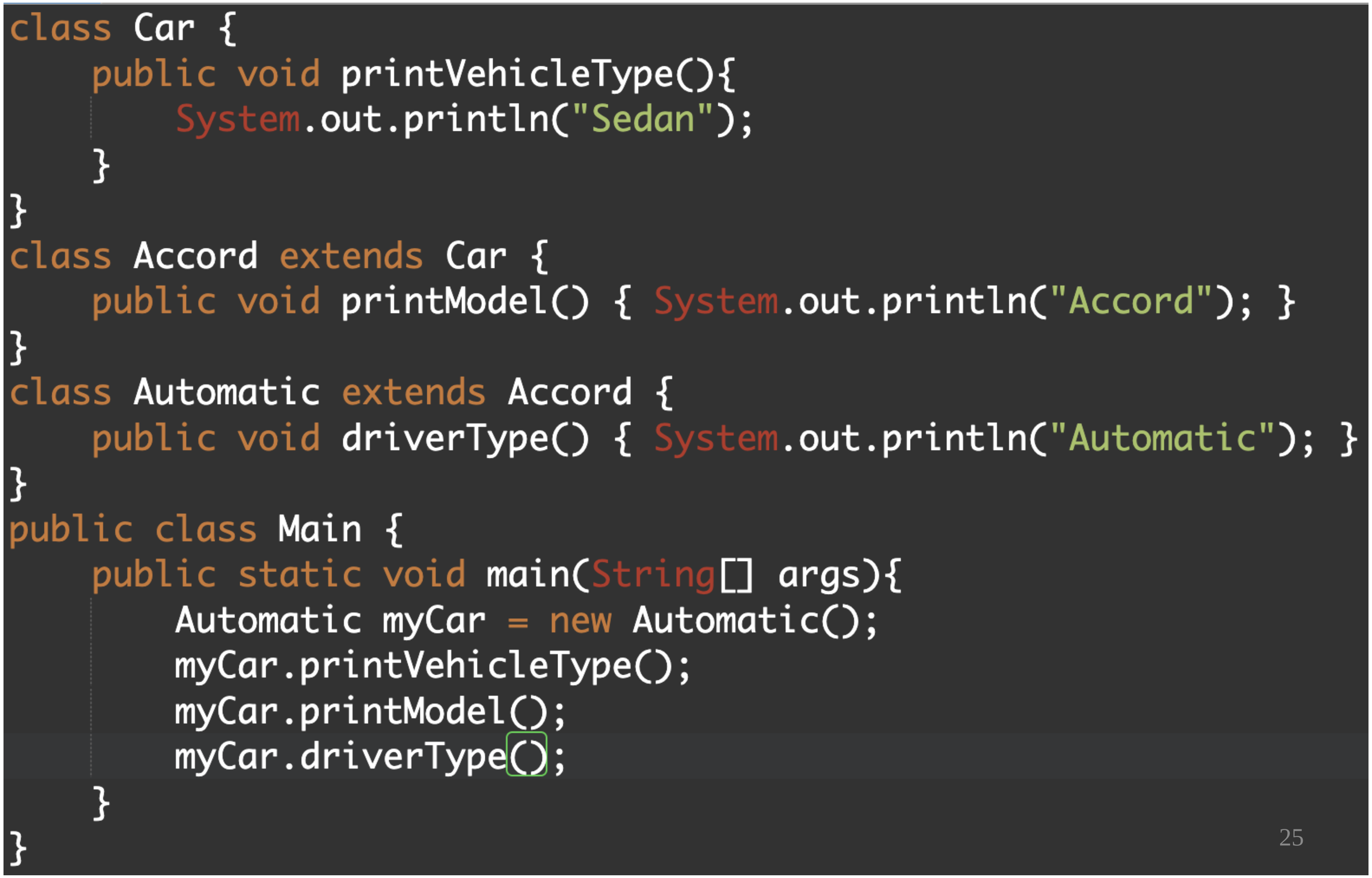
Single Inheritance

- Can only inherit from one Super class



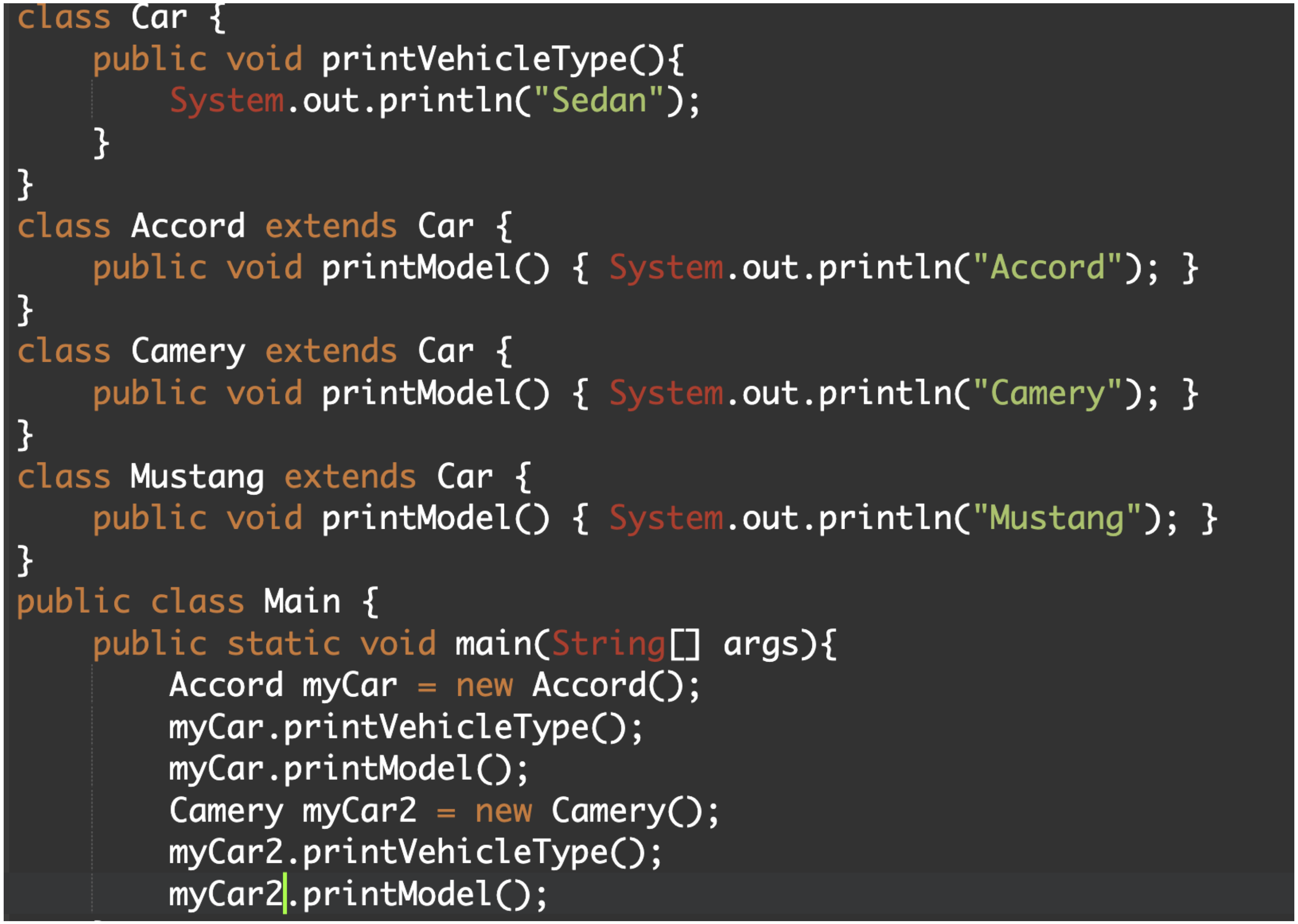
```java
class Car {
    public void printVehicleType(){
        System.out.println("Sedan");
    }
}
class Accord extends Car {
    public void printModel() { System.out.println("Accord"); }
}
public class Main {
    public static void main(String[] args){
        Accord myCar = new Accord();
        myCar.printVehicleType();
        myCar.printModel();
    }
}
```

Multilevel Inheritance



```
class Car {
    public void printVehicleType(){
        System.out.println("Sedan");
    }
}
class Accord extends Car {
    public void printModel() { System.out.println("Accord"); }
}
class Automatic extends Accord {
    public void driverType() { System.out.println("Automatic"); }
}
public class Main {
    public static void main(String[] args){
        Automatic myCar = new Automatic();
        myCar.printVehicleType();
        myCar.printModel();
        myCar.driverType();
    }
}
```

Heretical Inheritance



```
class Car {
    public void printVehicleType(){
        System.out.println("Sedan");
    }
}
class Accord extends Car {
    public void printModel() { System.out.println("Accord"); }
}
class Camery extends Car {
    public void printModel() { System.out.println("Camery"); }
}
class Mustang extends Car {
    public void printModel() { System.out.println("Mustang"); }
}
public class Main {
    public static void main(String[] args){
        Accord myCar = new Accord();
        myCar.printVehicleType();
        myCar.printModel();
        Camery myCar2 = new Camery();
        myCar2.printVehicleType();
        myCar2.printModel();
```

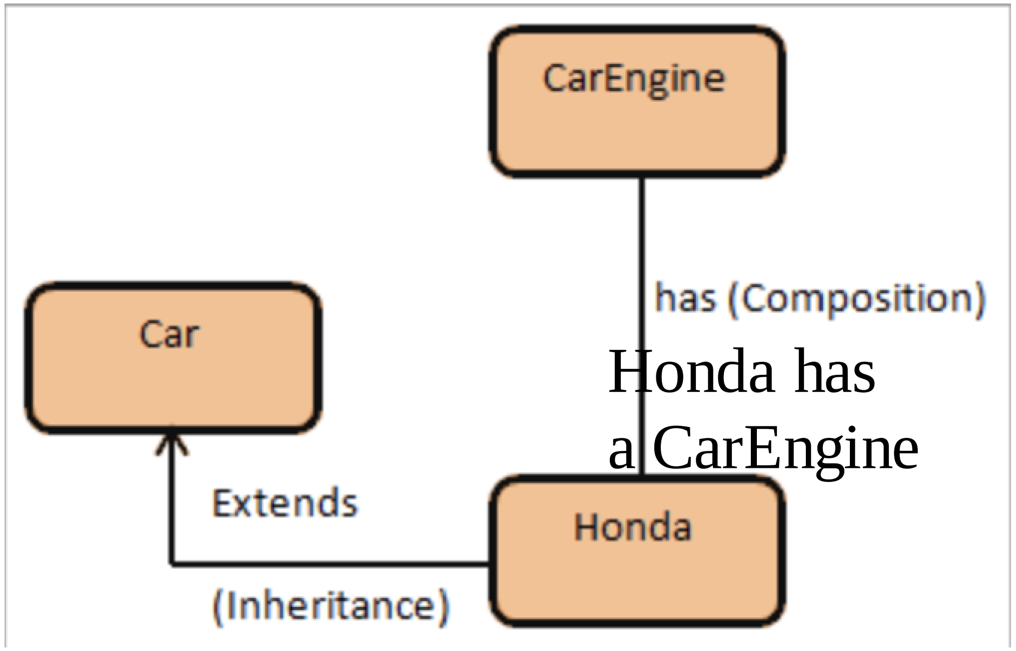
Multiple Inheritance

- Java does NOT support multiple Inheritance (C++ does).
- Use Interfaces as a workaround

Hybrid Inheritance

- Is-a relationship
  - Base class is Animal and child class is Dog
  - Dog is-a Animal makes sense
  - Dog has-a Animal does not make sense
  - The is-a relationship allows the child class to inherit the characteristics and behaviors of the base class.

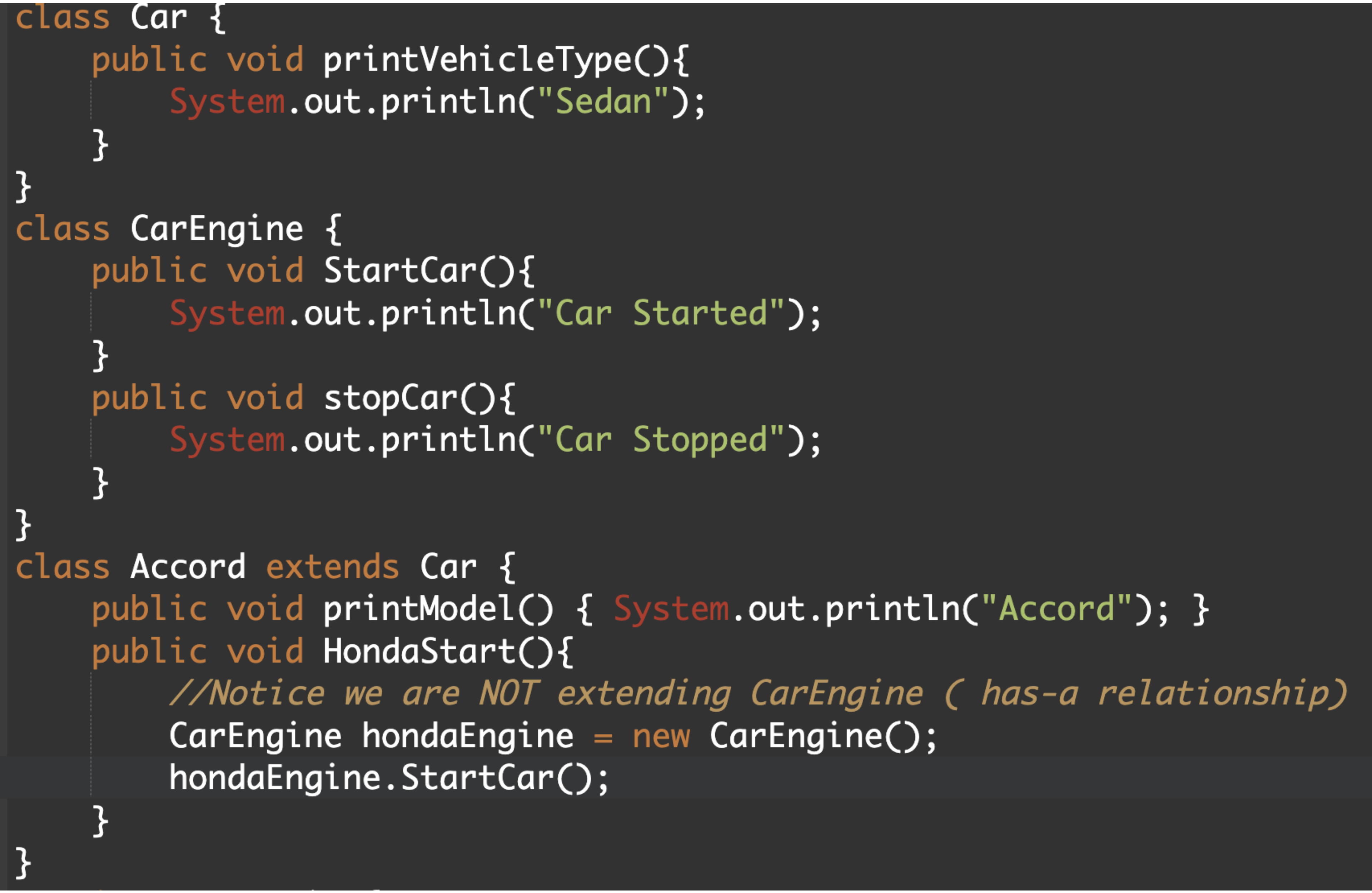
Hybrid Inheritance



Honda has a CarEngine

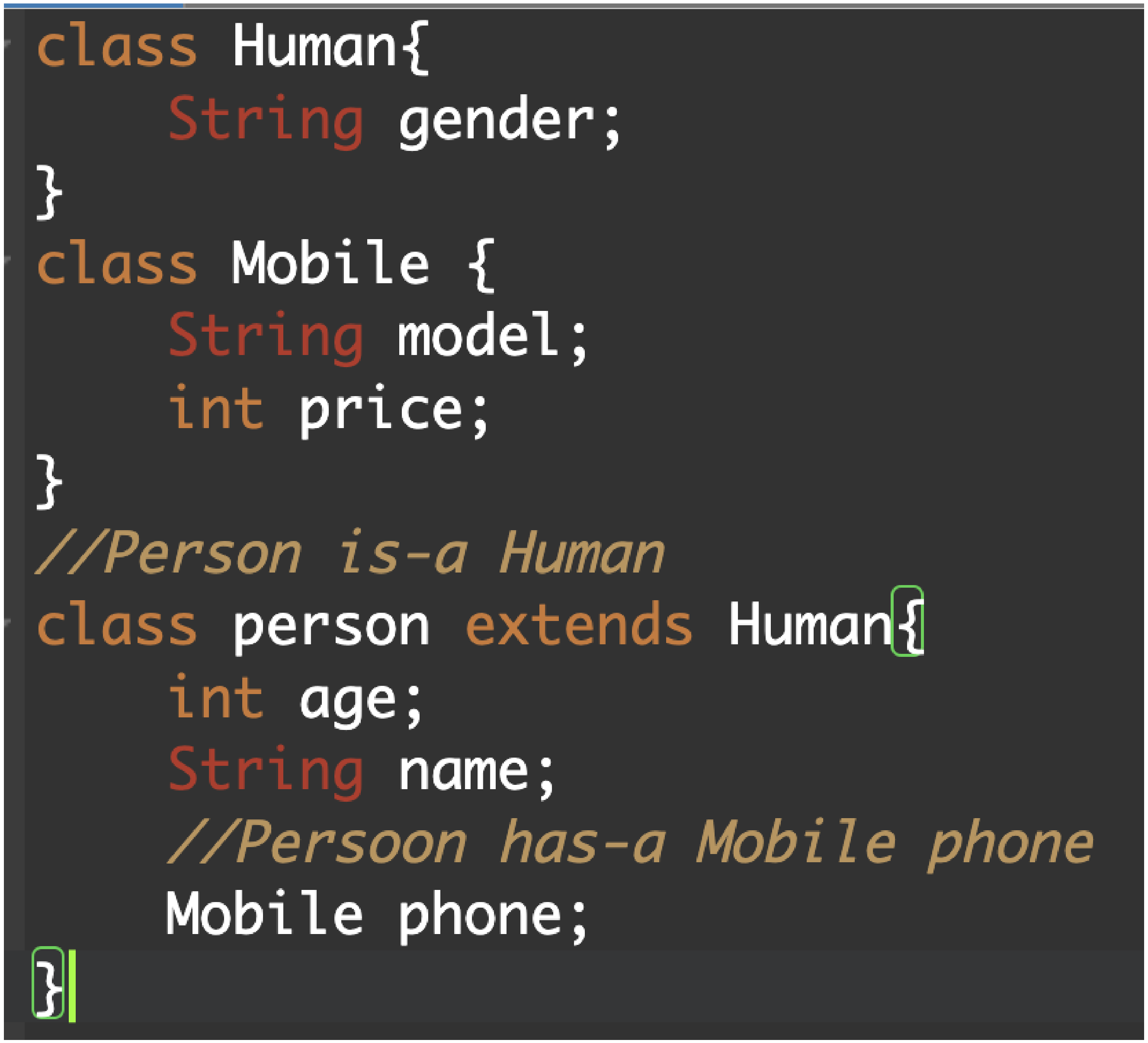
Honda is a Car

- Has-a relationship (Composition)
  - Car has-a Engine makes sense
  - Car is-a Engine does not make sense



```java
class Car {
    public void printVehicleType(){
        System.out.println("Sedan");
    }
}
class CarEngine {
    public void StartCar(){
        System.out.println("Car Started");
    }
    public void stopCar(){
        System.out.println("Car Stopped");
    }
}
class Accord extends Car {
    public void printModel() { System.out.println("Accord"); }
    public void HondaStart(){
        //Notice we are NOT extending CarEngine ( has-a relationship)
        CarEngine hondaEngine = new CarEngine();
        hondaEngine.StartCar();
    }
}
```

Another example



```
class Human{
	String gender;
}
class Mobile {
	String model;
	int price;
}
//Person is-a Human
class person extends Human{
	int age;
	String name;
	//Persoon has-a Mobile phone
	Mobile phone;
}
```

Is-A

- Simply be achieved by using **extends** Keyword
- Code reusability is a huge benefit
- Unidirectional -> Bulb is a device but device is NOT a bulb, cause TV is also a device
- Tightly coupled, which means changing one entity will affect another entity

Quick Q&A

- You use the keyword **inherits** to achieve inheritance in Java
- A derived class has access to all its parent nonprivate methods?
- Subclasses are more specific than the superclass they extend?
- When you use inheritance, you save time and reduce errors?

Classes (continued)

- `class`: reserved word; collection of a fixed number of components
- Components: members of a class
- Members accessed by name
- Class categories/modifiers
  - `private`
  - `protected`
  - `public`

Classes (continued)

- `private`: members of class are not accessible outside class
- `public`: members of class are accessible outside class
- Class members: can be methods or variables
- Variable members declared like any other variables

Constructors

- Two types of constructors
    - With parameters
    - Without parameters (**default constructor)**

Constructors (continued)

- Constructors have the following properties:
  - The name of a constructor is the same as the name of the class
  - A constructor, even though it is a method, has no type
  - A class can have more than one constructor; all constructors of a class have the same name
  - If a class has more than one constructor, any two constructors must have different *signatures*
  - Constructors are automatically executed when a class object is instantiated
  - If there are multiple constructors, which constructor executes depends on the type of values passed to the class object when the class object is instantiated

`class Clock`: Constructors

- Default constructor is:
  - `public Clock()`
- Constructor with parameters is:
  - `public Clock(int hours, int minutes, int seconds)`

Variable Declaration and Object Instantiation

- The general syntax for using the operator `new` is:

```
new className()
```

or:

```
new className(argument1, argument2, ..., argumentN)
```

```
Clock myClock;
Clock yourClock;

myClock = new Clock();
yourClock = new Clock(9, 35, 15);
```

Variable Declaration and Object Instantiation (continued)

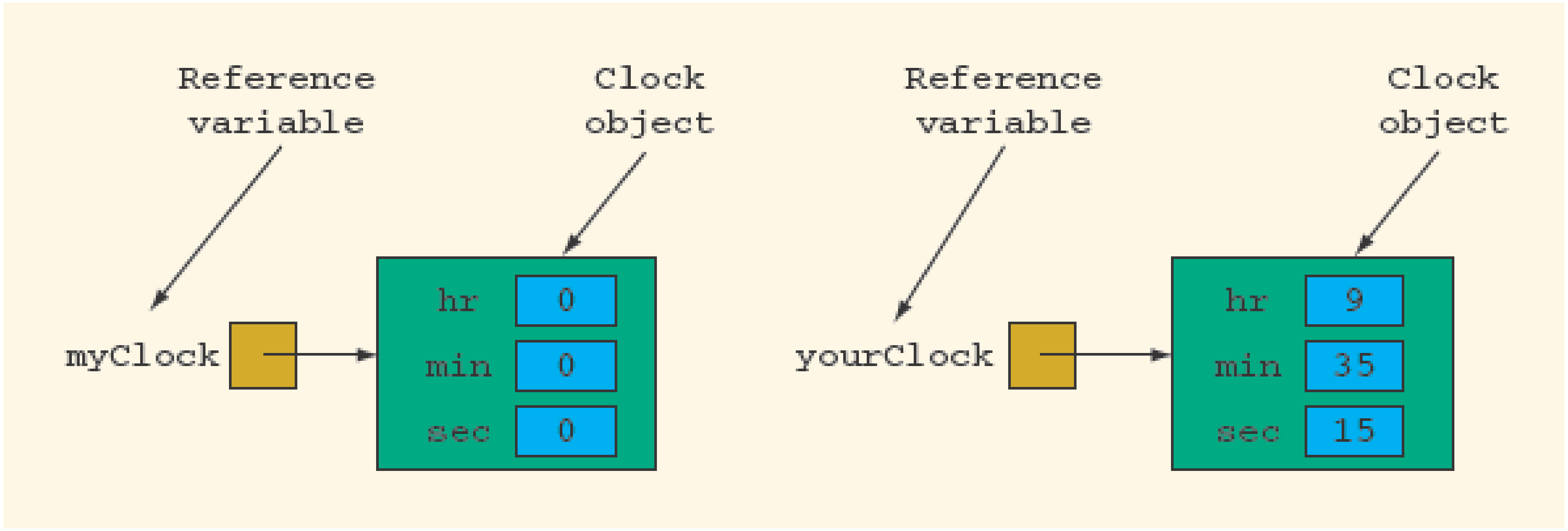


FIGURE 8-2 Variables `myClock` and `yourClock` and associated `Clock` objects

Variable Declaration and Object Instantiation (continued)

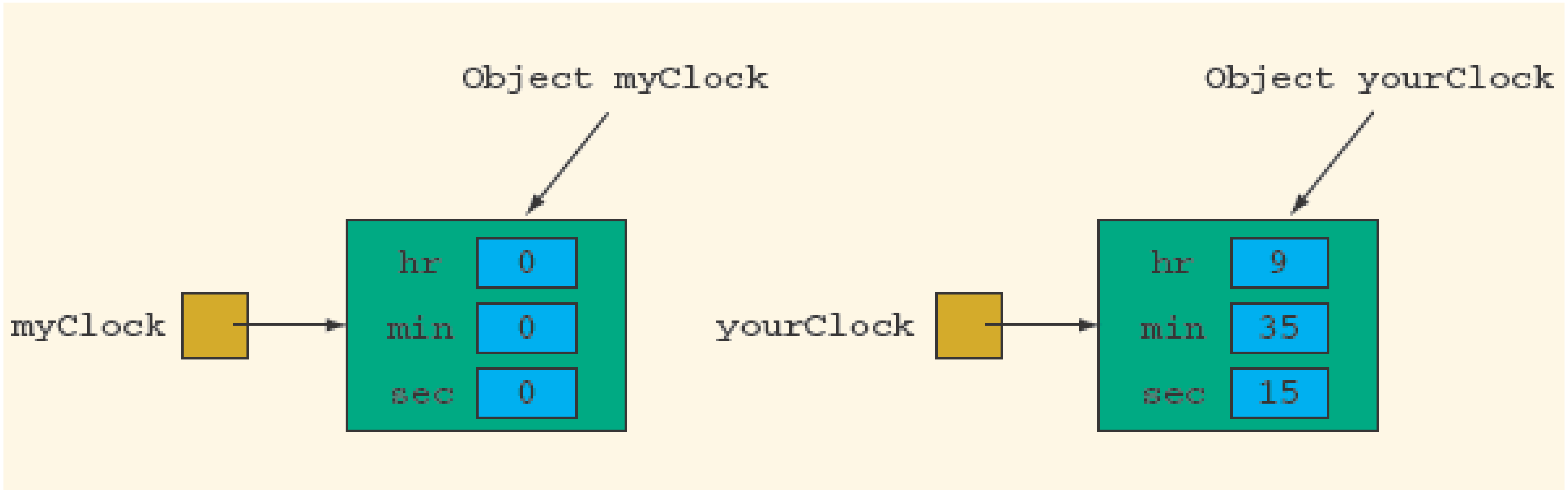


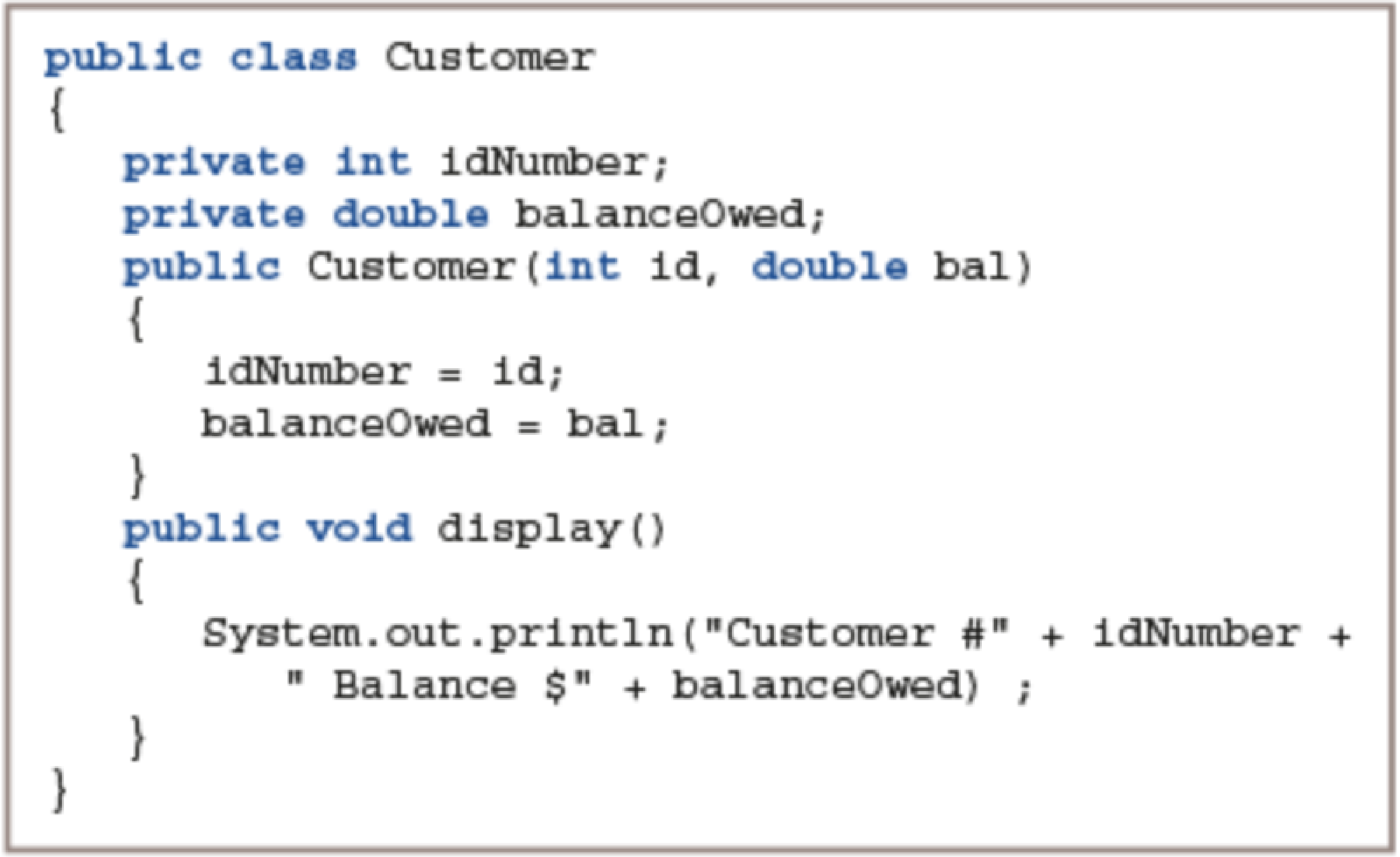
FIGURE 8-3 Objects myClock and yourClock

EXAMPLE 8-3

```
public class Inventory
{
    private String name;
    private int itemNum;
    private double price;
    private int unitsInStock;

    public Inventory()                              //Constructor 1
    {
        name = "";
        itemNum = -1;
        price = 0.0;
        unitsInStock = 0;
    }
    public Inventory(String n)                             //Constructor 2
    {
        name = n;
        itemNum = -1;
        price = 0.0;
        unitsInStock = 0;
    }

    public Inventory(String n, int iNum,
                     double cost)                          //Constructor 3
    {
        name = n;
        itemNum = iNum;
        price = cost;
        unitsInStock = 0;
    }

    public Inventory(String n, int iNum, double cost,
                     int inStock)                          //Constructor 4
    {
        name = n;
        itemNum = iNum;
        price = cost;
        unitsInStock = inStock;
    }

    //Add additional methods
}
```

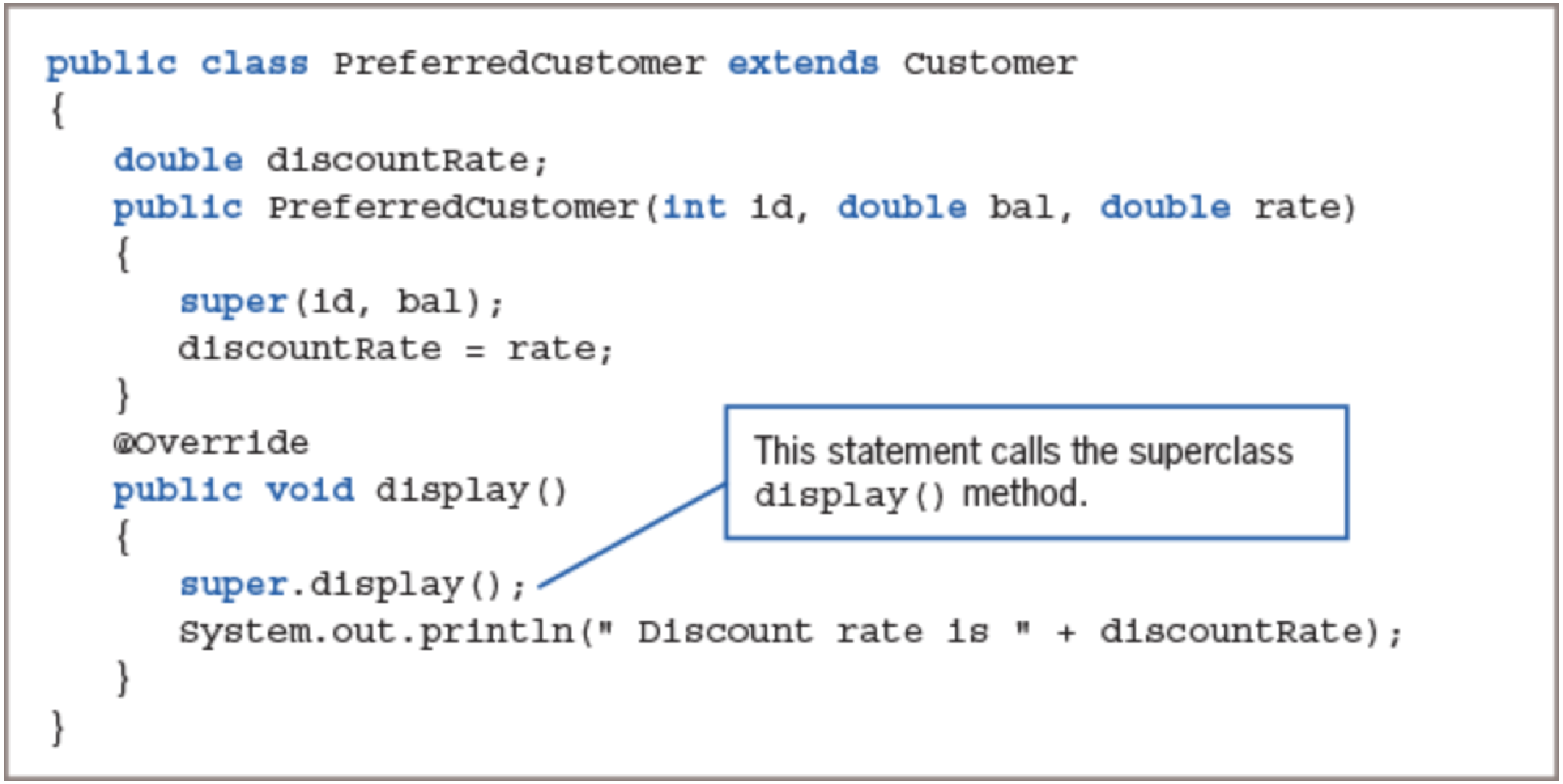
```
Inventory item1 = new Inventory();
Inventory item2 = new Inventory("Dryer");
Inventory item3 = new Inventory("Washer", 2345, 278.95);
Inventory item4 = new Inventory("Toaster", 8231, 34.49, 200);
```

Polymorphism



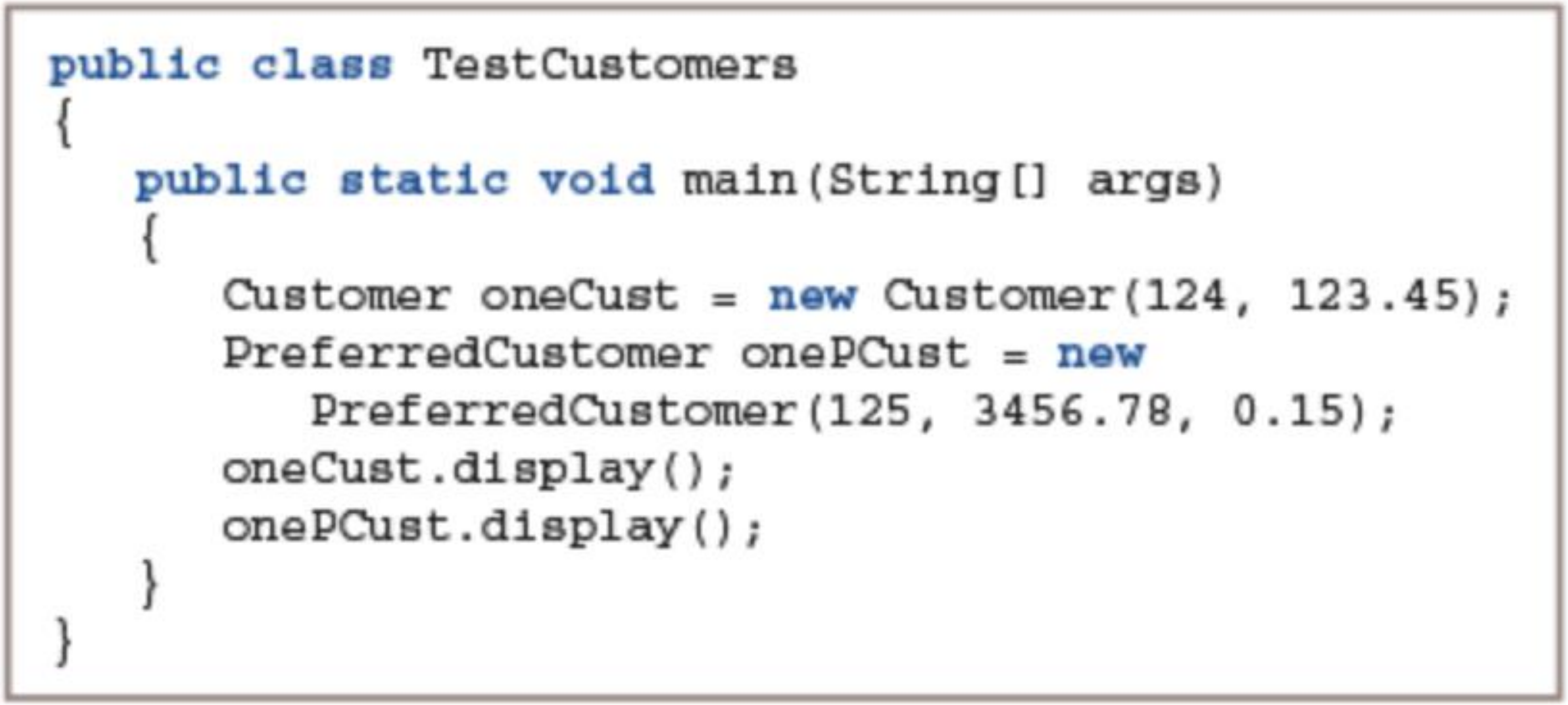
```
public class Customer
{
   private int idNumber;
   private double balanceOwed;
   public Customer(int id, double bal)
   {
      idNumber = id;
      balanceOwed = bal;
   }
   public void display()
   {
      System.out.println("Customer #" + idNumber +
         " Balance $" + balanceOwed) ;
   }
}
```

Super Constructor



```java
public class PreferredCustomer extends Customer
{
   double discountRate;
   public PreferredCustomer(int id, double bal, double rate)
   {
      super(id, bal);
      discountRate = rate;
   }
   @Override
   public void display()
   {
      super.display();
      System.out.println(" Discount rate is " + discountRate);
   }
}
```

This statement calls the superclass `display()` method.



```java
public class TestCustomers
{
   public static void main(String[] args)
   {
      Customer oneCust = new Customer(124, 123.45);
      PreferredCustomer onePCust = new
         PreferredCustomer(125, 3456.78, 0.15);
      oneCust.display();
      onePCust.display();
   }
}
```