NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 5: 5th Mar 2024

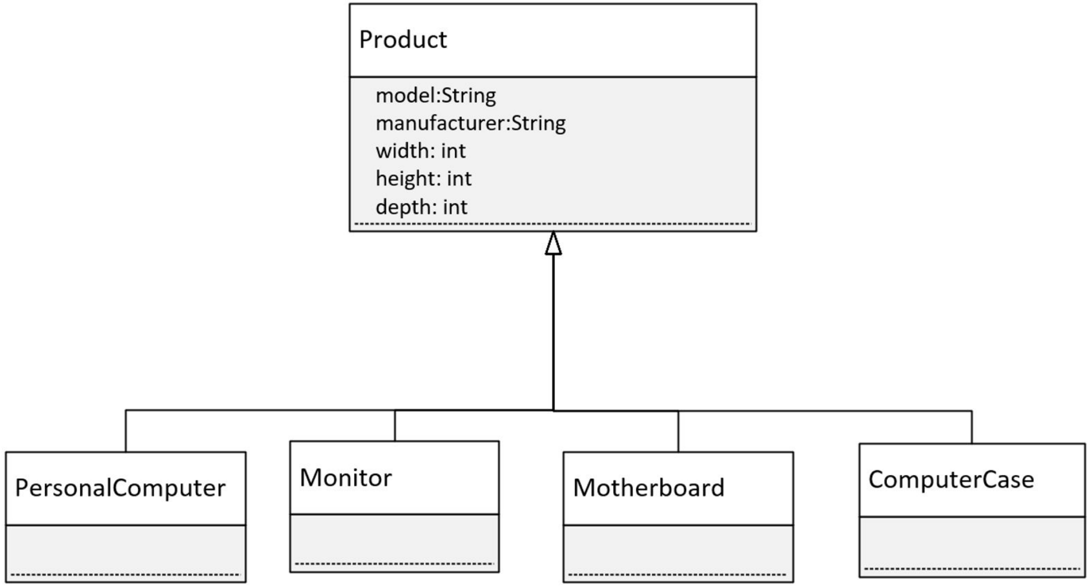
Composition

Inheritance defines an **IS A** relationship.

Composition defines a **HAS A** relationship.

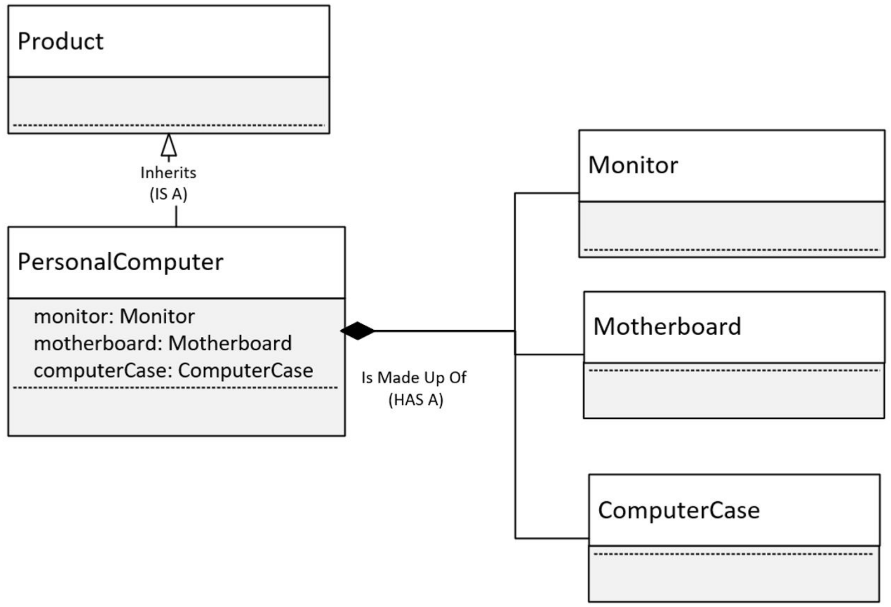
Inheritance

- In this instance, we have a base class called Product.
- All of our computer parts are going to inherit from Product.
- All our parts will then have the same set of attributes, a manufacturer and model, and dimensions, the width, height, and depth in other words.
- All of these items are products, a particular type of Product.

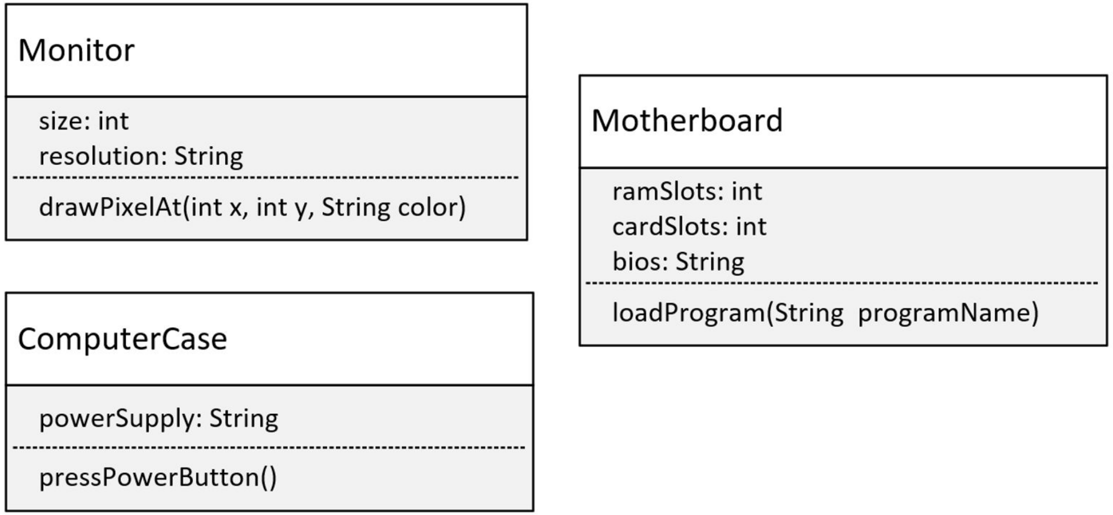


Inheritance Vs Composition

- To keep this diagram simple, PersonalComputer inherits from Product.
- But a Personal Computer, in addition to being a product, is actually made up of other parts.
- Composition is actually modeling parts, and those parts make up a greater whole.
- In this case we're going to model the personal computer.
- And we're modeling the **has a** relationship, with the motherboard, the case, and the monitor.



The Parts

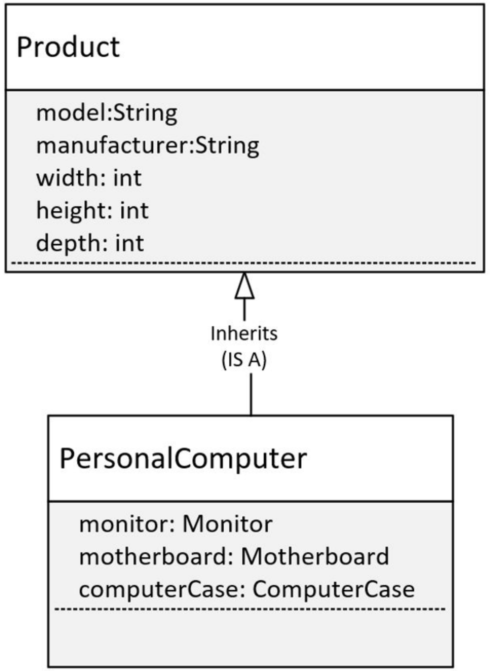


This diagram shows the 3 classes that will make up the personal computer.

PersonalComputer

This will be our personal computer, and we've said it inherits from Product.

But it also has 3 fields, which are classes, these are Monitor, Motherboard, and ComputerCase.



Composition

Inheritance is a way to reuse functionality and attributes.

Composition is a way to make the combination of classes, act like a single coherent object.

Composition is creating a whole from different parts

We built this personal computer, by passing objects, to the constructor, like assembling the computer.

We can actually hide the functionality further.

In this case, we're not going to allow the calling program, to access those objects, the parts, directly.

We don't want anybody to access the Monitor, Motherboard, or ComputerCase directly.

Use Composition or Inheritance or Both?

As a general rule, when you're designing your programs in Java, you probably want to look at composition first.

Most of the experts will tell you, that as a rule, look at using composition before implementing inheritance.

You saw in this example, we actually used both.

All of our parts were able to inherit a set of attributes, like the manufacturer and model.

The calling code didn't have to know anything about these parts, to get Personal Computer to do something.

Why is Composition preferred over Inheritance in many designs?

The reasons composition is preferred over inheritance:

- Composition is more flexible. You can add parts in, or remove them, and these changes are less likely to have a downstream effect.
- Composition provides functional reuse outside of the class hierarchy, meaning classes can share attributes & behavior, by having similar components, instead of inheriting functionality from a parent or base class.
- Java's inheritance breaks encapsulation, because subclasses may need direct access to a parent's state or behavior.

Why is Inheritance less flexible?

Inheritance is less flexible.

Adding a class to, or removing a class from, a class hierarchy, may impact all subclasses from that point.

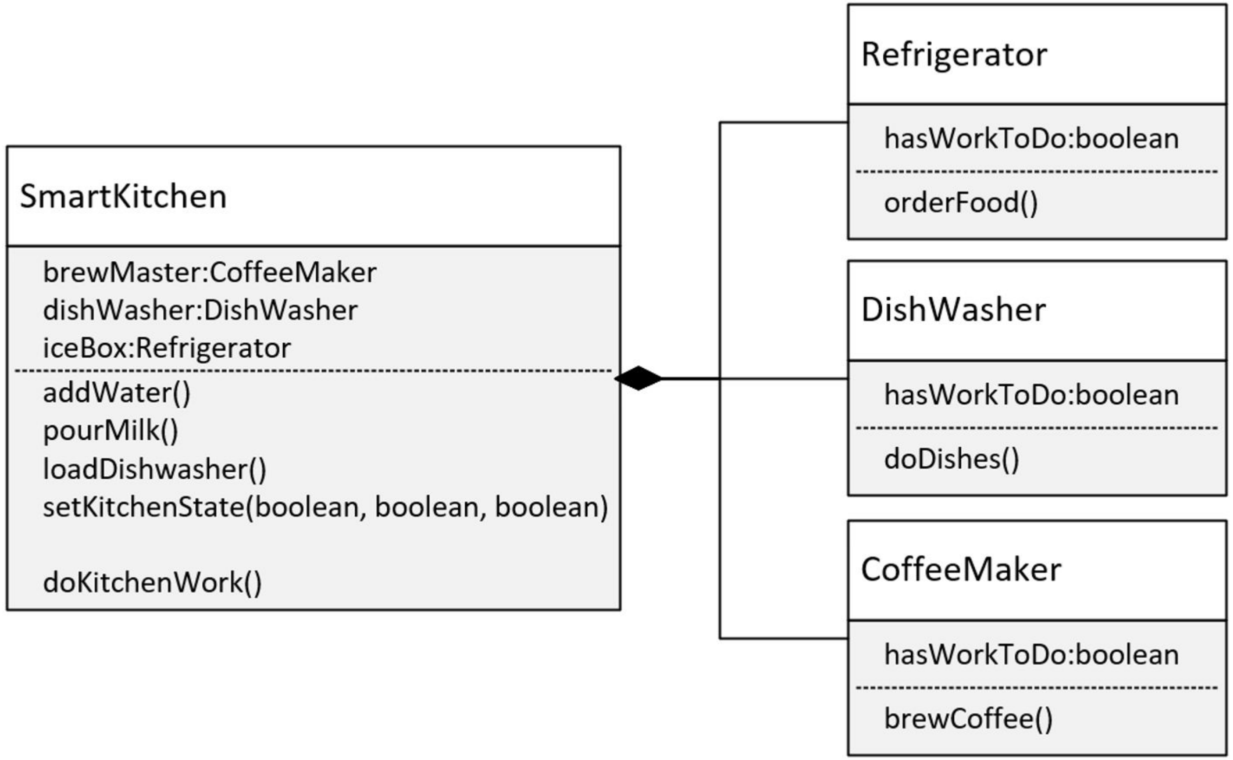
In addition, a new subclass may not need all the functionality or attributes of its parent class.

The Composition Challenge

In this challenge, you need to create an application for controlling a smart kitchen.

Your smart kitchen will have several appliances.

Your appliances will be Internet Of Things (IoT) devices, which can be programmed.



The Composition Challenge

It's your job to write the code, to enable your Smart Kitchen application, to execute certain jobs.

Methods on your SmartKitchen class, will determine what work needs to be done:

- addWater() will set the Coffee Maker's hasWorkToDo field to true.
- pourMilk() will set Refrigerator's hasWorkToDo to true.
- loadDishwasher() will set the hasWorkToDo flag to true, on that appliance.

Alternately, you could have a single method, called setKitchenState, that takes three boolean values, which could combine the three methods above.

The Composition Challenge

To execute the work needed to be done by the appliances, you'll implement this in two ways:

First, your application will access each appliance (by using a getter), and execute a method.

- The appliance methods are orderFood() on Refrigerator, doDishes() on DishWasher, and brewCoffee() on CoffeeMaker.
- These methods should check the hasWorkToDo flag, and if true, print a message out, about what work is being done.

Second, your application won't access the appliances directly.

- It should call doKitchenWork(), which delegates the work, to any of its appliances.

Encapsulation

Encapsulation means hiding things, by making them private, or inaccessible.

Why hide things?

Why would we want to hide things in Java?

- To make the interface simpler, we may want to hide unnecessary details.
- To protect the integrity of data on an object, we may hide or restrict access to some of the data and operations.
- To decouple the published interface from the internal details of the class, we may hide actual names and types of class members.

What do we mean by interface here?

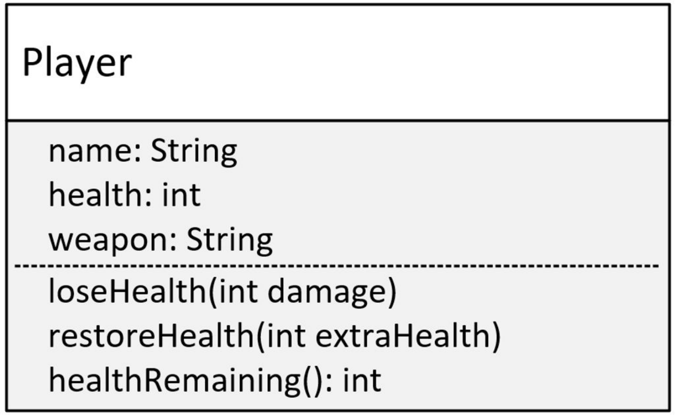
Although Java has a type called interface, that's not what we're talking about here.

When we talk about a class's public or published interface, we're really talking about the class members that are exposed to, or can be accessed by, the calling code.

Everything else in the class is internal, or private to it.

An application programming interface, or API, is the public contract, that tells others how to use the class.

The Player Class



This is the model for a Player class.

The Player will have three variables: name, health, and weapon.

And this class will have three methods, loseHealth(), restoreHealth(), and healthRemaining(), which I'll explain in a bit.

And we're going to create this class without using encapsulation.

Problems

- Allowing direct access to data on an object, can potentially bypass checks, and additional processing, your class has in place to manage the data.
- Allowing direct access to fields, means calling code would need to change, when you edit any of the fields.
- Omitting a constructor, that would accept initialization data, may mean the calling code is responsible for setting up this data, on the new object.

The problems when classes aren't properly encapsulated

Allowing direct access to data on an object, can bypass checks and operations.

It encourages an interdependency, or coupling, between the calling code and the class.

For the previous example, we showed that changing a field name, broke the calling code.

And we also showed, that the calling code had to take on the responsibility, for properly initializing a new Player.

Benefits of Encapsulation

That's really one of the huge benefits of encapsulation, is that you're not actually affecting any other code.

It's sort of like a black box in many ways.

But the EnhancedPlayer class has more control over it's data.

Staying in Control

This is why we want to use encapsulation.

We protect the members of the class, and some methods, from external access.

This prevents calling code from bypassing the rules and constraints, we've built into the class.

When we create a new instance, it's initialized with valid data.

But likewise, we're also making sure that there's no direct access to the fields.

That's why you want to always use encapsulation.

It's something that you should really get used to.

Encapsulation Principles

To create an encapsulated class, you want to:

Create constructors for object initialization, which enforces that only objects with valid data will get created.

Use the private access modifier for your fields.

Use setter and getter methods sparingly, and only as needed.

Use access modifiers that aren't private, only for the methods that the calling code needs to use.

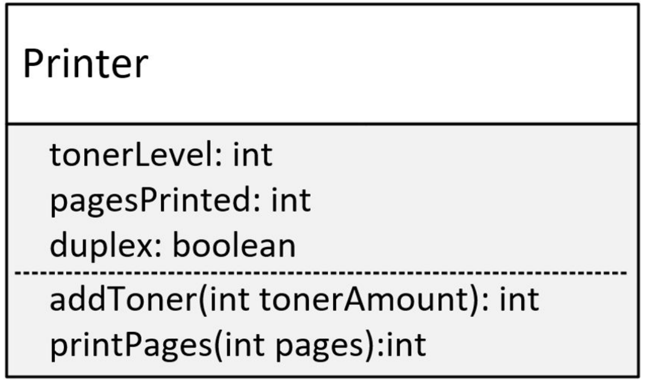
Encapsulation Challenge

In this challenge, you need to create a class named Printer.

The fields on this class are going to be:

- **tonerLevel**, which is the percentage of how much toner level is left.
- **pagesPrinted**, which is the count of total pages printed by the Printer.
- **duplex**, which is a boolean indicator. If true, it can print on 2 sides of a single sheet of paper.

You'll want to initialize your printer, by specifying a starting toner amount, and whether the printer is duplex or not.



Encapsulation Challenge

On the Printer class, you want to create two methods, which the calling code should be able to access.

These methods are:

- addToner() which takes a tonerAmount argument.
  - tonerAmount is added to the tonerLevel field.
  - The tonerLevel should never exceed 100 percent, or ever get below 0 percent.
  - If the amount being added makes the level fall outside that range, return a -1 from the method, otherwise return the actual toner level.

Encapsulation Challenge

printPages() which should take pages to be printed as the argument.

- It should determine how many sheets of paper, will be printed based on the duplex value, and return this sheet number from the method.
- The sheet number should also be added to the pagesPrinted variable.
- If it's a duplex printer, print a message that it's a duplex printer.

Polymorphism

Polymorphism lets us write code to call a method, but at runtime, this method's behavior can be different, for different objects.

This means the behavior that occurs, while the program is executing, depends on the runtime type of the object.

And the runtime type, might be different from the declared type in the code.

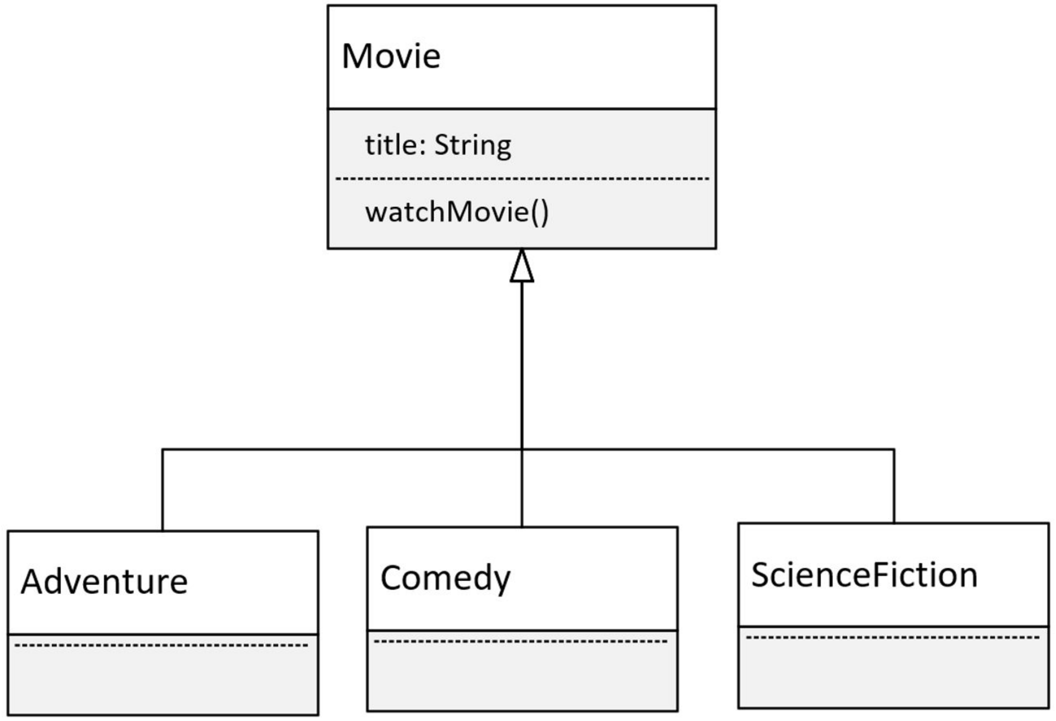
The declared type has to have some kind of relationship to the runtime type, and inheritance is one way to establish this relationship.

Movie Genres

This time, we're going to look at a polymorphism example, using movies.

We'll have a base class of Movie, which has the title of the movie.

And Movie will have one method, watchMovie.



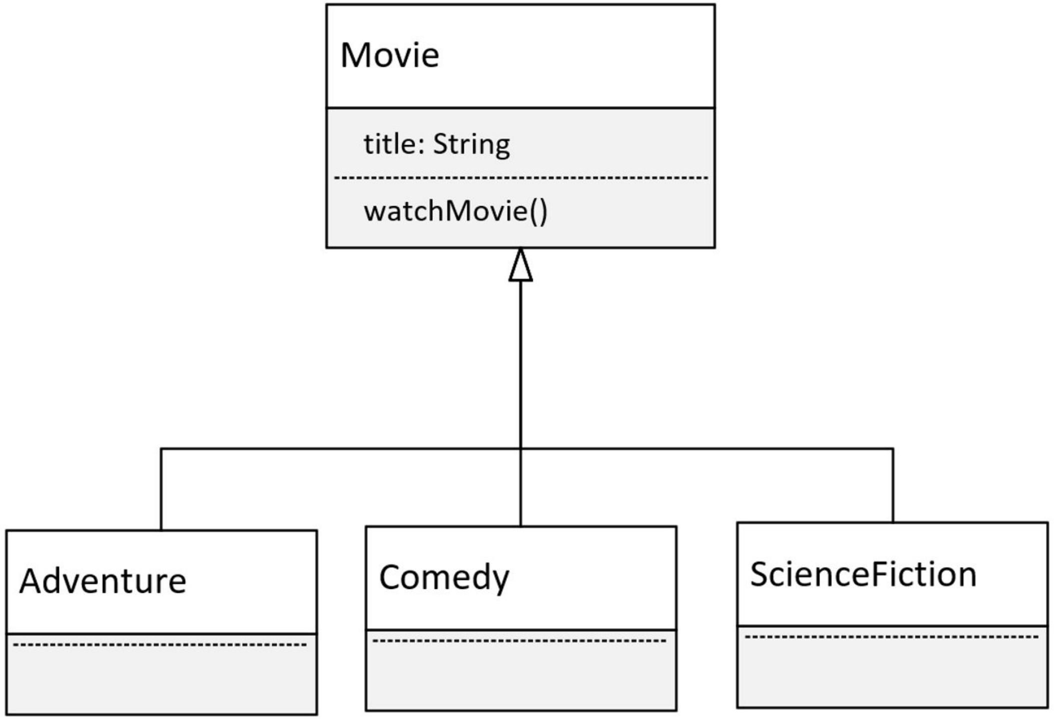
Movie Genres

We'll have 3 subclasses, each a different kind of movie.

We'll have an Adventure film, a Comedy, and a Science Fiction movie.

These are the different categories, so we'll use these as the subclasses.

All of these will override, and implement unique behavior, for the watchMovie method.



Polymorphism in action

That was polymorphism in action.

It's the ability to execute different behavior, for different types, which are determined at runtime.

And yet we did it with just two statements, in the main method, as shown here.

```
Movie movie = Movie.getMovie(type, title);
movie.watchMovie();
```

Polymorphism enables you to write generic code, based on the base class, or a parent class.

And this code, in the main method, is extendable, meaning it doesn't have to change, as new subclasses become available.

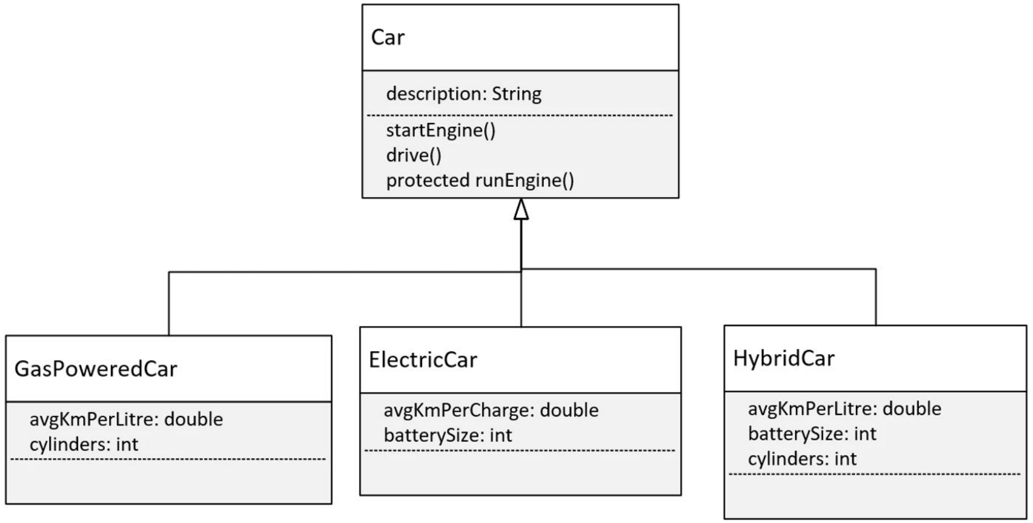
This code can handle any instances that are a Movie, or a subclass of movie, that are returned from the factory method.

The Polymorphism Challenge

This diagram shows a base class, Car, with one field, description, and three methods, startEngine(), drive() and runEngine().

The first two methods should be declared as public.

The method runEngine however, is protected, and it will only get called from the drive method in Car.



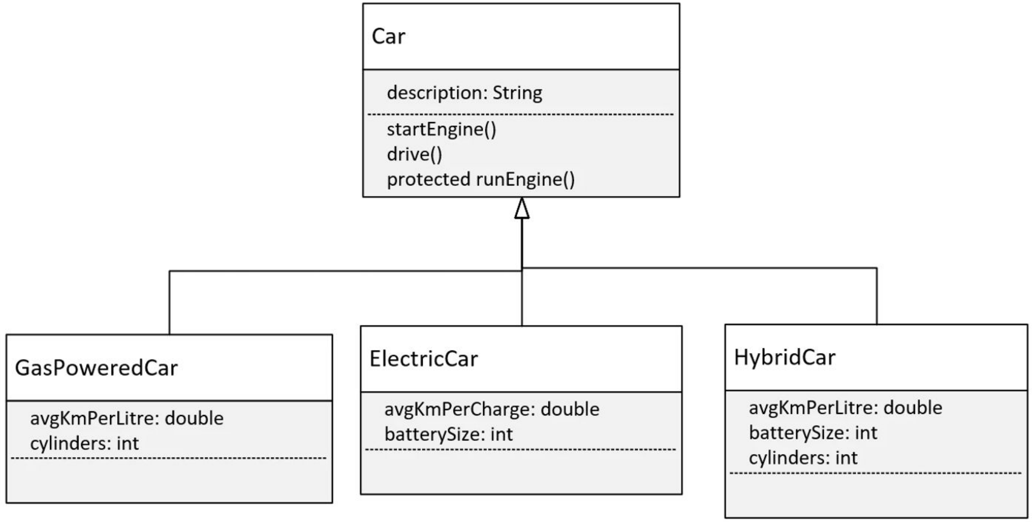
The Polymorphism Challenge

And here, we've given you three types of subclasses, or three types of cars that you might find on the road.

We have the GasPoweredCar, the ElectricCar, and the HybridCar.

You can imagine that these three subclasses, might have different ways to start their engine, or drive, depending on their engine type.

And each of these classes might have different variables or fields, that might be used in those methods.

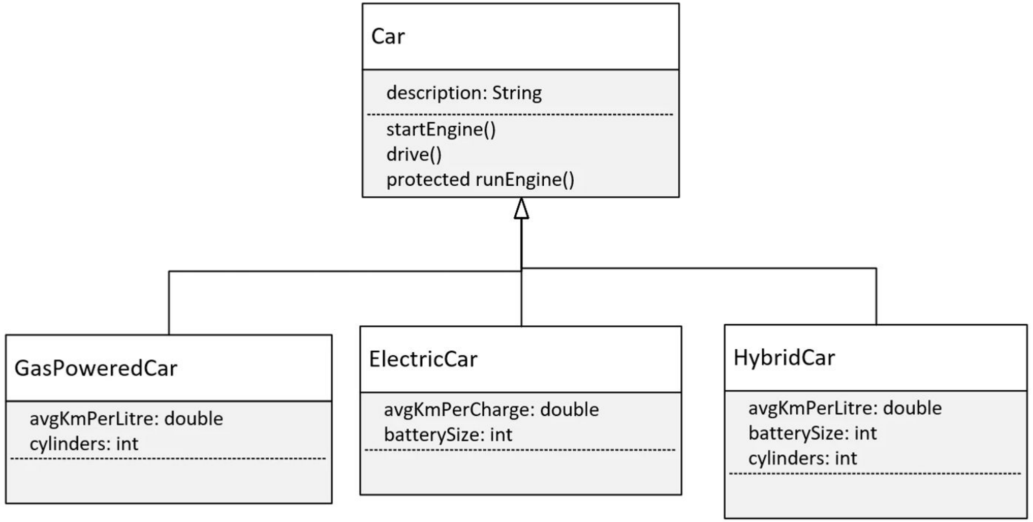


The Polymorphism Challenge

It's your job, to create this class structure in Java, and override some, or maybe all, of these methods appropriately.

And you'll write code in a Main class and main method, that creates an instance of each of these classes, and that executes different behavior for each instance.

At least one method should print the type of the runtime object.


Car
description: String
startEngine()
drive()
protected runEngine()
GasPoweredCar
avgKmPerLitre: double
cylinders: int
ElectricCar
avgKmPerCharge: double
batterySize: int
HybridCar
avgKmPerLitre: double
batterySize: int
cylinders: int