# Composition, Encapsulation, and Polymorphism

This is part 2 of the Object-Oriented Programming sections of the course.

We covered the basics of Object-Oriented Programming in the previous section.

It's now time to look at the remaining 3 major components of object-oriented programming.

# Composition, Encapsulation, and Polymorphism

These are **Composition**, **Encapsulation**, and **Polymorphism**.

By the end of this section, you should have a solid overview of what these concepts are and how to apply them to your programs.

Let's make a start on that right now.

# Composition

It's time now to talk about composition.

Composition is another component of object-oriented programming.

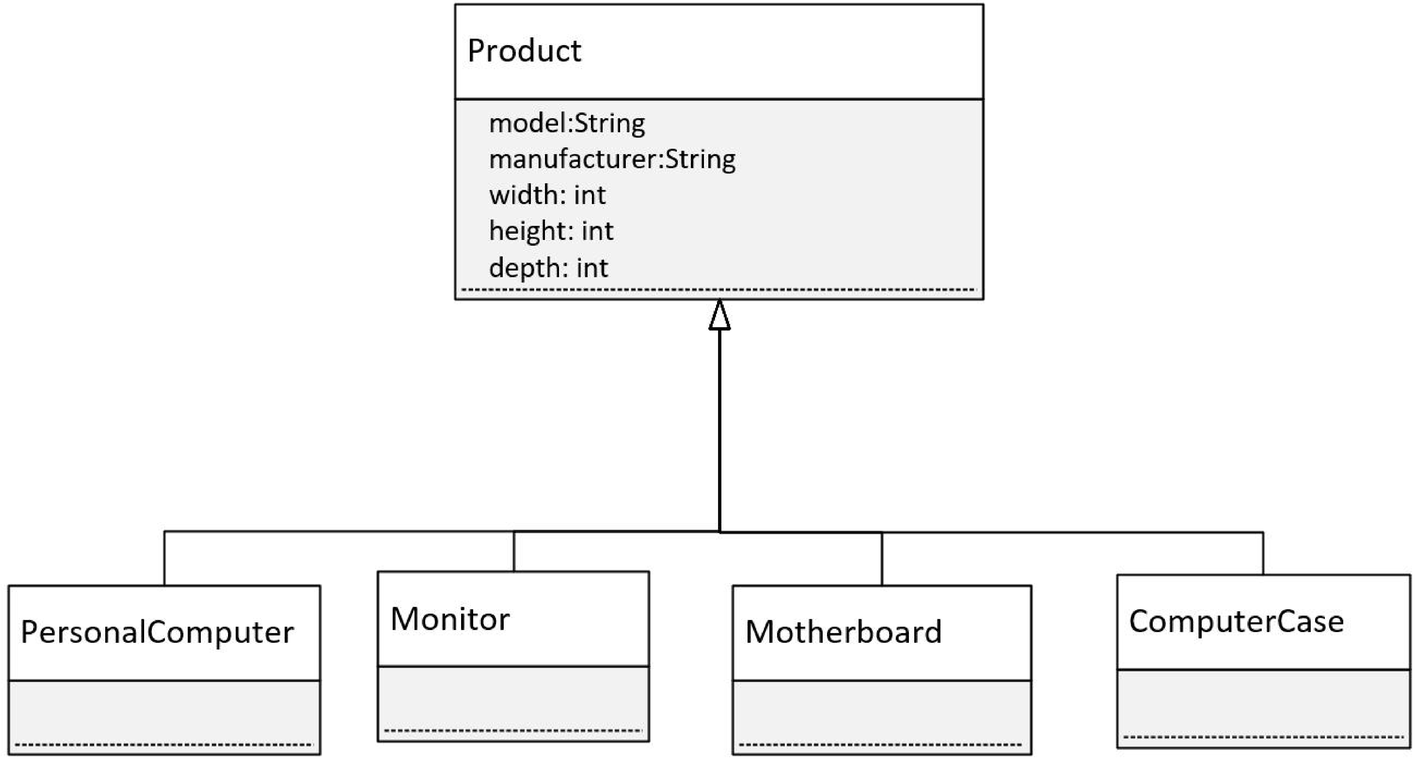
# Inheritance

In this instance, I have a base class called Product.

All of my computer parts are going to inherit from Product.

All my parts will then have the same base set of attributes: manufacturer, 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

Inheritance defines an **IS A** relationship.

Composition defines a **HAS A** relationship.

# Inheritance vs Composition

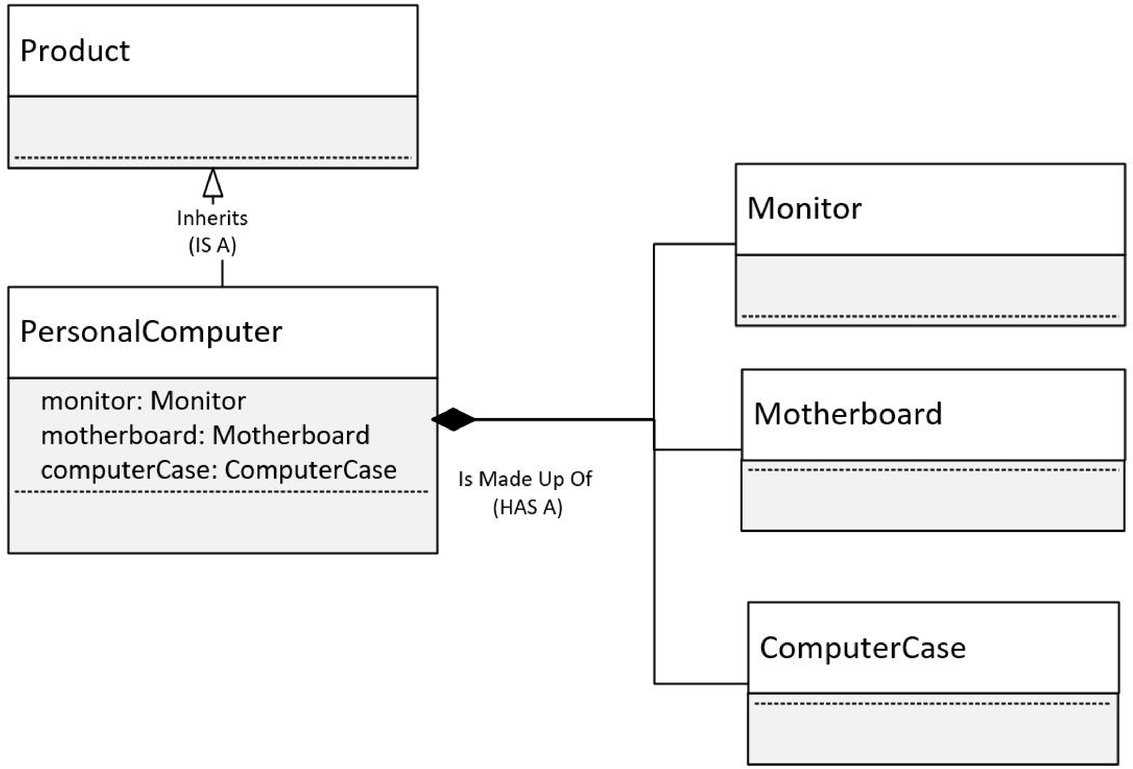
To keep this diagram simple, PersonalComputer inherits from Product.

But a Personal Computer, in addition to being a product, is actually made up of other parts.

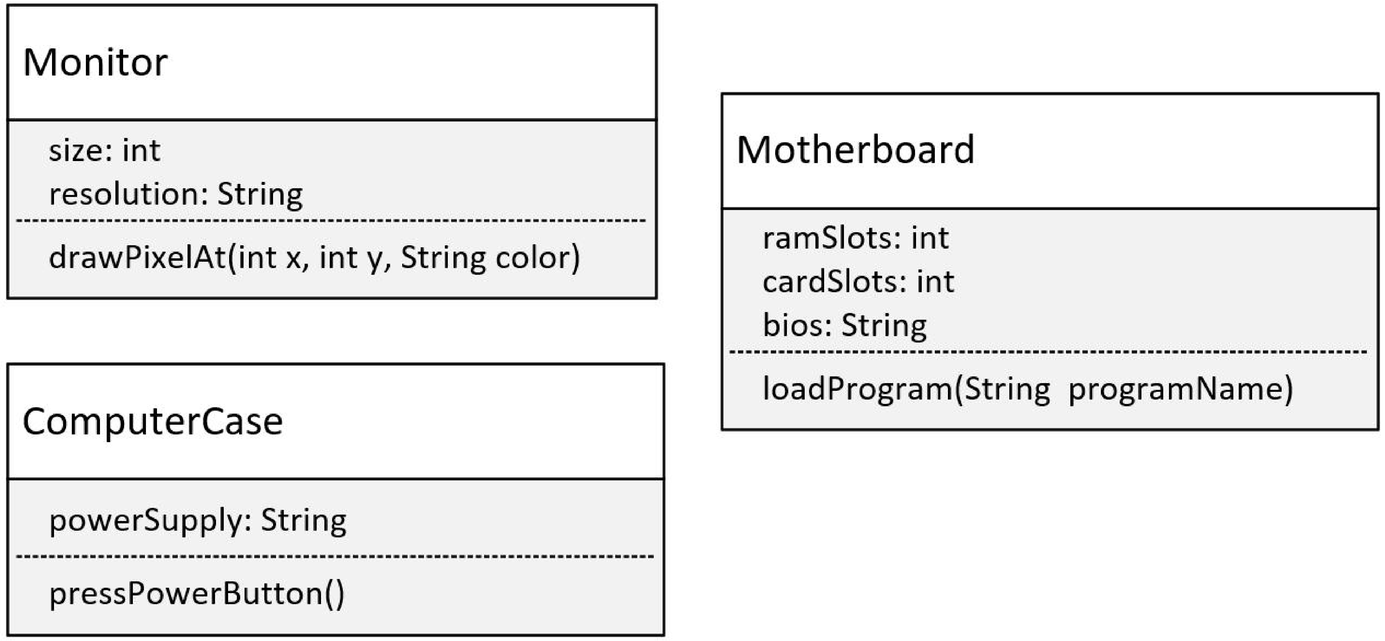
Composition is actually modeling parts, and those parts make up a greater whole.

In this case, I'm going to model the personal computer.

And I'm modeling the **has a** relationship with the motherboard, the case, and the monitor.

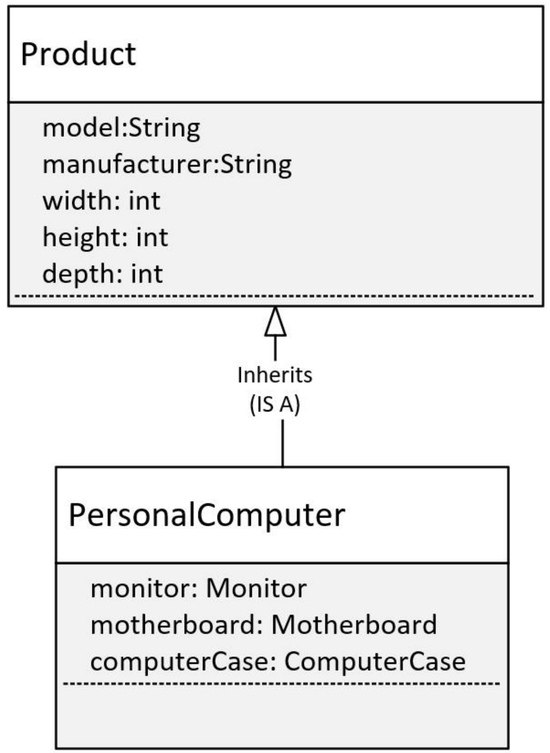


# The Parts




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

# PersonalComputer




This will be my personal computer, and I've said it inherits from Product.

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

# Composition

In the previous video, I talked about **composition** and compared it to inheritance.

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 par

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

I can actually hide the functionality further.

In this case, I'm not going to allow the **calling program** to **access those objects**, **the parts directly**.

I 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, I actually used both.

All of my 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 PersonalComputer to do something.

# Why is Composition preferred over Inheritance in many design

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.

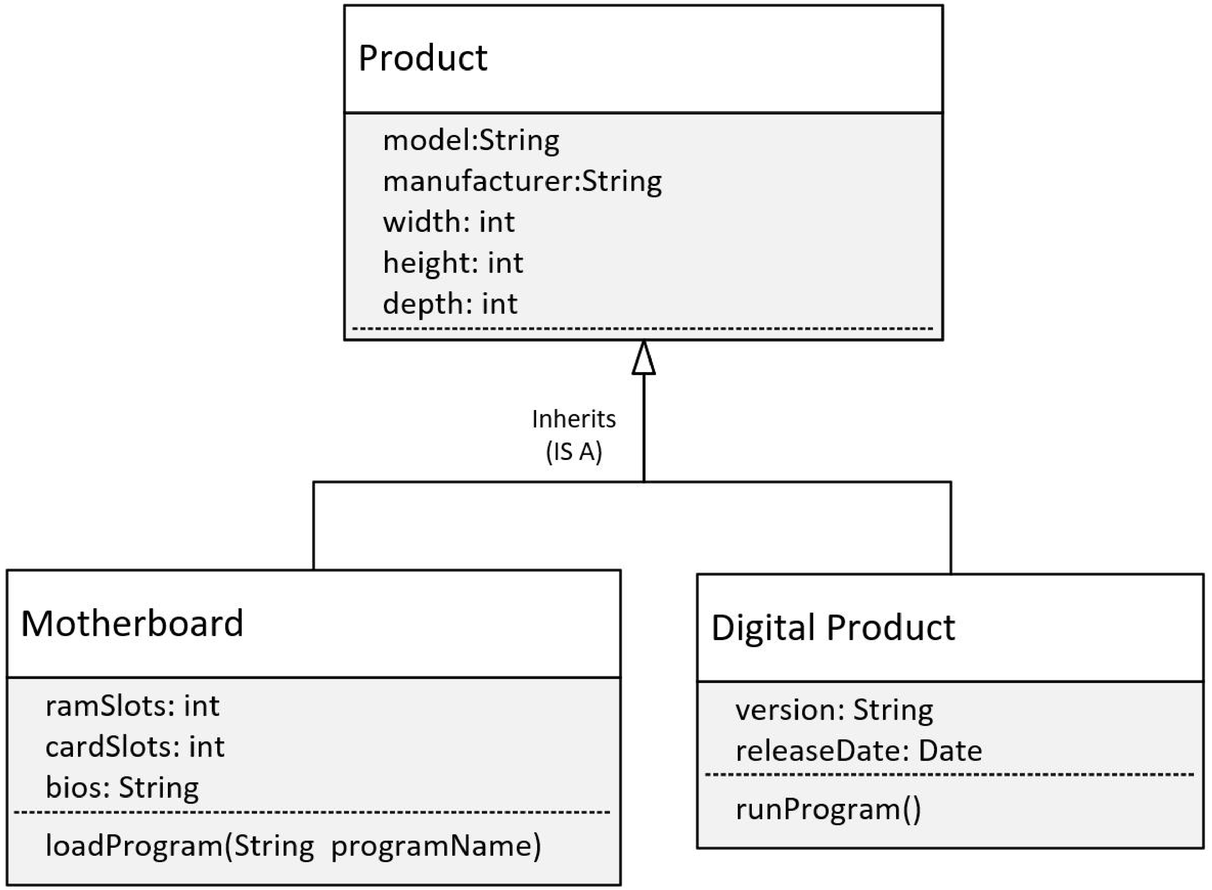
# Adding a Digital Product

Let's say I want to include digital products, such as software products in my product inventory.

Should Digital Product inherit from Product?

Here, I show the model with Digital Product, inheriting from my current definition of Product.

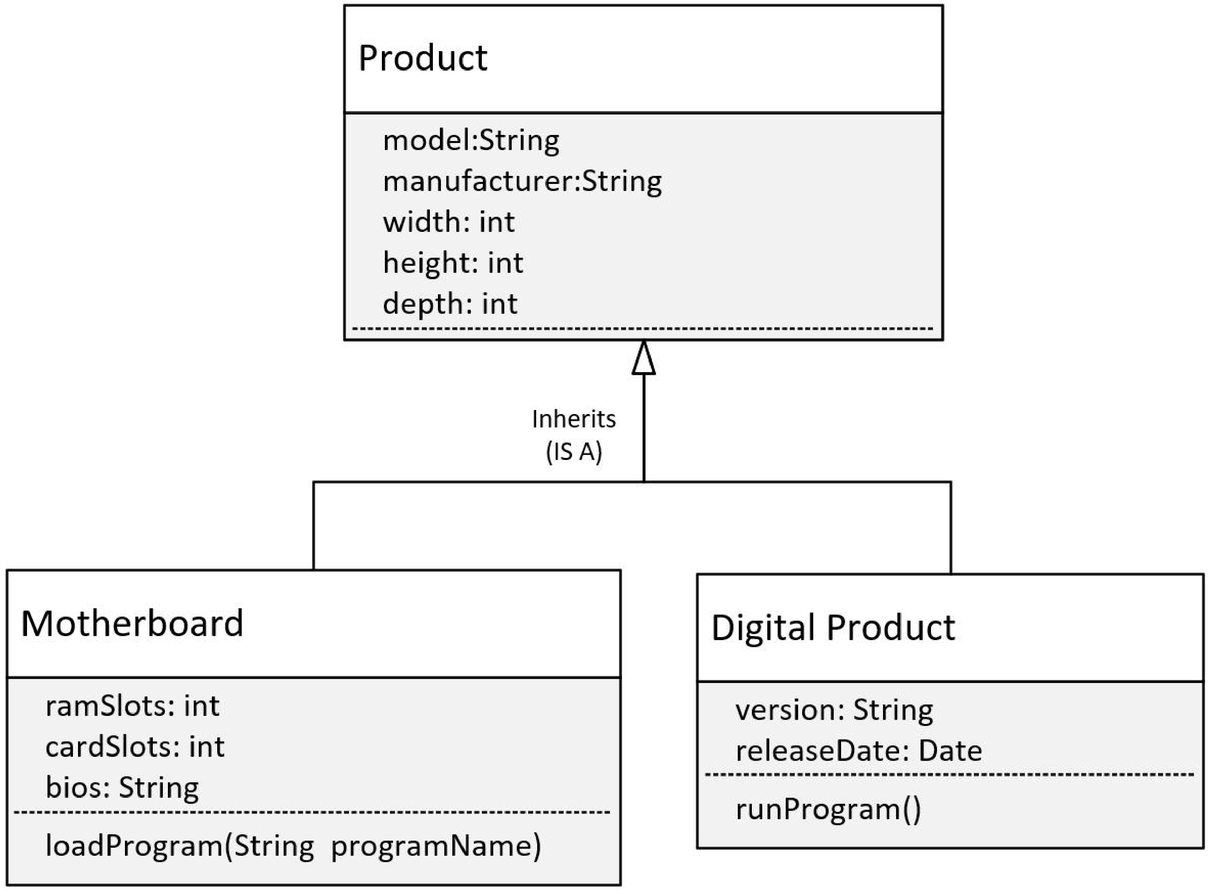
If I do this, this would mean Digital Product has Product's attributes, but this isn't true now.



# Adding a Digital Product

A digital product wouldn't really have width, height, and depth, so this model isn't a good representation of what I want to build.

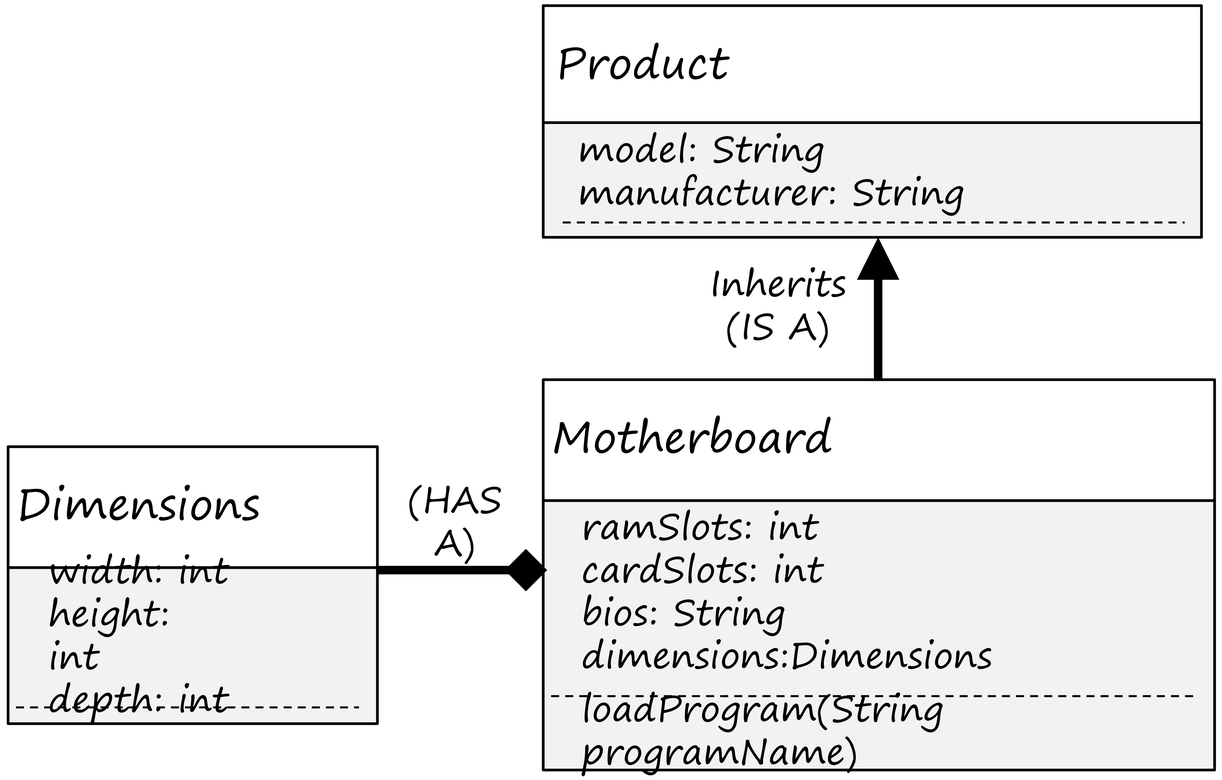
It would be better if I didn't have those three attributes on Product, but instead used composition to include them on certain products, but not all products.



# Revised Class Diagram

Consider this revised class diagram.

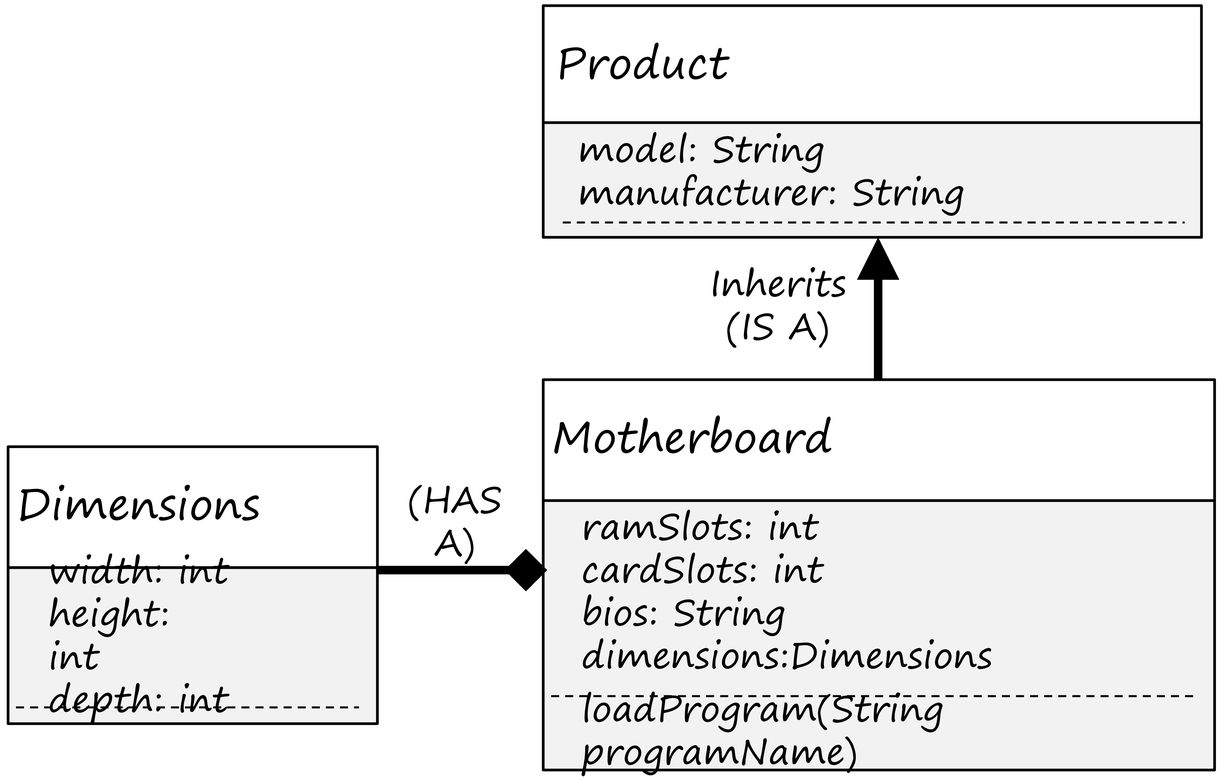
I haven't completely removed the class hierarchy, but I've made the base class, Product, more generic.



# Revised Class Diagram

I've removed the width, height, and depth attributes from Product and made a new class, Dimensions, with those attributes.

And I've added an attribute to Motherboard, which is dimensions, which has those attributes.



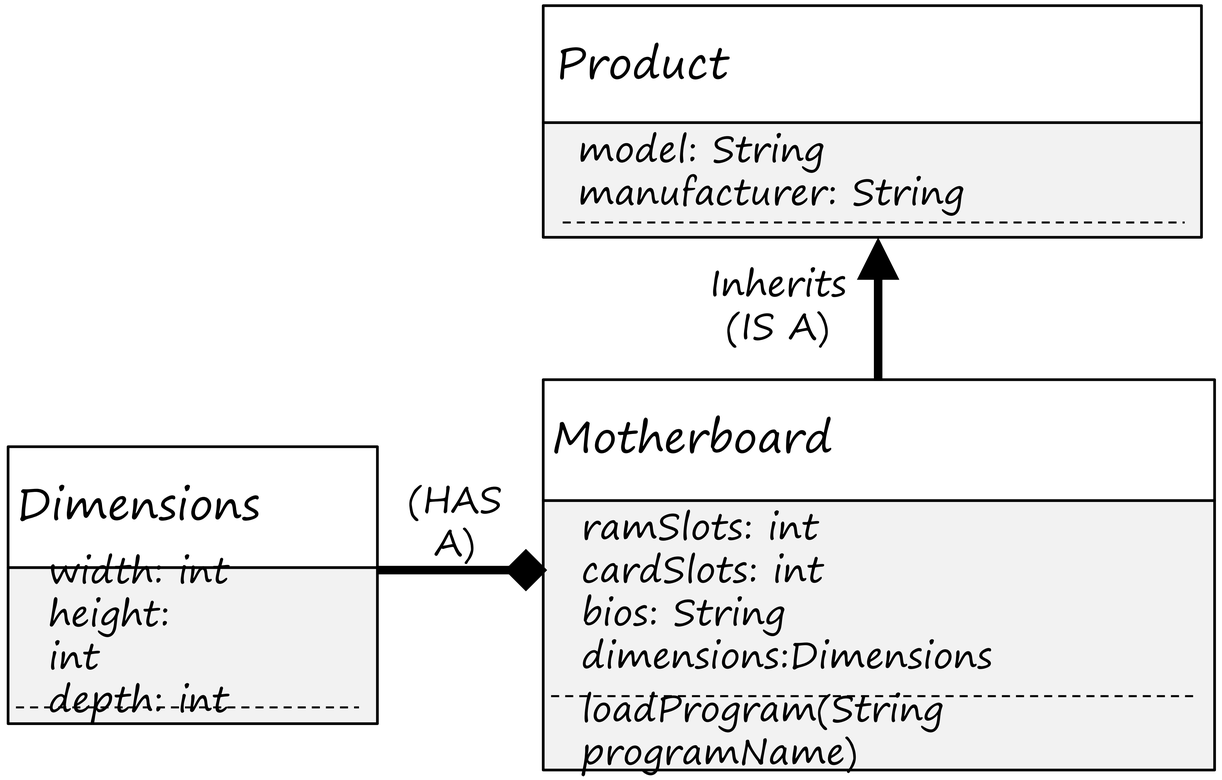
# Revised Class Diagram

Is this a better model?

Well, it's more flexible.

This design allows for future enhancements to be made, like the addition of the subclass Digital Product, without causing problems for existing code that may already be extending Product.

By placing width, height, and depth into a dimension class, I can use composition to apply those attributes to any product.

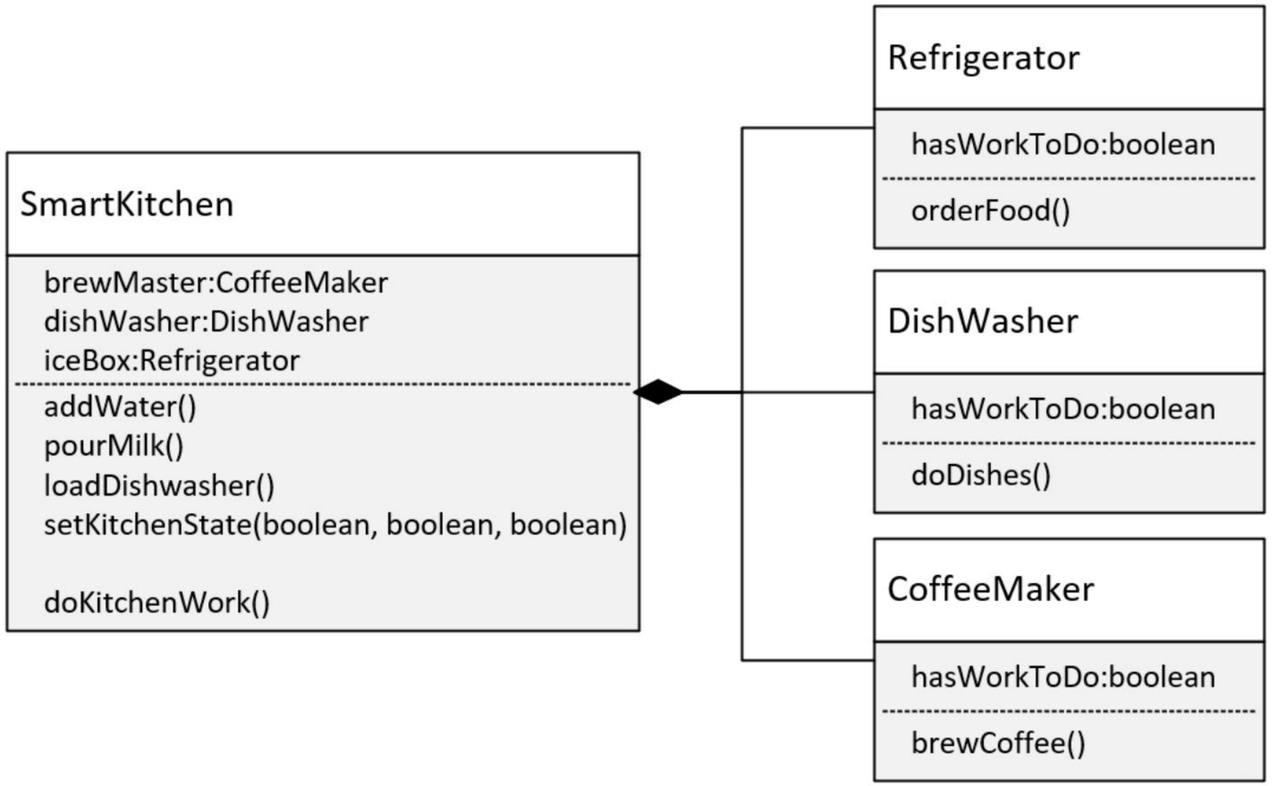


# The Composition Challenge

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 the Refrigerator's hasWorkToDo to true.
- loadDishwasher() will set the hasWorkToDo flag to true for that appliance.

Alternately, you could have a single method called setKitchenState that takes three boolean values, which would set each appliance accordingly.

# 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 indicating 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.

# What does Encapsulation Mean?

In Java, 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?

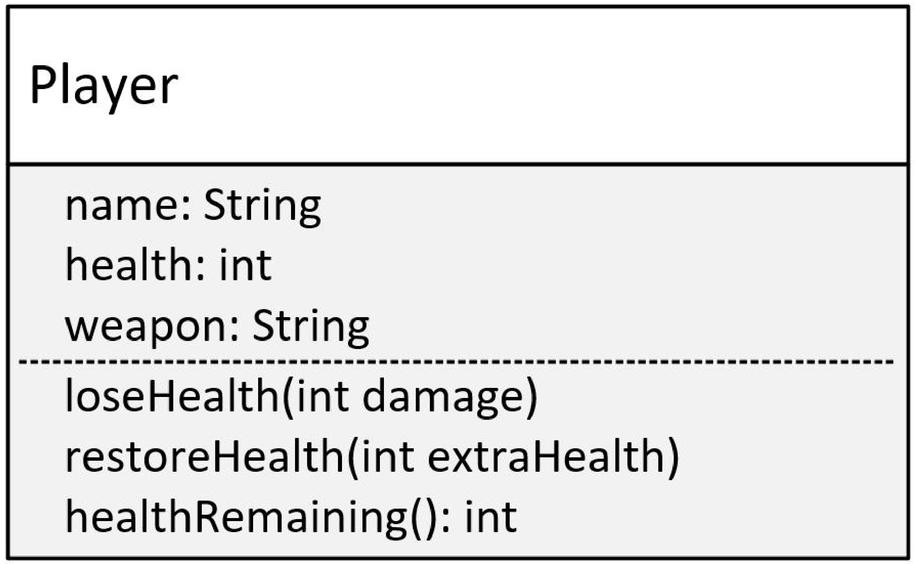
Although Java has a type called interface, that's not what I'm talking about here.

When I talk about a class's public or published interface, I'm 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 has three fields: name, health, and weapon.

This class has three methods: loseHealth(), restoreHealth(), and healthRemaining(), which I'll explain shortly.

I'm going to create this class without using encapsulation.

# Problem One

Allowing direct access to data on an object can potentially bypass checks and additional processing your class has in place to manage the data.

# Problem Two

Allowing direct access to fields means calling code would need to change when you edit any of the fields.

# Problem Three

Omitting a constructor that would accept initialization data means 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, I showed that changing a field name broke the calling code.

And I 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. Changes made do not affect any other code.

It's like a black box in many ways.

But the enhanced player class has full control over its data.

# Staying in Control

This is why you 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 I create a new instance, it's initialized with valid data.

But likewise, I'm 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 and use.

# Encapsulation Principles

To create an encapsulated class, you want to:

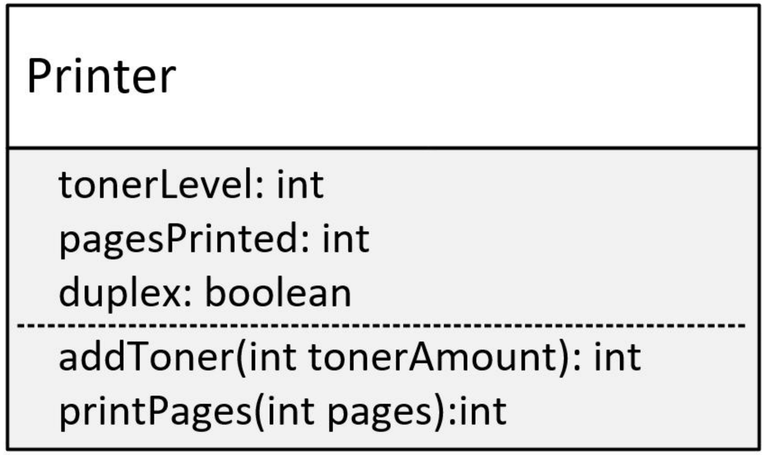
- Create constructors for object initialization, which enforces that only objects with valid data will get created.
- Use the private access modifier for your fields.
- Use setter methods sparingly and only as needed.
- Use access modifiers that aren't private, only for the methods that the calling code needs to use.

# Encapsulation Challenge

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

The fields on this class are as follows:

- **tonerLevel**, which is the percentage of toner left in the toner cartridge.
- **pagesPrinted**, which is the count of total pages printed.
- **duplex**, which is an indicator of whether the printer will print on both sides of a sheet of paper. True means it can, False means it can only print on one side of paper.




You'll want to initialize your printer by specifying a starting toner amount and whether the printer has duplex capabilities, 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 after adding the amount passed to the method.

# Encapsulation Challenge

- printPages() which should take pages to be printed as the argument.
  - It should determine how many sheets of paper will be printed. It should take into account the duplex value set for the printer. It should return the calculated number of sheets of paper.
  - The sheet number should also be added to the pagesPrinted field.
  - If it's a duplex printer, print a message that it's a duplex printer.

# Polymorphism

Simply stated, polymorphism means many forms.

How does this apply to code?

# Polymorphism

Polymorphism in Java allows us to write code that can call a method, but the actual method that gets executed can be different for different objects at runtime.

This means that the behavior that occurs during program execution depends on the runtime type of the object, which might differ from its declared type in the code.

For polymorphism to work, the declared type must have a relationship with the runtime type. Inheritance is one way to establish this relationship, where a subclass can override a method from its superclass, enabling polymorphic behavior.

There are other mechanisms to achieve polymorphism, but in this discussion, we'll focus on using inheritance to support polymorphism.

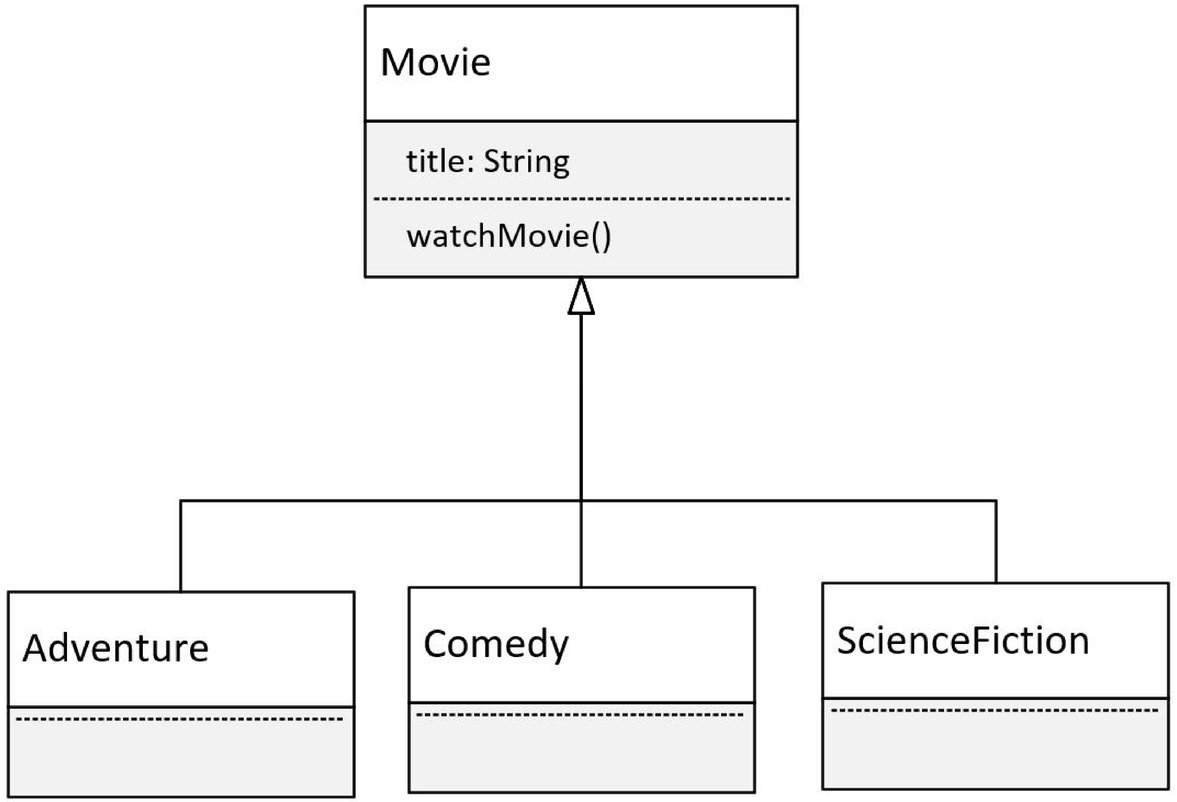
Previously, I touched on this concept in the inheritance videos, but now, we will delve into polymorphism specifically.

# Movie Genres

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

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

And Movie will have one method, watchMovie.



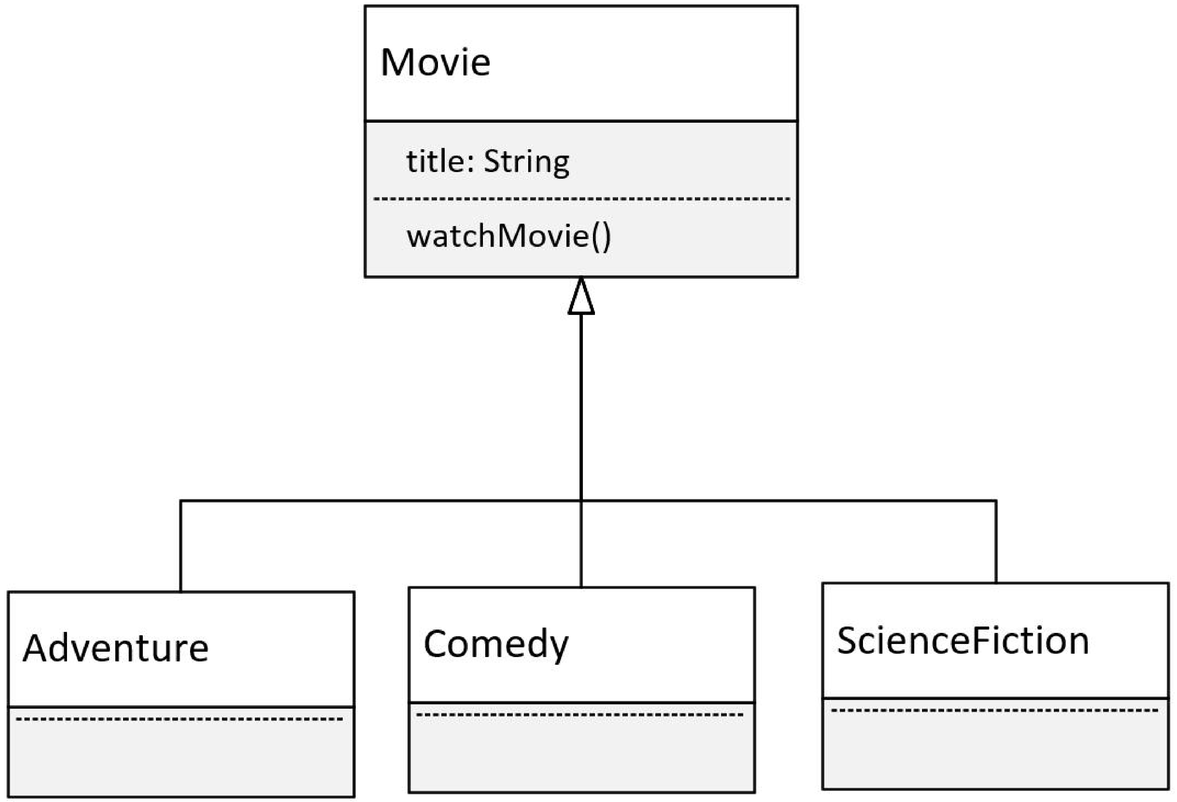
# Movie Genres

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

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

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

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



# Imports

In the Exception handling video in section 6, I showed you how to manually add import lines.

Because I still have Auto Imports enabled, you just saw how IntelliJ added the import for me automatically.

# 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, I 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.

# What is var?

var is a special contextual keyword in Java that lets our code take advantage of Local Variable Type Inference.

By using var as the type, we're telling Java to figure out the compile-time type for us.

# Local Variable Type Inference (LVTI)

Local Variable Type Inference was introduced in Java 10.

One of the benefits is to help with readability of code. Another is to reduce boilerplate code.

It's called Local Variable Type Inference for a reason, because:

- It can't be used in field declarations on a class.
- It can't be used in method signatures, either as a parameter type or a return type.
- It can't be used without an assignment because the type can't be inferred in that case.
- It can't be assigned a null literal, again, because a type can't be inferred in that case.

# Run Time vs. Compile Time Typing

Are you still confused about the difference between run time and compile time typing?

You can think of the compile time type as the **declared** type.

This type is declared as a variable reference, a method return type, or a method parameter, for example.

In the case of Local Variable Type Inference (LVTI), we don't explicitly declare a type for the compiled reference type. Instead, it gets inferred by the compiler, but the byte code generated is the same as if we had declared the type.

# Run Time vs. Compile Time Typing

In many cases, the compile time type is the declared type to the left of the assignment operator.

What is returned on the right side of the assignment operator from whatever expression or method is executed, sometimes can only be determined at runtime, when the code is executing conditionally through the statements in the code.

You can assign a runtime instance to a different compile time type, only if certain rules are followed.

In this course, up to now, we've looked at only one rule that applies, and that's the inheritance rule.

We can assign an instance to a variable of the same type, or a parent type, or a parent's parent type, including java.lang.Object, the ultimate base class.

# Run Time vs. Compile Time Typing

Why are runtime types different than compile time types?

Because of polymorphism.

Polymorphism lets us write code once, in a more generic fashion, like the code we started this lecture with.

We saw that those two lines of code, using a single compile time type of Movie, actually supported four different runtime types.

Each type was able to execute behavior unique to the class.

# Evaluating what the runtime type is

How can we test the actual runtime type of a variable if its declared type is different?

We can determine the runtime type of an object in several ways.

# instanceof operator

The instanceof operator lets you test the type of an object or instance.

The reference variable you are testing is the left operand.

The type you are testing for is the right operand.

```
unknownObject instanceof Adventure
```

It's important to see that adventure is not in quotes, meaning, I'm not testing the type name but the actual type itself.

This operator returns true if unknownObject is an instance of Adventure.

# Pattern Matching for the instanceof Operator

If the JVM can identify that an object matches the type, it can extract data from the object without casting.

For this operator, the object can be assigned to a variable which I've called syfy.

From my example:

```
unknownObject instanceof ScienceFiction syfy
```

The variable syfy (if the instanceof method returns true) is already typed as a ScienceFiction variable.

# Polymorphism Challenge

Welcome to the Polymorphism Challenge video.

I've said that polymorphism means many forms.

What we want to do in this challenge is have our runtime code execute different behavior for different objects.

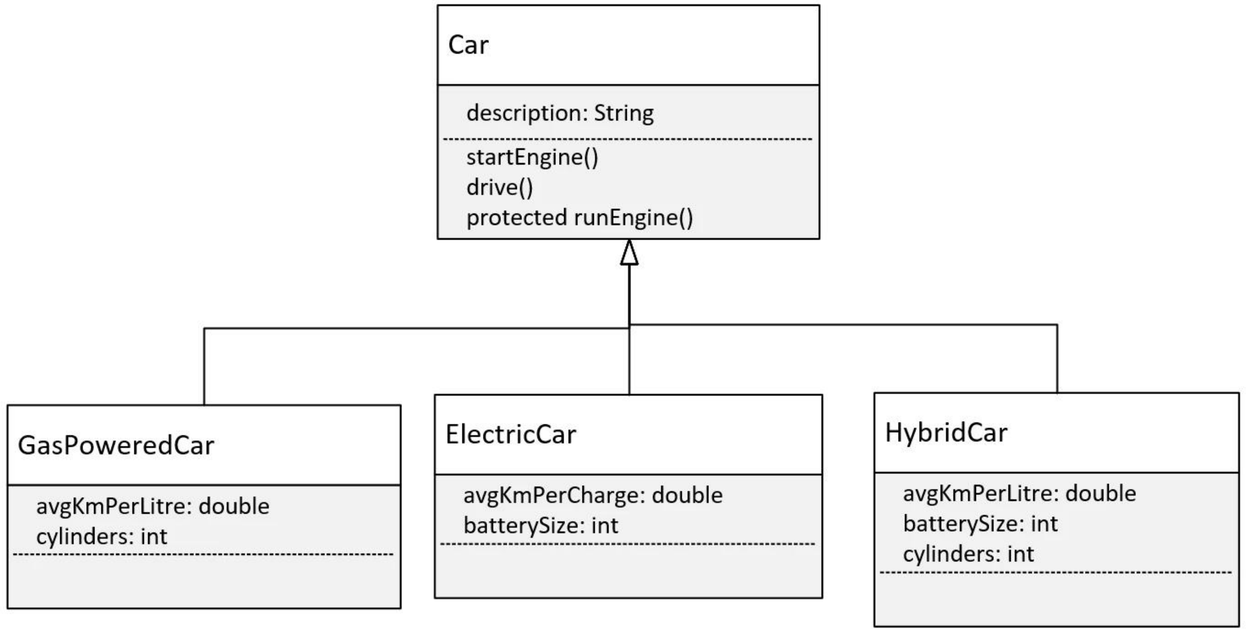
# The Challenge, the Car Class Diagram

Let's talk about the requirements for this 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.



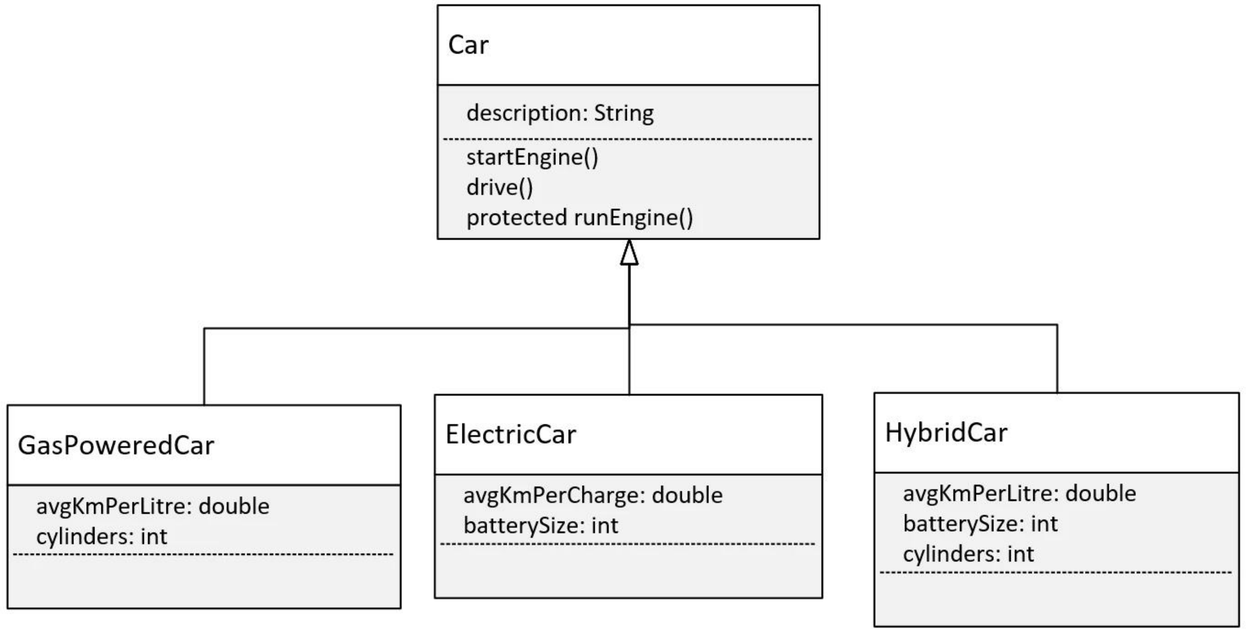
# The Challenge, the Car Class Diagram

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

You 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.

Each of these classes might also have different fields that might be used in those methods.

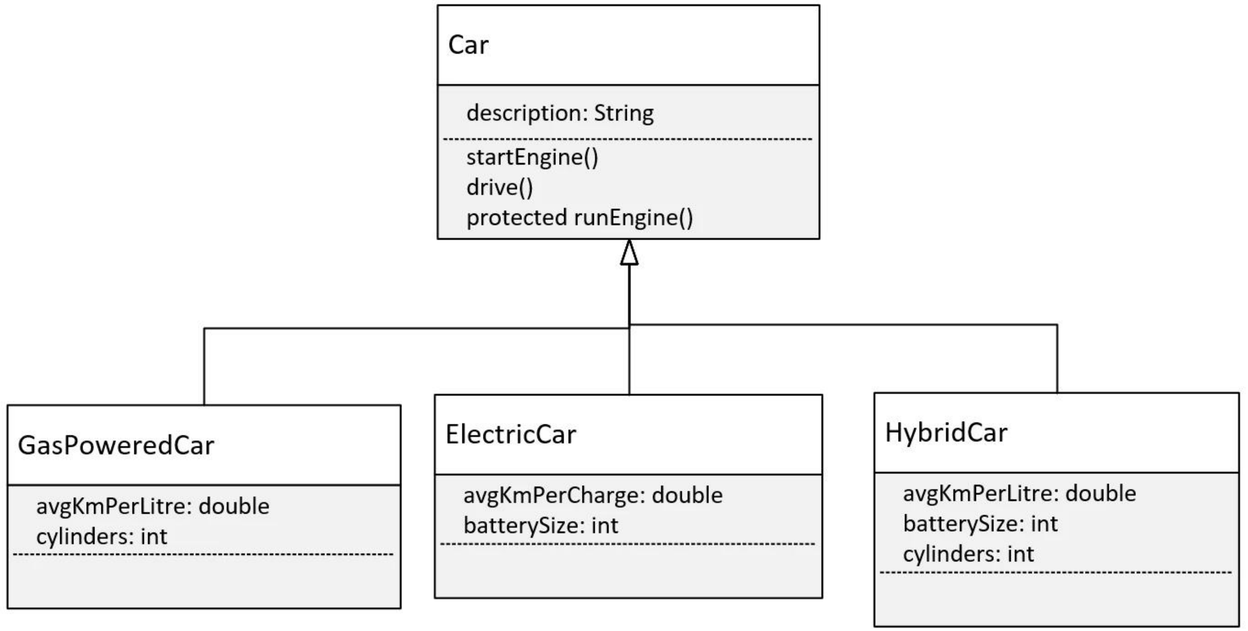


# The Challenge, the Car Class Diagram

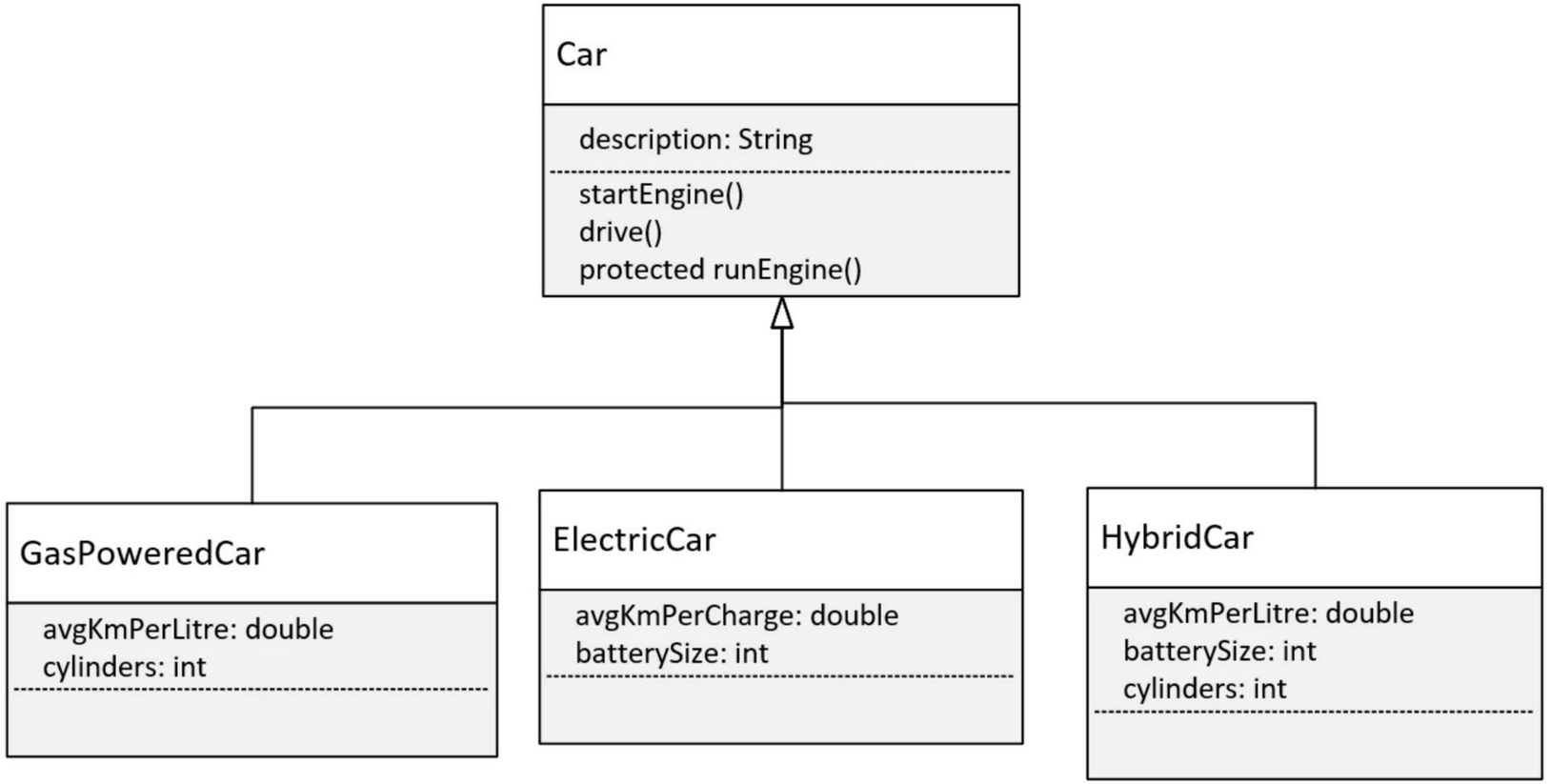
It's your job to create this class structure in Java and override methods appropriately.

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.

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



# The Challenge, the Car Class Diagram



# Welcome to the Object-Oriented Programming Master Challen

Congratulations, you've made it to the **object-oriented programming** master challenge.

In this challenge, you're going to build a complete application using all the **principles of object-oriented programming** that we have covered in the last two sections of this course.

The challenge is to write an application for a food restaurant.  Let's start by finding out some details about the restaurant and what is required.

# Bill's Burger Challenge

Bill runs a fast-food hamburger restaurant and sells hamburger meals.

His meal orders are composed of three items, the hamburger, the drink, and the side item.

Your challenge is to write an application to let Bill select the type of burgers and some of the additional items or extras that can be added to the burgers, as well as the actual pricing.

# Bill's Burger Challenge - The objects

You need a meal order.

- This should be composed of exactly one burger, one drink, and one side item.
- The most common meal order should be created without any arguments, like a regular burger, a small coke, and fries, for example.
- You should be able to create other meal orders by specifying different burgers, drinks, and side items.

You need a drink and side item.

- The drink should have at least a type, size, and price, and the price of the drink should change for each size.
- The side item needs at least a type and price.

# Bill's Burger Challenge - The objects

You need burgers.

- Every hamburger should have a burger type, a base price, and up to a maximum of three extra toppings.
- The constructor should include only the burger type and price.
- Extra Toppings on a burger need to be added somehow and priced according to their type.

The deluxe burger bonus.

- Create a deluxe burger meal with a deluxe burger that has a set price, so that any additional toppings do not change the price.
- The deluxe burger should have room for an additional two toppings.

# Bill's Burger Challenge - The functionality

Your main method should have code to do the following:

- Create a default meal that uses the no arguments constructor.
- Create a meal with a burger and the drink and side item of your choice, with up to 3 extra toppings.
- Create a meal with a deluxe burger where all items, drink, side item, and up to 5 extra toppings are included in the burger price.

# Bill's Burger Challenge - The functionality

For each meal order, you'll want to perform these functions:

- Add some additional toppings to the burger.
- Change the size of the drink.
- Print the itemized list. This should include the price of the burger, any extra toppings, the drink price based on size, and the side item price.
- Print the total due amount for the meal.

# Bill's Burger Challenge - The objects

You need a meal order.

- This should be composed of exactly one burger, one drink, and one side item.
- The most common meal order should be created without any arguments, like a regular burger, a small coke, and fries, for example.
- You should be able to create other meal orders by specifying different burgers, drinks, and side items.

You need a drink and side item.

- The drink should have at least a type, size, and price, and the price of the drink should change for each size.
- The side item needs at least a type and price.

# Bill's Burger Challenge - The objects

You need burgers.

- Every hamburger should have a burger type, a base price, and up to a maximum of three extra toppings.
- The constructor should include only the burger type and price.
- Extra Toppings on a burger need to be added somehow and priced according to their type.

The deluxe burger bonus.

- Create a deluxe burger meal with a deluxe burger that has a set price, so that any additional toppings do not change the price.
- The deluxe burger should have room for an additional two toppings.

# Initial Design Considerations

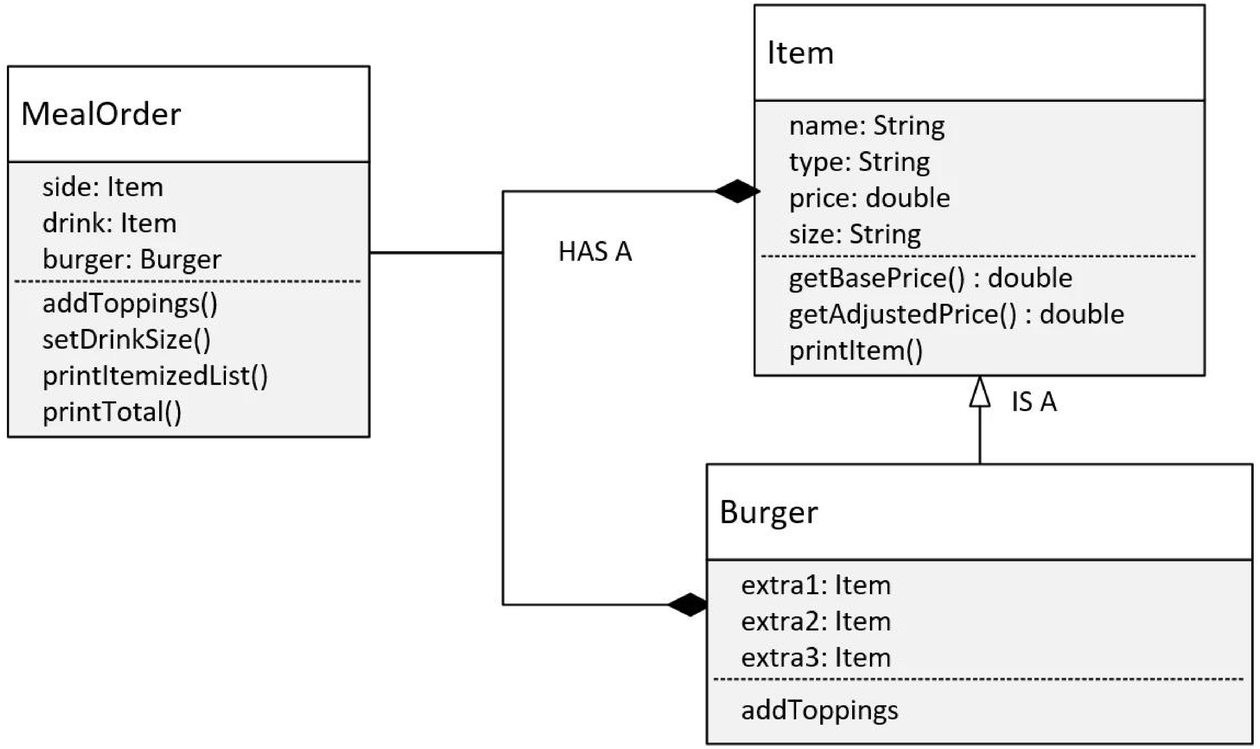
Let's start with the diagram of my design.

This diagram doesn't include the DeluxeBurger class. We'll look at that a bit later.

In my solution, the MealOrder class uses **composition** in the design. It's composed of a burger as well as a drink and a side, which will just be Items.

I've used **inheritance** for the Item and Burger relationships, which means Burger is an Item.

Every Item has a name, type, price or base price, and a size.

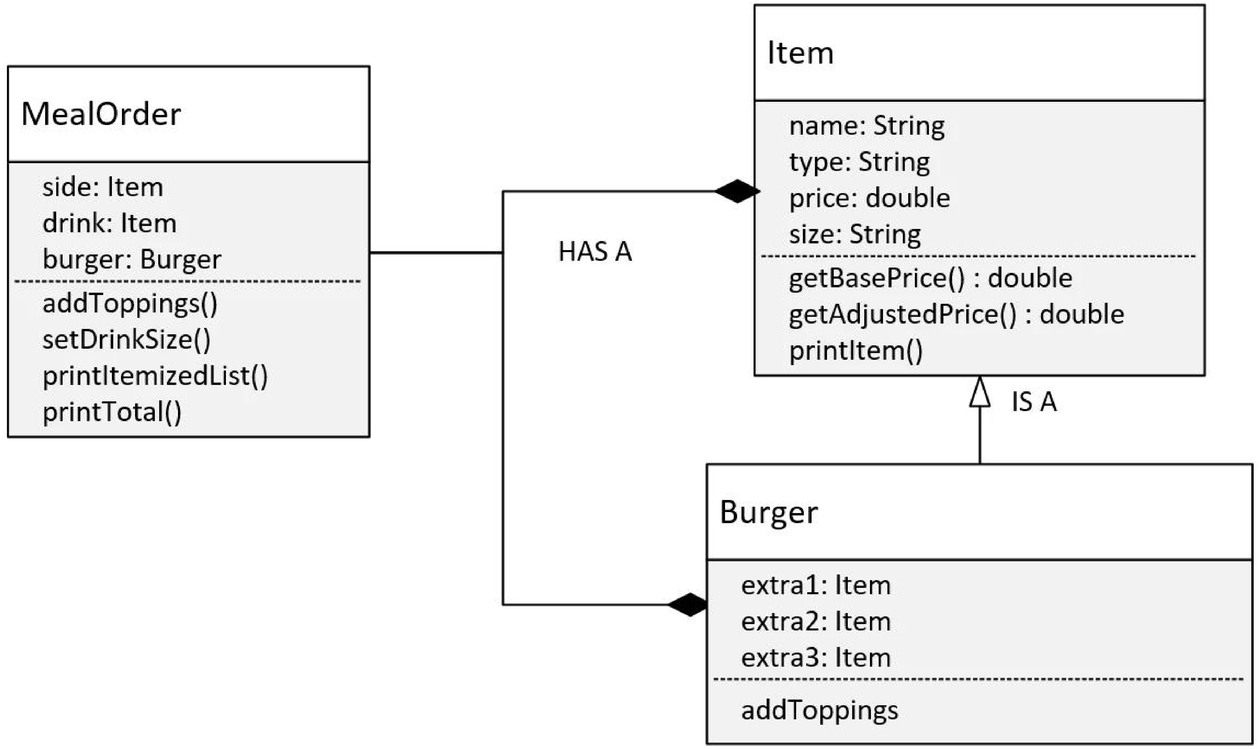


# Initial Design Considerations

Item has the method, getBasePrice, which is really just a getter method for the price, but the name is more descriptive.

Item also has getAdjustedPrice and the printItem method.

These methods will exhibit different behavior based on the runtime type, and we know that's **polymorphism**.

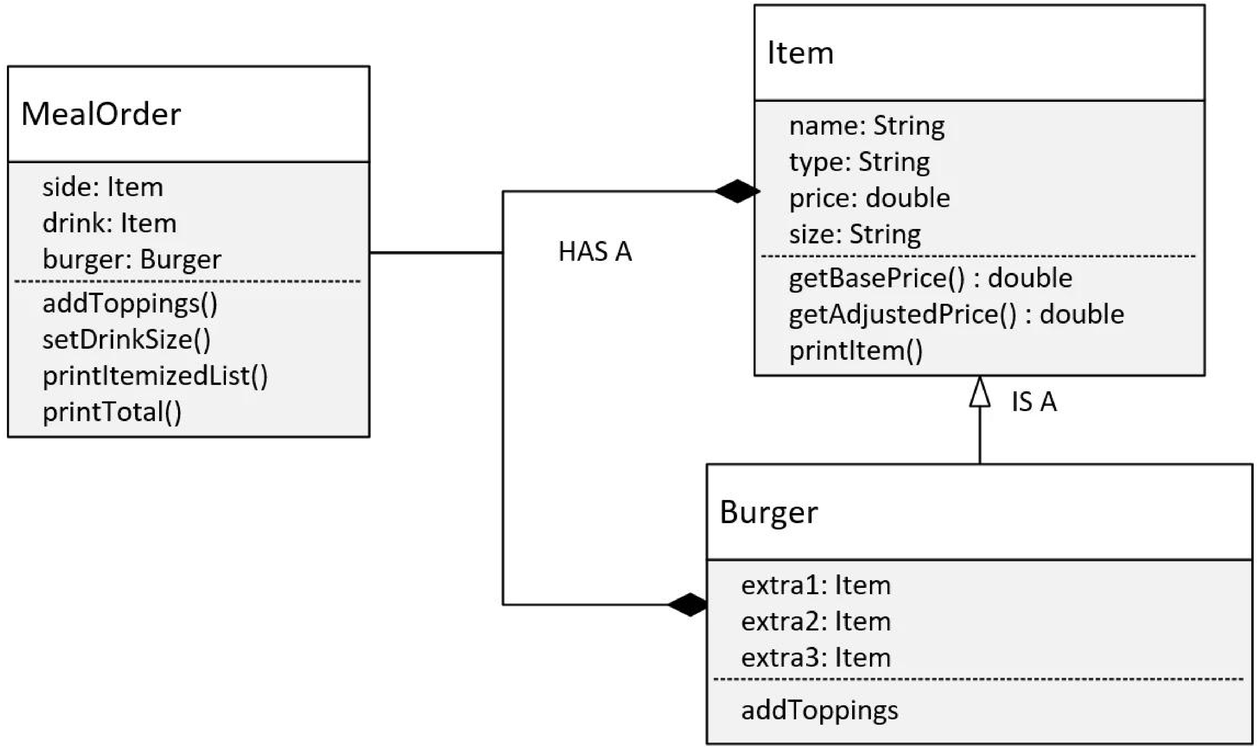


# Initial Design Considerations

For the burger, the toppings or extras are individual attributes and also have the type Item.

I'm going to use the MealOrder class to hide some of the implementation details from the calling code.

This means I'm going to use **encapsulation** techniques on MealOrder and Item.



# Welcome to the Object-Oriented Programming Master Challen

Congratulations, you've made it to the **object-oriented programming** master challenge.

In this challenge, you're going to build a complete application using all the **principles of object-oriented programming** that we have covered in the last two sections of this course.

The challenge is to write an application for a food restaurant.  Let's start by finding out some details about the restaurant and what is required.

# Bill's Burger Challenge

Bill runs a fast-food hamburger restaurant and sells hamburger meals.

His meal orders are composed of three items, the hamburger, the drink, and the side item.

Your challenge is to write an application to let Bill select the type of burgers and some of the additional items or extras that can be added to the burgers, as well as the actual pricing.

# Bill's Burger Challenge - The objects

You need a meal order.

- This should be composed of exactly one burger, one drink, and one side item.
- The most common meal order should be created without any arguments, like a regular burger, a small coke, and fries, for example.
- You should be able to create other meal orders by specifying different burgers, drinks, and side items.

You need a drink and side item.

- The drink should have at least a type, size, and price, and the price of the drink should change for each size.
- The side item needs at least a type and price.

# Bill's Burger Challenge - The objects

You need burgers.

- Every hamburger should have a burger type, a base price, and up to a maximum of three extra toppings.
- The constructor should include only the burger type and price.
- Extra Toppings on a burger need to be added somehow and priced according to their type.

The deluxe burger bonus.

- Create a deluxe burger meal with a deluxe burger that has a set price, so that any additional toppings do not change the price.
- The deluxe burger should have room for an additional two toppings.

# Bill's Burger Challenge - The functionality

Your main method should have code to do the following:

- Create a default meal that uses the no arguments constructor.
- Create a meal with a burger and the drink and side item of your choice, with up to 3 extra toppings.
- Create a meal with a deluxe burger where all items, drink, side item, and up to 5 extra toppings are included in the burger price.

# Bill's Burger Challenge - The functionality

For each meal order, you'll want to perform these functions:

- Add some additional toppings to the burger.
- Change the size of the drink.
- Print the itemized list. This should include the price of the burger, any extra toppings, the drink price based on size, and the side item price.
- Print the total due amount for the meal.

# Bill's Burger Challenge - The objects

You need a meal order.

- This should be composed of exactly one burger, one drink, and one side item.
- The most common meal order should be created without any arguments, like a regular burger, a small coke, and fries, for example.
- You should be able to create other meal orders by specifying different burgers, drinks, and side items.

You need a drink and side item.

- The drink should have at least a type, size, and price, and the price of the drink should change for each size.
- The side item needs at least a type and price.

# Bill's Burger Challenge - The objects

You need burgers.

- Every hamburger should have a burger type, a base price, and up to a maximum of three extra toppings.
- The constructor should include only the burger type and price.
- Extra Toppings on a burger need to be added somehow and priced according to their type.

The deluxe burger bonus.

- Create a deluxe burger meal with a deluxe burger that has a set price, so that any additional toppings do not change the price.
- The deluxe burger should have room for an additional two toppings.

# Initial Design Considerations

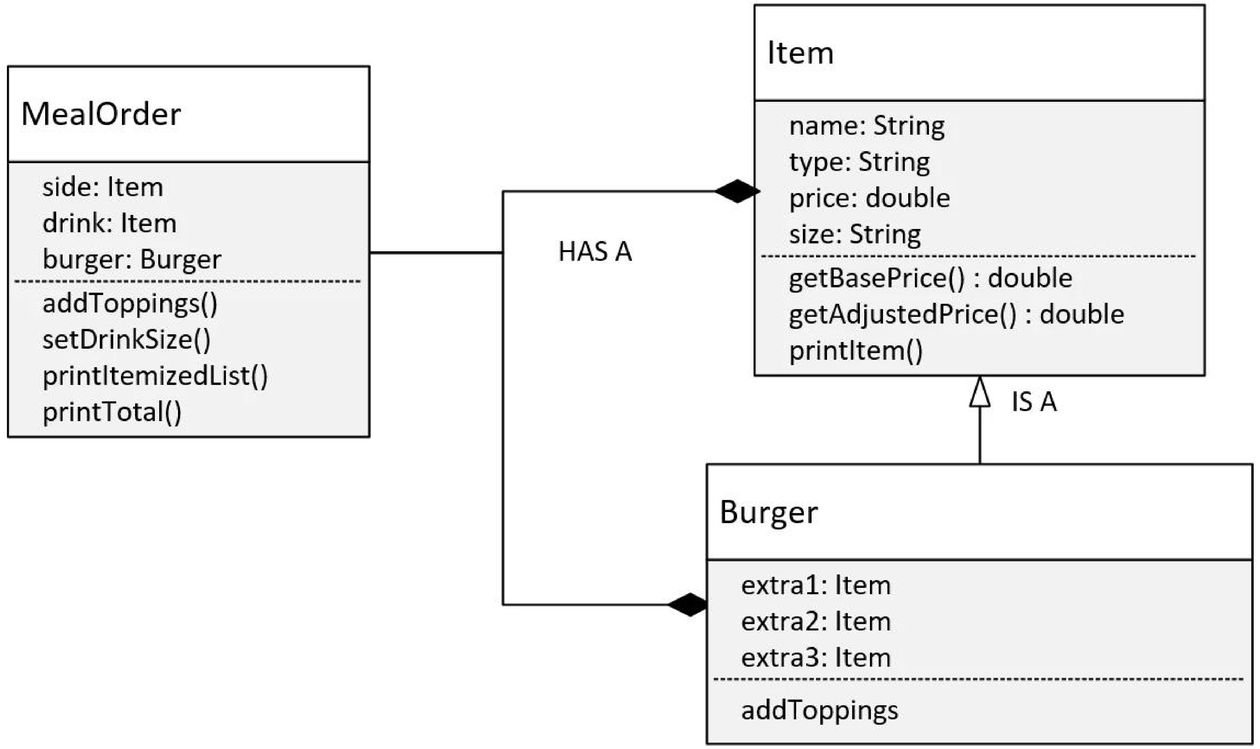
Let's start with the diagram of my design.

This diagram doesn't include the DeluxeBurger class. We'll look at that a bit later.

In my solution, the MealOrder class uses **composition** in the design. It's composed of a burger as well as a drink and a side, which will just be Items.

I've used **inheritance** for the Item and Burger relationships, which means Burger is an Item.

Every Item has a name, type, price or base price, and a size.

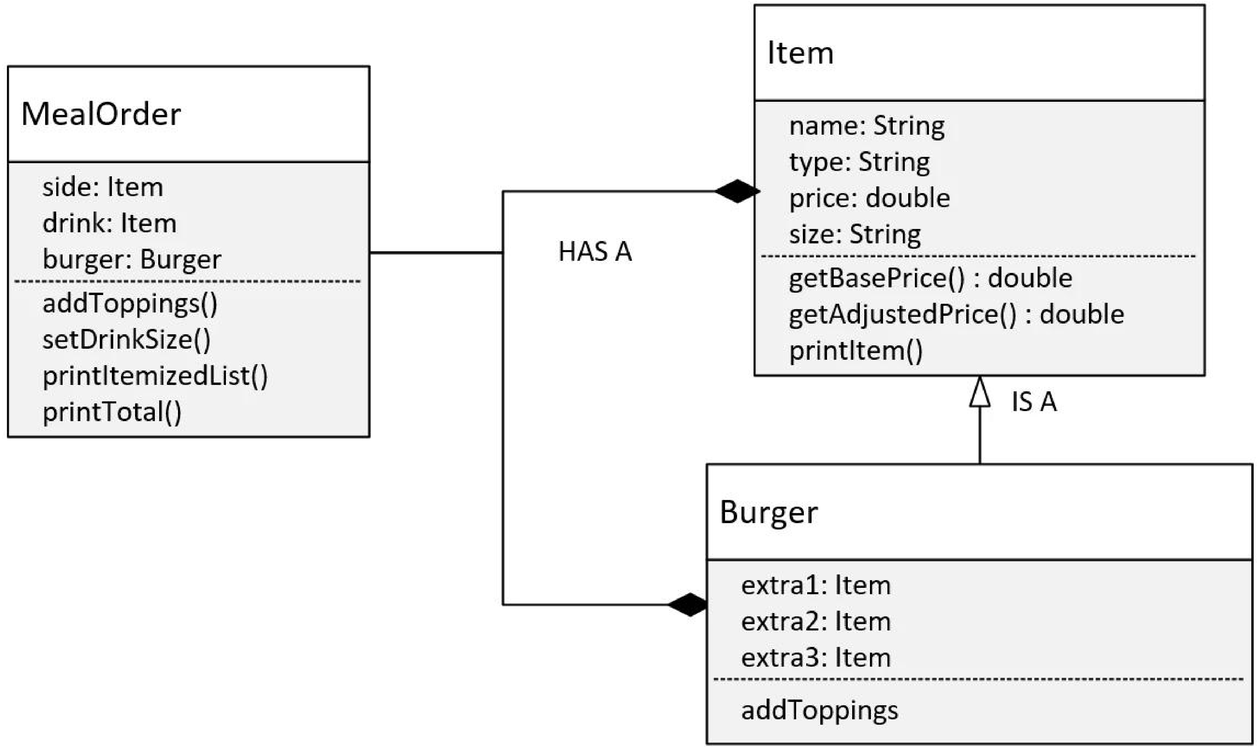


# Initial Design Considerations

Item has the method, getBasePrice, which is really just a getter method for the price, but the name is more descriptive.

Item also has getAdjustedPrice and the printItem method.

These methods will exhibit different behavior based on the runtime type, and we know that's **polymorphism**.

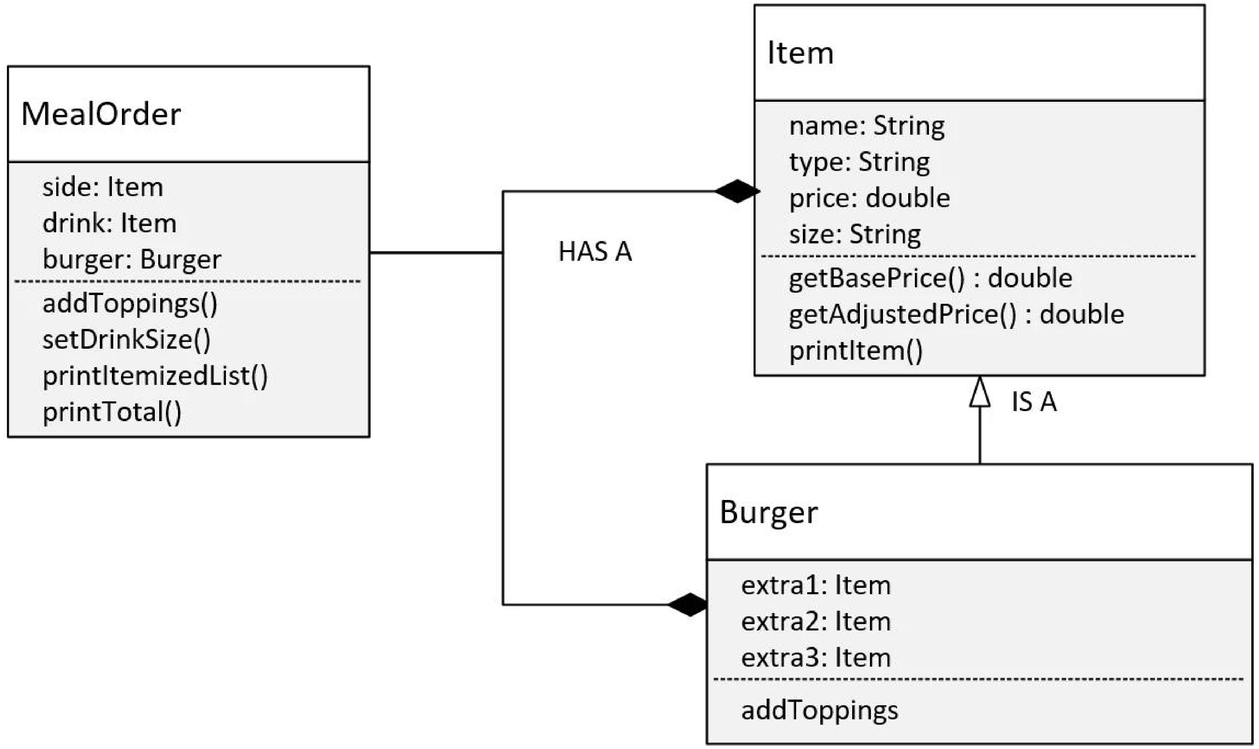


# Initial Design Considerations

For the burger, the toppings or extras are individual attributes and also have the type Item.

I'm going to use the MealOrder class to hide some of the implementation details from the calling code.

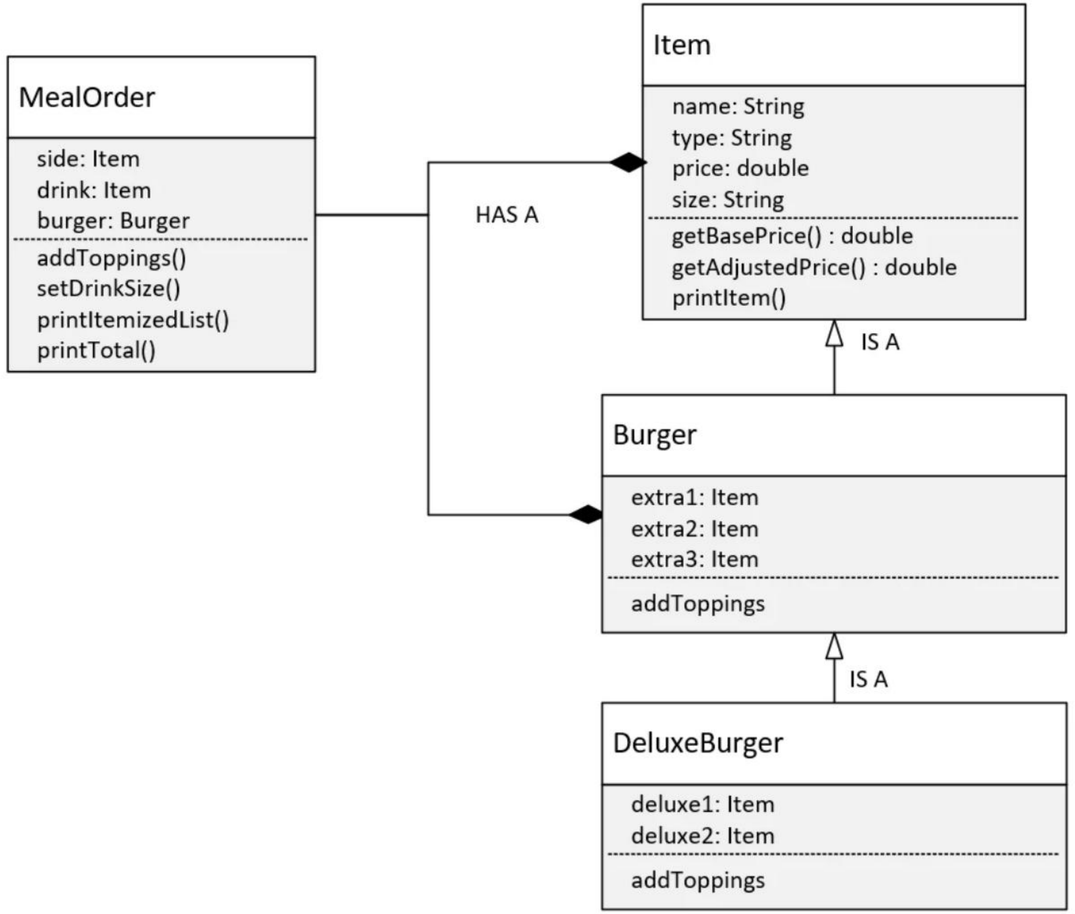
This means I'm going to use **encapsulation** techniques on MealOrder and Item.



# Bills Burger Challenge with the Bonus - the DeluxeBurger

In that last video, we created all the classes for a default meal order and provided methods for adding toppings and customizing a drink size.

Now, I want to create a burger subclass, the DeluxeBurger, which has two additional toppings, so a total of five, all up.



# Organizing Java Classes

Up until this point in the class, we haven't created a lot of classes, so we haven't had to think much about organizing those classes.

As the course progresses, we're going to be using more and more of Java's libraries, and our applications are going to get more complex.

This feels like a good time to talk about the package and import statements in more detail.

I've talked briefly about import statements when I used the Scanner class, and I mentioned packages when I talked about access modifiers.

In this video, I want to focus on what a package is, why we'll be using them moving forward, and how to access classes in different packages.

# package

As per the Oracle Java Documentation:

A package is a namespace that organizes a set of related types.

In general, a package corresponds to a folder or directory on the operating system, but this isn't a requirement.

When using an IDE like IntelliJ, we don't have to worry about how packages and classes are stored on the file system.

# package

The package structure is hierarchical, meaning you group types in a tree-like fashion.

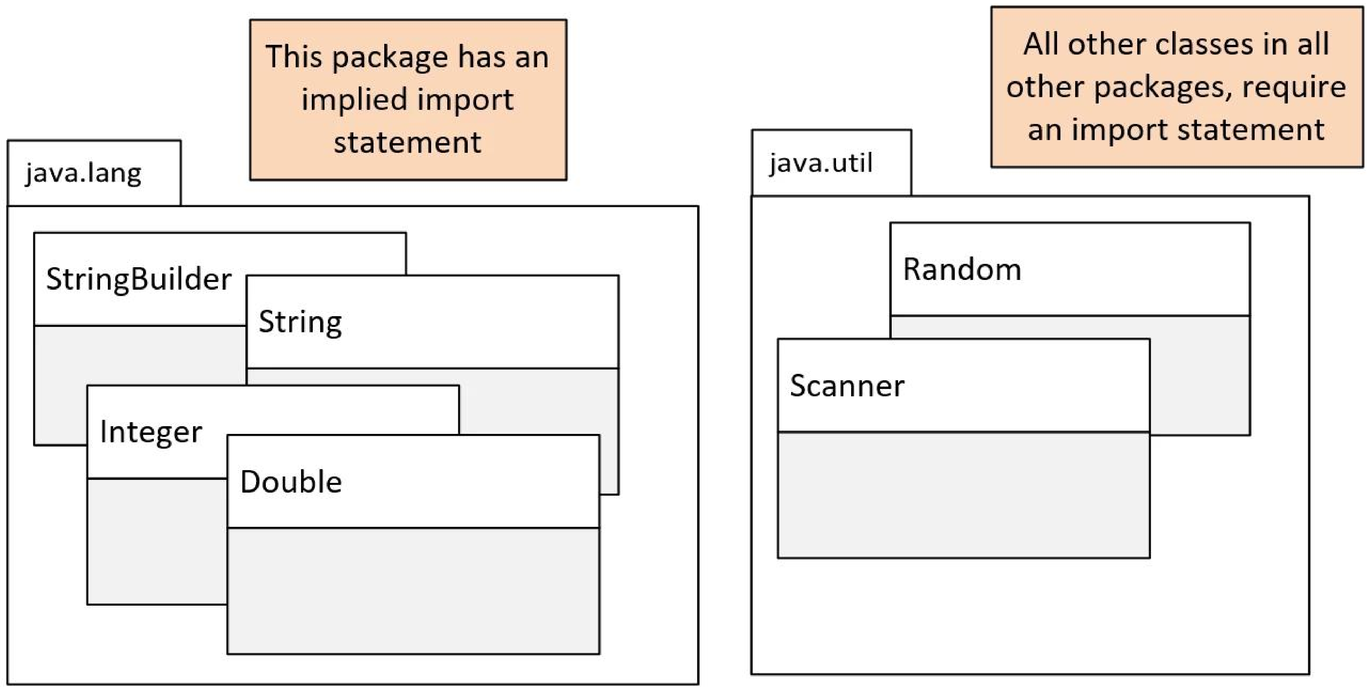
You can use any legal Java identifier for your package names, but common practice has package names in all lowercase.

The period separates the hierarchical levels of the package.

# Java packages

By now, you're familiar with two of Java's packages, java.lang and java.util.

On this slide, I'm showing these two packages with some of the classes we've used to date, as shown within these packages.



# Using classes from packages other than java.lang - the import staten

You may remember, when we used the Scanner and Random classes, we were required to use an import statement.

I'm showing you an example of using the Scanner class in this code.

The import statement has to be declared before any class or type declarations but after any package statement.

In this code, I don't have a package statement, so the import statement must be the first statement in the code.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
    }
}
```

# Multiple import statements

There is no limit to the number of import statements you can have.

This code shows two import statements, one for the Random class and one for the Scanner class.

```
import java.util.Random;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
    }
}
```

# Using import statements with wildcards

Alternately, we could use a wildcard with the asterisk character in the import statement.

In the following code, we're telling Java to import all classes from the java.util package with the use of the asterisks after the java.util package reference.

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
    }
}
```

# What is the purpose of packages?

Packages let us reuse common class names across different libraries or applications and provide a way to identify the correct class either with an import statement or a qualifying name.

For example, you might have a package for utility classes that can provide common functionality for all of your classes to access.

Packages let us organize our classes by functionality or relationships.

Packages also let us encapsulate our classes from classes in other packages.

You might have a package of tightly coupled classes that should only be accessed by each other but not by the outside world, as an example.

# What would a package name look like?

We've seen that Java starts their package names with java in some of the examples we've looked at.

However, it is common practice to use the reverse domain name to start your own package naming conventions.

If your company is abccompany.com, for example, your package prefixes would be com.abccompany.

For the rest of the course, I'll be using dev.lpa, which is the reverse domain of one of my Learn Programming Academy domains.

The package name hierarchy is separated by periods.

# using the package statement

The package statement needs to be the first statement in the code, with the exception of comments and whitespace.

The package statement comes before any import statements.

There can be only one package statement because a class or type can only be in a single package.

```
package dev.lpa.package_one;

import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
    }
}
```

# The Fully Qualified Class Name (FQCN)

In this code, I created a class called Main.

A class's fully qualified class name (FQCN) consists of the package name and the class name.

So, this class's fully qualified name is dev.lpa.package_one.Main.

```
package dev.lpa.package_one;

import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
    }
}
```

# The Fully Qualified Class Name (FQCN)

It's unlikely, this class, with its fully qualified name, will have a naming conflict with a Main class in another package.

As another example, the fully qualified class name of the Scanner class in this code is java.util.Scanner.

```
package dev.lpa.package_one;

import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
    }
}
```

# The Fully Qualified Class Name vs the import statem

You can use the fully qualified class name instead of the import statement, as I show you here on this slide.

This code does not use the import statement, but instead, where the Scanner class is referenced, the fully qualified class name is used.

```
package dev.lpa.package_one;

public class Main {

    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

    }
}
```

# The Fully Qualified Class Name vs the import statem

You can probably imagine this could get very tedious, very quickly, if you wrote your code this way.

Later in the course, I'll talk about using a combination of the import statement and the fully qualified class name to resolve conflicts.

```
package dev.lpa.package_one;

public class Main {

    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

    }
}
```

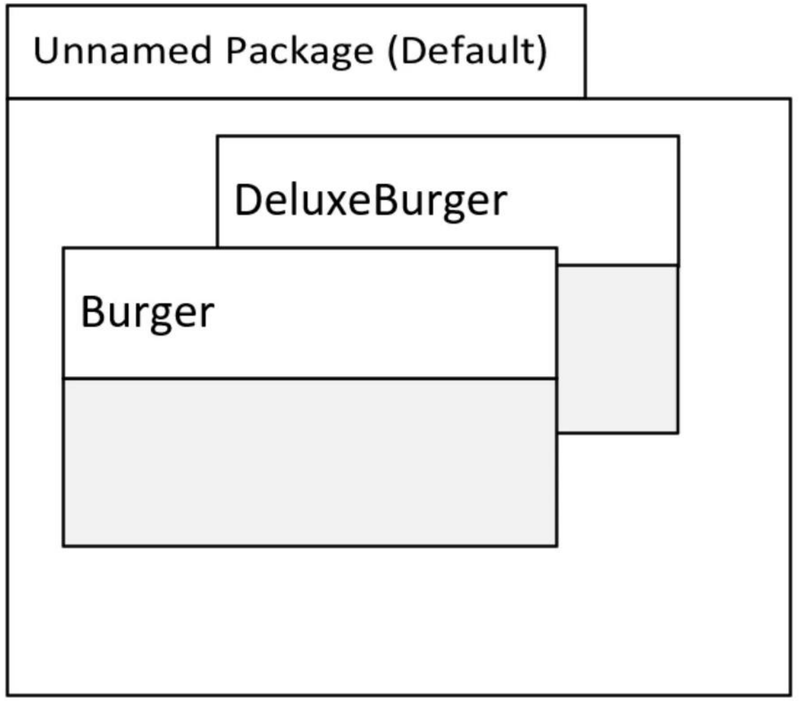
# Using the package statement

In this slide, I'm showing two classes from our previous challenge in the unnamed or default package.

This is where classes are implicitly placed if we don't specify a package statement.

For your applications, you should always specify a package statement and avoid using the default or unnamed package.

Although that's all we've been using up until now, it has some disadvantages when you work in a true development environment.



# Using the package statement

The main disadvantage is that you can't import types from the default package into other classes outside of the default package.

In other words, you can't qualify the name if it's in the default package, and you can't import classes from the default package.