Generics Introduction

Welcome to a new section focusing on another crucial topic in Java: Generics.

If you've been following this course sequentially, you should now be comfortable with generalizing your class designs conceptually.

This involves identifying commonalities among your classes to think about them in a broader sense.

Generics enable us to create and design classes in a generalized manner, without being concerned about the specific details of the elements they might contain.

A prime example of a generic class in Java is the ArrayList.

Generics Introduction

We can use an ArrayList for any type of object because many of its methods are applicable to any type.

Let's dive in to learn how to create generic types, explore their usage, and understand when they should be used.

What are Generics?

Java supports generic types, such as classes, records and interfaces.

It also supports generic methods. Sound confusing?

Declaring a Class vs. Declaring a generic Class

On this slide, I'm showing you a regular class declaration, next to a generic class.

The thing to notice with the generic class, is that the class declaration has angle brackets with a T in them, directly after the class name.

T is the placeholder for a type that will be specified later.

This is called a type identifier, and it can be any letter or word, but T which is short for Type is most commonly used.

Regular Class	Generic Class

`class ITellYou {`
 	
`    private String field;`
 	
`}`
	`class YouTellMe<T> {`
	
	`    private T field;`
	
	`}`

Declaring a Class vs. Declaring a generic Class

For the generic class, the field's type is that placeholder, just T, and this means it can be any type at all.

The T in the angle brackets means it's the same type as the T, specified as the type of the field.

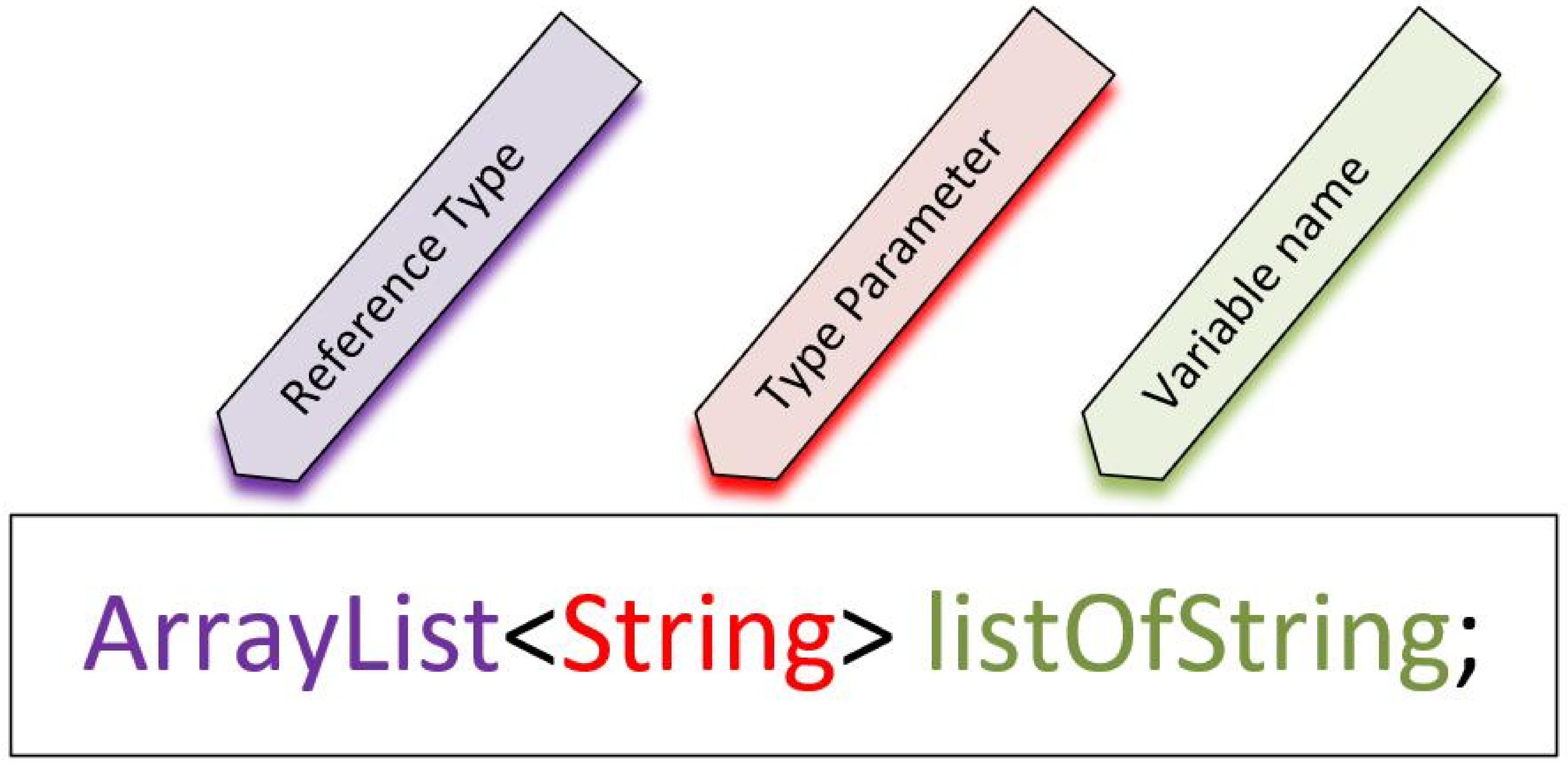
Regular Class	Generic Class
`class ITellYou {` `private String field;` `}`	`class YouTellMe<T> {` `private T field;` `}`

Using a generic class as a reference type

On this slide, I have a variable declaration of the generic ArrayList class.

In the declaration of a reference type that uses generics, the type parameter is declared in angle brackets.

The reference type is ArrayList, the type parameter (or parameterized type) is String, which is declared in angle brackets, and listOfString is the variable name.

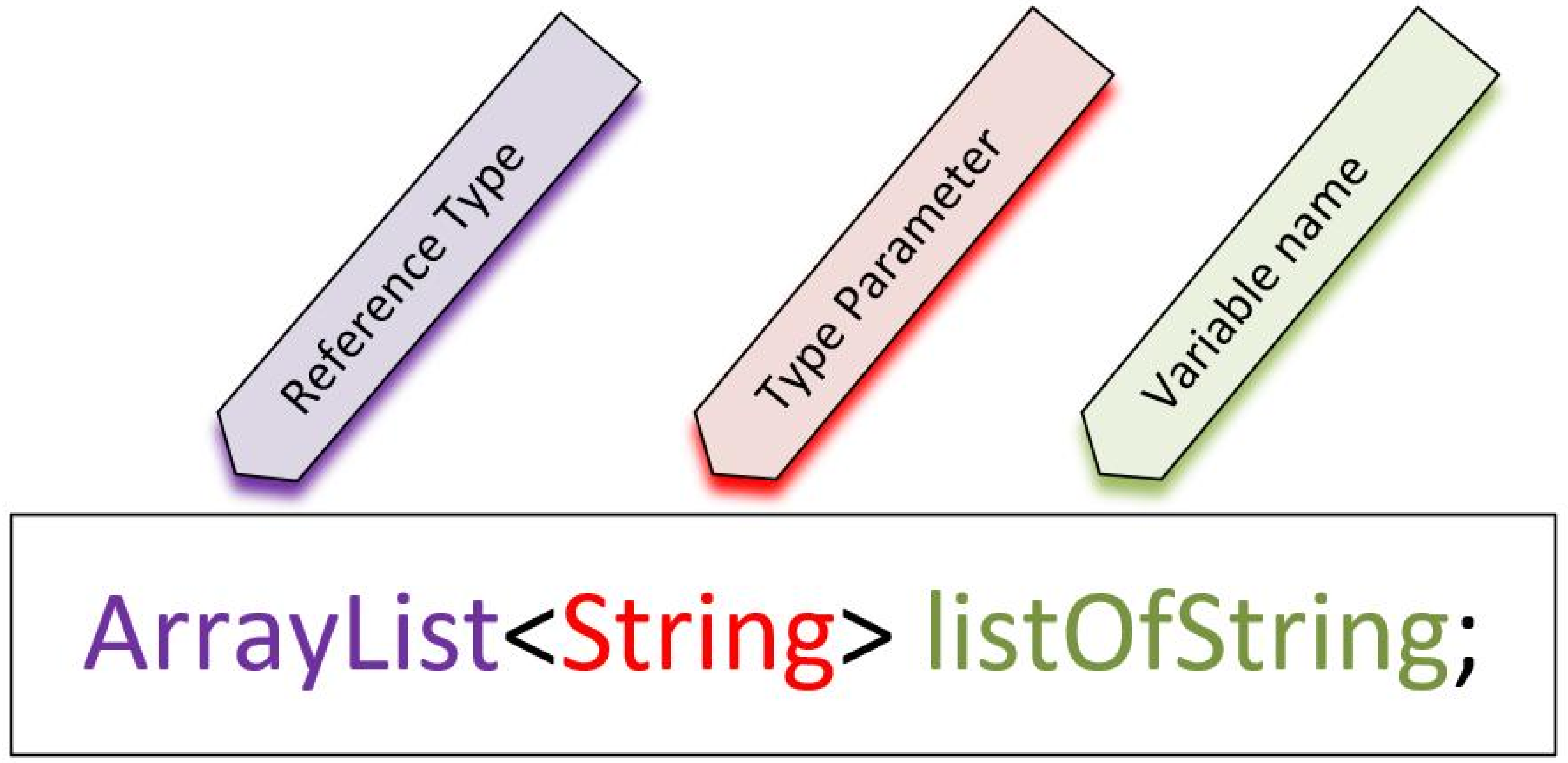


```
ArrayList<String> listOfString;
```

Using a generic class as a reference type

Many of Java's libraries are written using generic classes and interfaces, so we'll be using them a lot moving forward.

It's still a good idea to learn to write your own generic class though, to help you understand the concept.



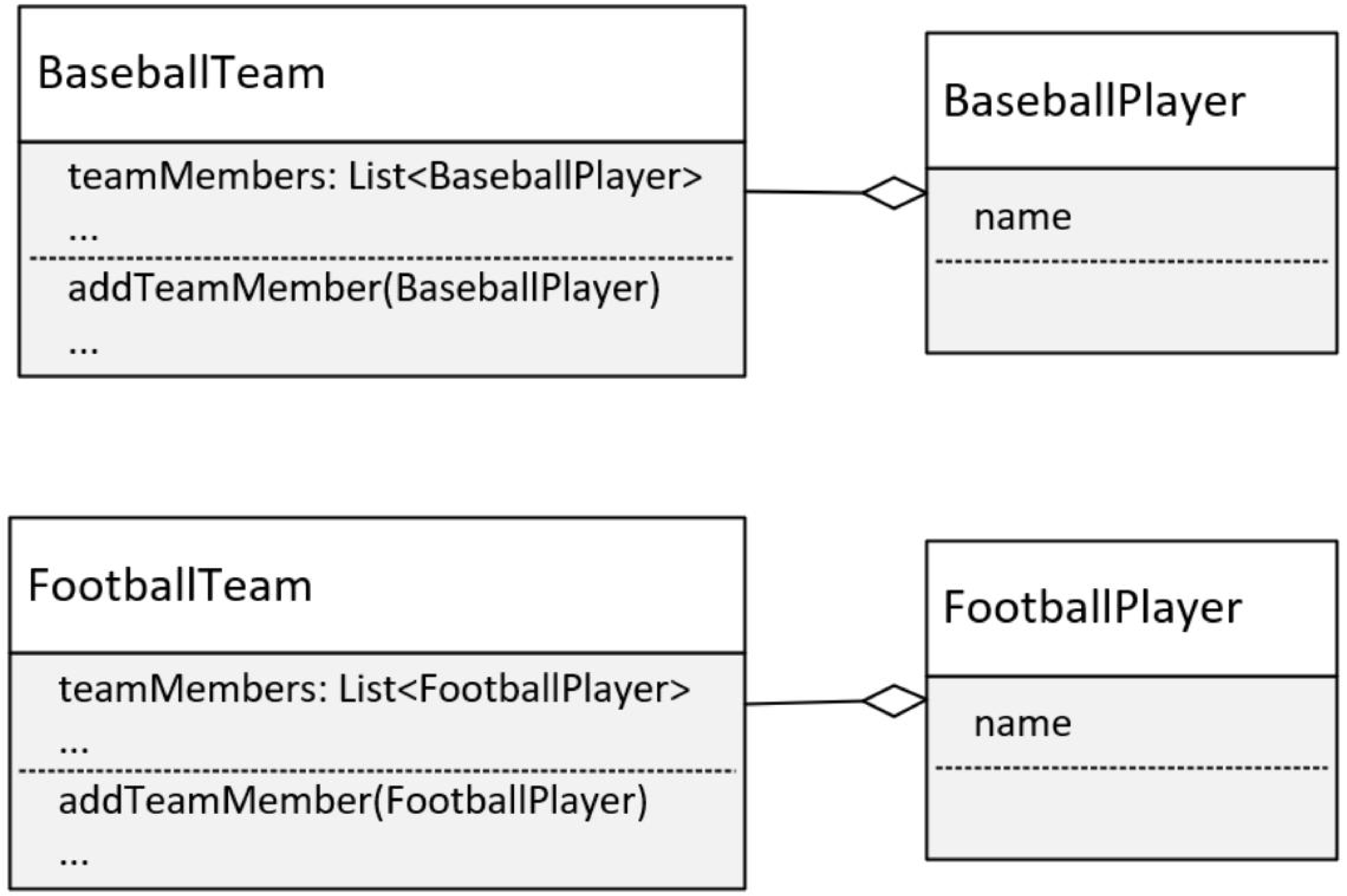
```
ArrayList<String> listOfString;
```

Solution 1: Duplicate code

I could copy and paste the BaseballTeam, and rename everything for FootballTeam, and create a FootballPlayer, as I'm showing on this slide.

This means I'd have to make sure any changes I make to one team or player that made sense for the other team and player, had to be made in both sets of code.

This is rarely a recommended approach, unless team operations are significantly different.


BaseballTeam
teamMembers: List<BaseballPlayer>
...
addTeamMember(BaseballPlayer)
...
BaseballPlayer
name
FootballTeam
teamMembers: List<FootballPlayer>
...
addTeamMember(FootballPlayer)
...
FootballPlayer
name

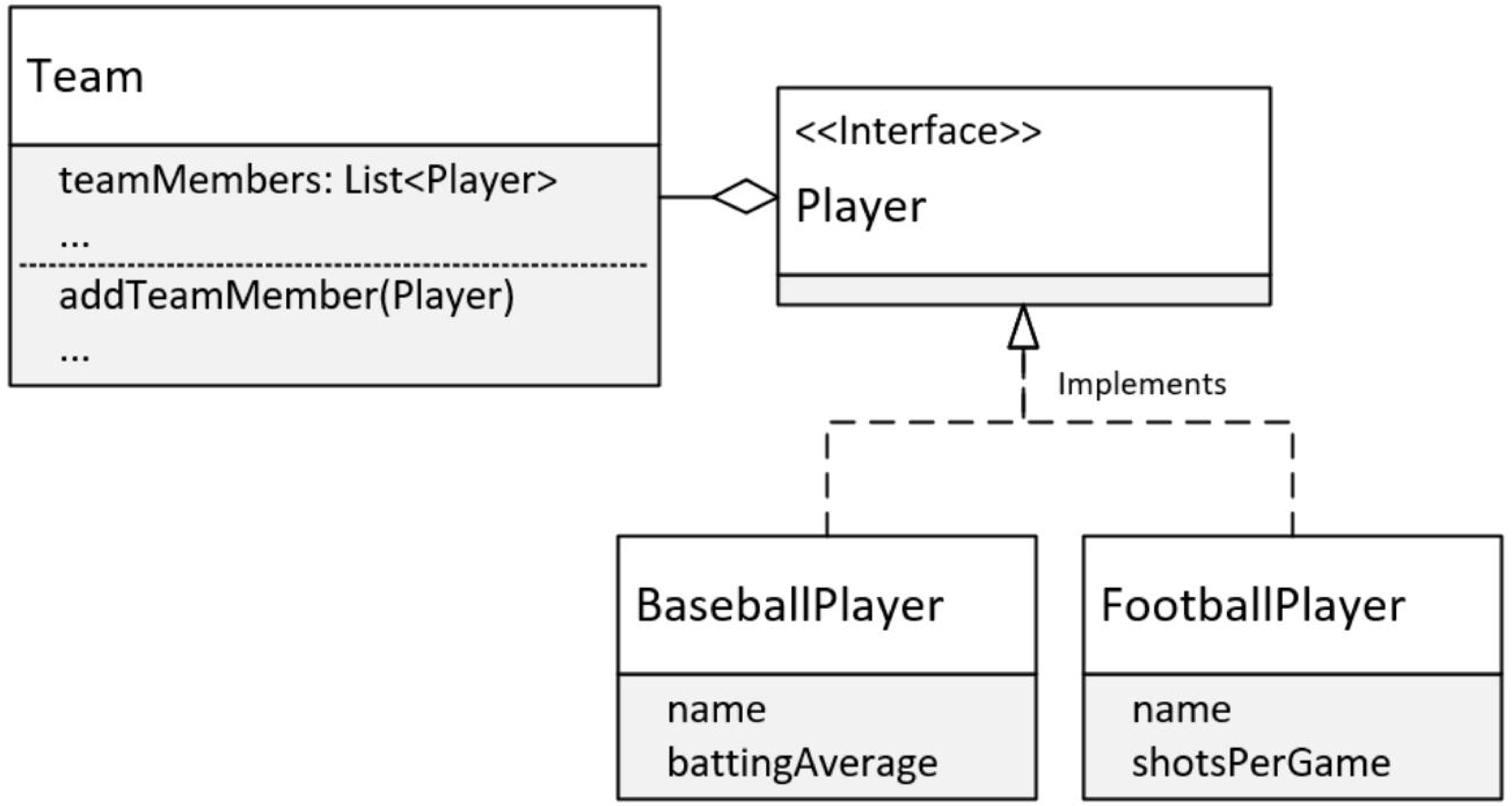
Solution 2: Use a Player Interface or abstract class to support different types of

I could change Baseball team to simply Team and use an interface type (or either an abstract or base class) called Player.

On this slide, I show a Team Class and on this class the members are a List of Players.

I've made Player an interface and have BaseballPlayer and FootballPlayer classes that implement that interface.

This is a better design than the previous one, but it's still got problems.



Generic Type Parameters

I've already shown you that one way to declare a generic class, is to include a type parameter which I show here, in the angle brackets.

```java
public class Team<T> {
```

Now, using T is just a convention, short for whatever type you want to use this Team class for.

But you can put anything you want in there.

Single letter types are the convention however, and they're a lot easier to spot in the class code, so I encourage you to stick to this convention.

Generic Type Parameters

You can have more than one type parameter, so I could have T1, T2 and T3, etc.

```java
public class Team<T1, T2, T3> {
```

Convention says that instead of using type parameters like this, it's easier to read the code with alternate letter selections.

And these are usually S, U, and V, in that order.

So, if we had three types, you'd probably want to declare this class as shown here, with T, S, and U.

```java
public class Team<T, S, U> {
```

Generic Type Parameters

A few letters are reserved for special use cases.

The most commonly used type parameter identifiers are:

- E for Element (used extensively by the Java Collections Framework).
- K for Key (used for mapped types).
- N for Number.
- T for Type.
- V for Value.
- S, U, V etc. for 2nd, 3rd, 4th types.

Raw usage of generic classes.

When you use generic classes, either referencing them or instantiating them, it's definitely recommended that you include a type parameter.

But you can still use them without specifying one. This is called the **Raw Use** of the reference type.

The raw use of these classes is still available for backwards compatibility, but it's discouraged for several reasons.

- Generics allow the compiler to do compile-time type checking when adding and processing elements in the list.
- Generics simplify code, because we don't have to do our own type checking and casting, as we would if the type of our elements was Object.

Generic classes can be bounded, limiting the types that can use it.

On this slide, I'm showing the code from my class.

This extends keyword doesn't have the same meaning as extends when it's used in a class declaration.

This isn't saying our type T extends Player although it could.

This is saying the parameterized type T, has to be a Player or a **subtype** of Player.

Player in this case could have been either a class or an interface, the syntax would be the same.

This declaration establishes what is called an **upper bound** on the types that are allowed to be used with this class.

```
public class Team<T extends Player> {
```

Why specify an upper bound?

An upper bound permits access to the bounded type's functionality.

An upper bound limits the kind of type parameters you can use when using a generic class. The type used must be equal to or a subtype of the bounded type.

Generic Class Challenge

Now it's your turn to create your own generic class.

In the Interface Challenge, I created a Mappable Interface and introduced you to different Map geometry types, POINT, LINE, and POLYGON.

The challenge then created a map marker or icon, and a map label, but didn't do anything with locations.

In this challenge, you'll use another Mapping example, but use location data in the output.

As you are probably aware, you can use Google Maps to determine the location of any point on a map.

The Generic Class challenge

You'll start with a **Mappable Interface** that has one abstract method called render.

You'll create two classes **Point** and **Line**, that implement this interface.

You'll also create **two specific classes** that extend each of these, for a mappable item of interest.

The Generic Class challenge

In my solution, I'll be mapping US National Parks and a couple of major rivers in the US. The parks will be points and the rivers will be lines.

The data I'll be using is shown here.

I'll be creating a **Park** class that extends Point, and a **River** class that extends Line, to support this data.

US National Parks & selected locations

Name	Type	Googled Locations
Yellowstone	National Park	44.4882, -110.5916
Grand Canyon	National Park	36.0636, -112.1079
Yosemite	National Park	37.8855, -119.5360

US Rivers & selected locations

Name	Type	Googled Locations
Mississippi	River	47.2160, -95.2348 35.1556, -90.0659 29.1566, -89.2495
Missouri	River	45.9239, -111.4983 38.8146, -90.1218
Colorado	River	40.4708, -105.8286 36.1015, -112.0892 34.2964, -114.1148 31.7811, -114.7724
Delaware	River	42.2026, -75.00836 39.4955, -

The Generic Class challenge

You should have constructors or methods to support adding a couple of attributes, and some location data to your two specific classes.

Name	Googled Location of a Point
Yellowstone National Park	44.4882, -110.5916

You can pass the location data of a point type, as a String, or a set of double values, representing latitude and longitude.

You can pass the multiple locations of a line, as a set of strings, or a two-dimensional array of doubles that represents the multiple points on your line.

Name	Googled Locations of Points in a River
Mississippi River	47.2160, -95.2348 35.1556, -90.0659 29.1566, -89.2495

The Generic Class challenge

In addition to these classes, you'll create a **generic class** called **Layer**.

Your Layer class should have **one type parameter** and should only **allow Mappable elements** as that type.

This generic class should have a **single private field**, a **list of elements** to be mapped.

This class should have a method or constructor, or both, to add elements.

You should create a method, called renderLayer that loops through all elements and executes the method render on each element.

The Generic Class challenge

Your main method should create some instances of your specific classes that include some location data.

These should get added to a typed Layer, and the renderLayer method called on that.

Sample output is shown here:

```
Render Grand Canyon National Park as POINT ([40.1021, -75.4231])
Render Mississippi River as LINE ([[47.216, -95.2348], [29.1566, -89.2495], [35.1556, -90.0659]])
```

The plan – the class diagram

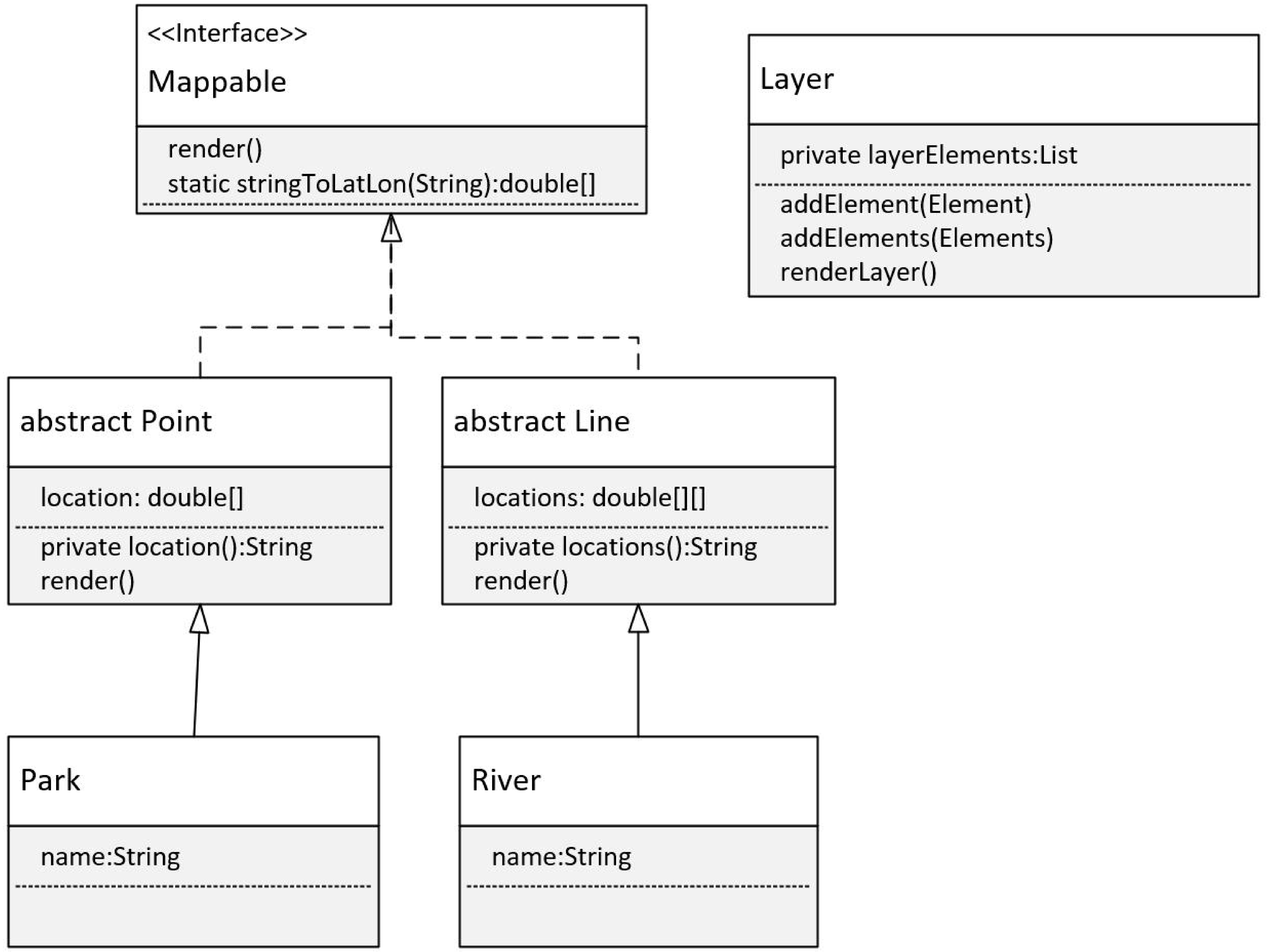
This diagram shows how I plan to build this.

You can see my Mappable interface has the method render on it, and by default that's both public and abstract.

I've also added a static method that will take a String, and split it into a double array, which will have the latitude and longitude values in them.

I've made two classes, Point and Line, abstract because I don't really want anyone to instantiate these classes.

Point has a location field, which is a double array, and will just have two doubles the latitude and longitude.



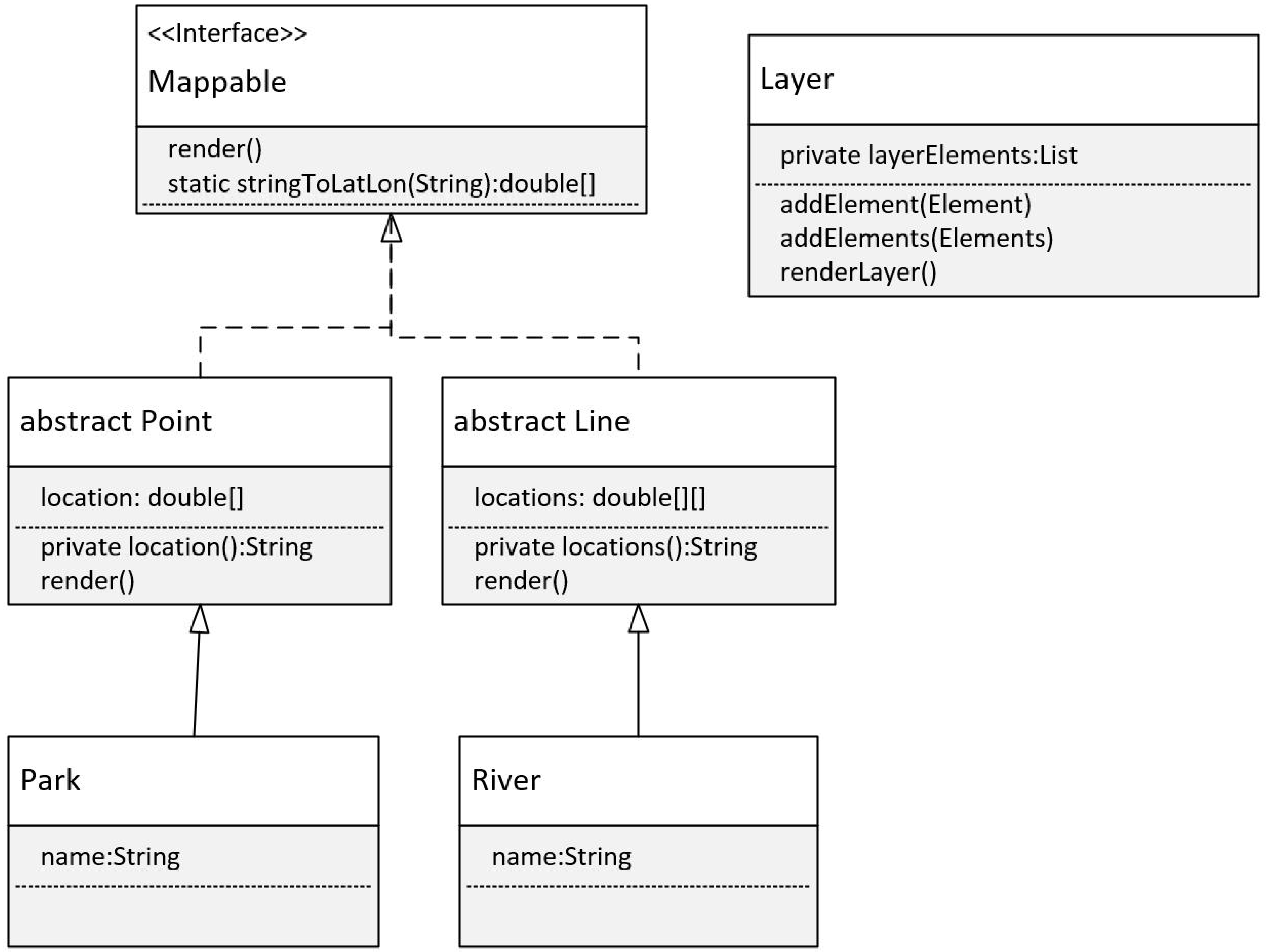
The plan - the class diagram

I also have a method that will print that array to a string, called location.

And then the render method is implemented.

And the same with abstract Line, except a line will have multiple latitude, longitude pairs, and these will be represented as a two-dimensional array.

Then I'm going to have a Park class extend Point with just a name field.

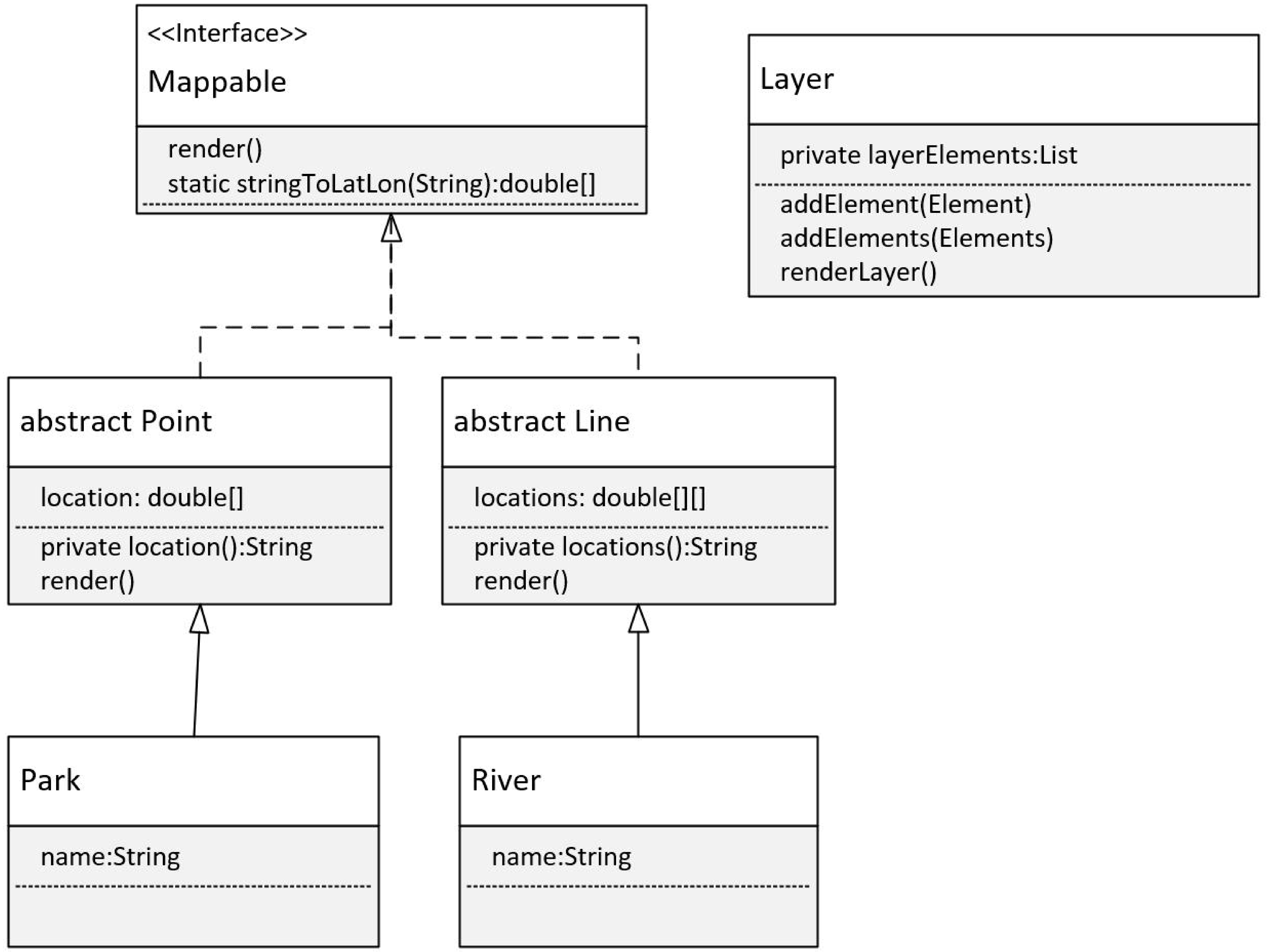


The plan – the class diagram

River will extend Line and that has just a name as well, for simplicity.

Lastly, there's the Layer class. This is the generic class, and it has a list of layerElements and methods to add one or more of these.

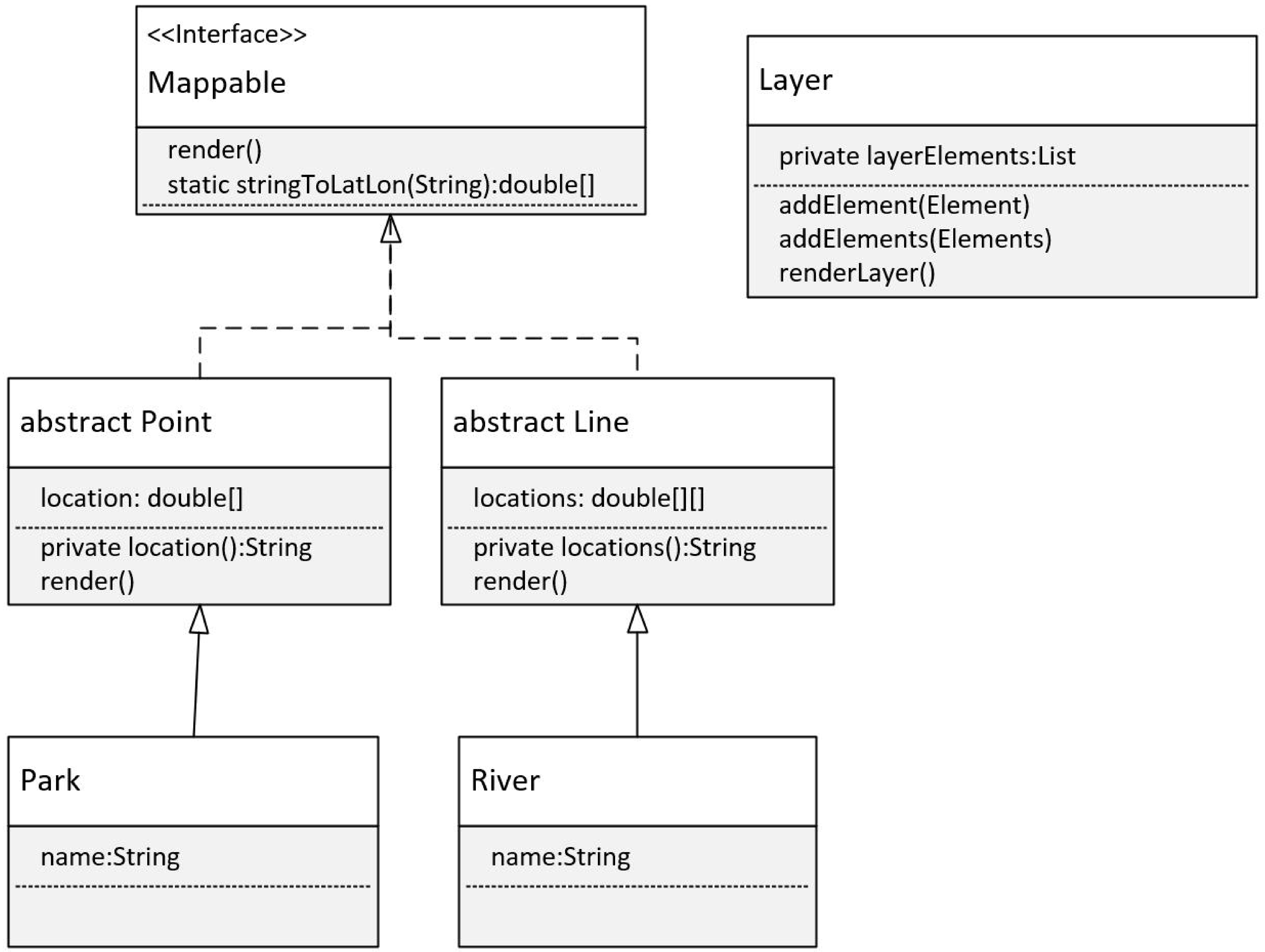
It also has the method, renderLayer.



Generic Class Challenge, Part 2

I'm showing my class diagram here again, as a reminder of what I'm working towards.

I'll start with the Park class that extends Point.



Interfaces used for sorting

Now that I've covered interfaces and generic classes, I want to review interfaces we used in previous lectures, in more detail.

The first is Comparable.

For an array, we can simply call Arrays.sort and pass it an array, but as I've previously mentioned, the elements in the array need to implement Comparable.

Types like String or primitive wrapper classes like Integer or Character are sortable, and this is because they do actually implement this interface.

Comparable Interface

The interface declaration in Java.

```java
public interface Comparable<T> {

    int compareTo(T o);
}
```

It's a generic type, meaning it's parameterized.

Any class that implements this interface, needs to implement the compareTo method.

Comparable Interface

```
public interface Comparable<T> {

    int compareTo(T o);
}
```

This method takes one object as an argument, shown on this slide as the letter o, and compares it to the current instance, shown as this.

The table on this slide shows what the results of the compareTo method should produce, when implemented.

resulting Value	**Meaning**
zero	`O == this`
negative value	`this < o`
positive value	`this > o`

The Comparator Interface

The Comparator interface is similar to the Comparable interface, and the two can often be confused with each other.

Its declaration and primary abstract method are shown here, in comparison to Comparable.

Notice that the method names are different, compare vs. compareTo.

Comparator	Comparable

`public interface Comparator<T> {`
 	
`    int compare(T o1, T o2);`
`}`
	`public interface Comparable<T> {`
	
	`    int compareTo(T o);`
	`}`

The Comparator Interface

The compare method takes two arguments vs. one for compareTo, meaning that it will compare the two arguments to one another and not one object to the instance itself.

I'll review Comparator in code, but in a slightly manufactured way.

It's common practice to include a Comparator as a nested class.

Comparator	**Comparable**
`public interface Comparator<T> {`	`public interface Comparable<T> {`
`    int compare(T o1, T o2);`	`    int compareTo(T o);`
`}`	`}`

Summary of Differences

Comparable	Comparator
`int compareTo(T o);`	`int compare(T o1, T o2);`
Compares the argument with the current instance. Called from the instance of the class that implements Comparable. Best practice is to have this.compareTo(o) == 0 result in this.equals(o) being true. Arrays.sort(T[] elements) requires T to implement Comparable.	Compares two arguments of the same type with each other. Called from an instance of Comparator. Does not require the class itself to implement Comparator, though you could also implement it this way. Array.sort(T[] elements, Comparator<T>) does not require T to implement Comparable.

What's left to know about Generics?

In the next few videos, I want to cover the following topics.

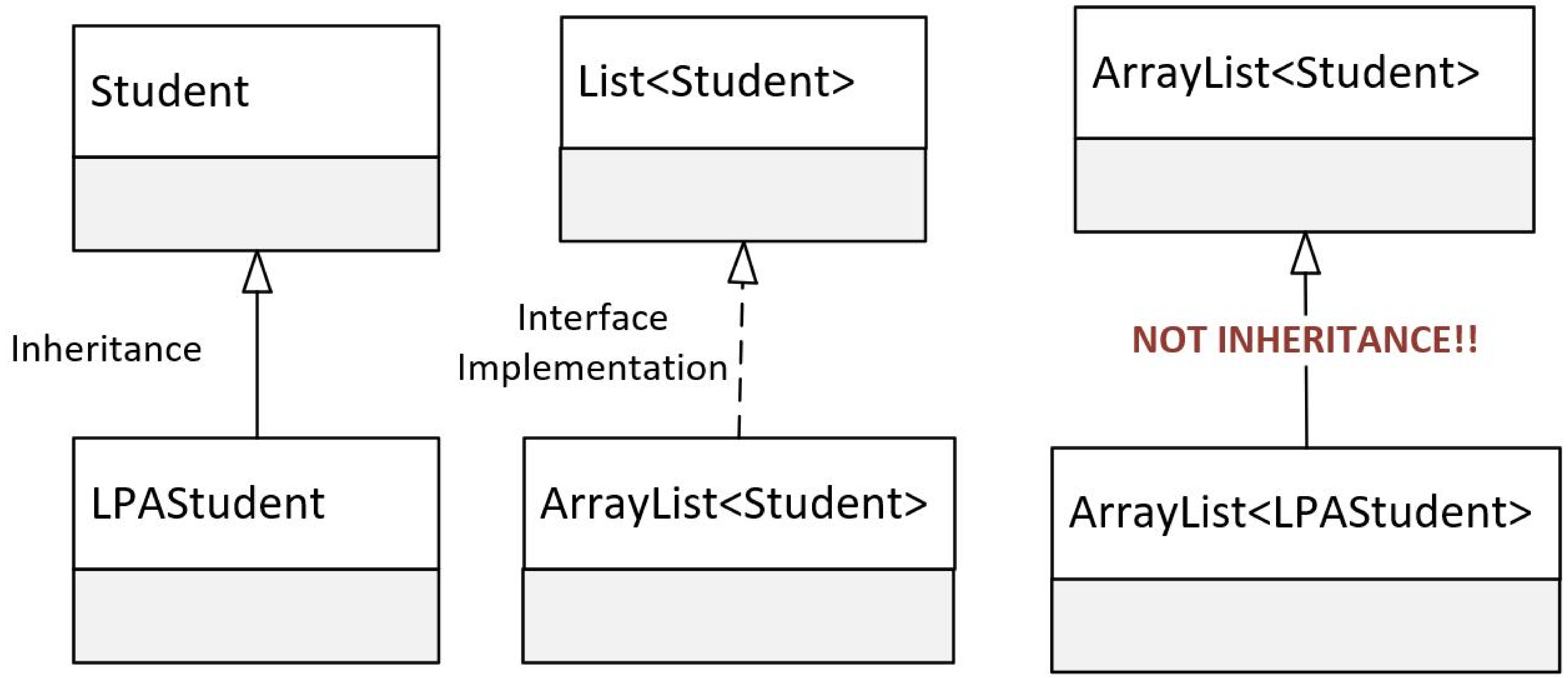
- Using generic references that use type arguments, declared in method parameters and local variables.
- Creating generic methods, apart from generic classes.
- Using wildcards in the type argument.
- Understanding static methods with generic types.
- Using multiple upper bounds.

This isn't inheritance

We know LPAStudent inherits from Student, and we can pass an instance of LPAStudent to any method, or assign it to any reference type, declared with the type of Student.

We also know that ArrayList implements List, and we can pass an ArrayList to a method or assign it to a reference of the List type, and we saw this in both cases for our Student ArrayList.

But why can't we pass an ArrayList of LPA Student, to the method parameter that's declared as a List of Student?



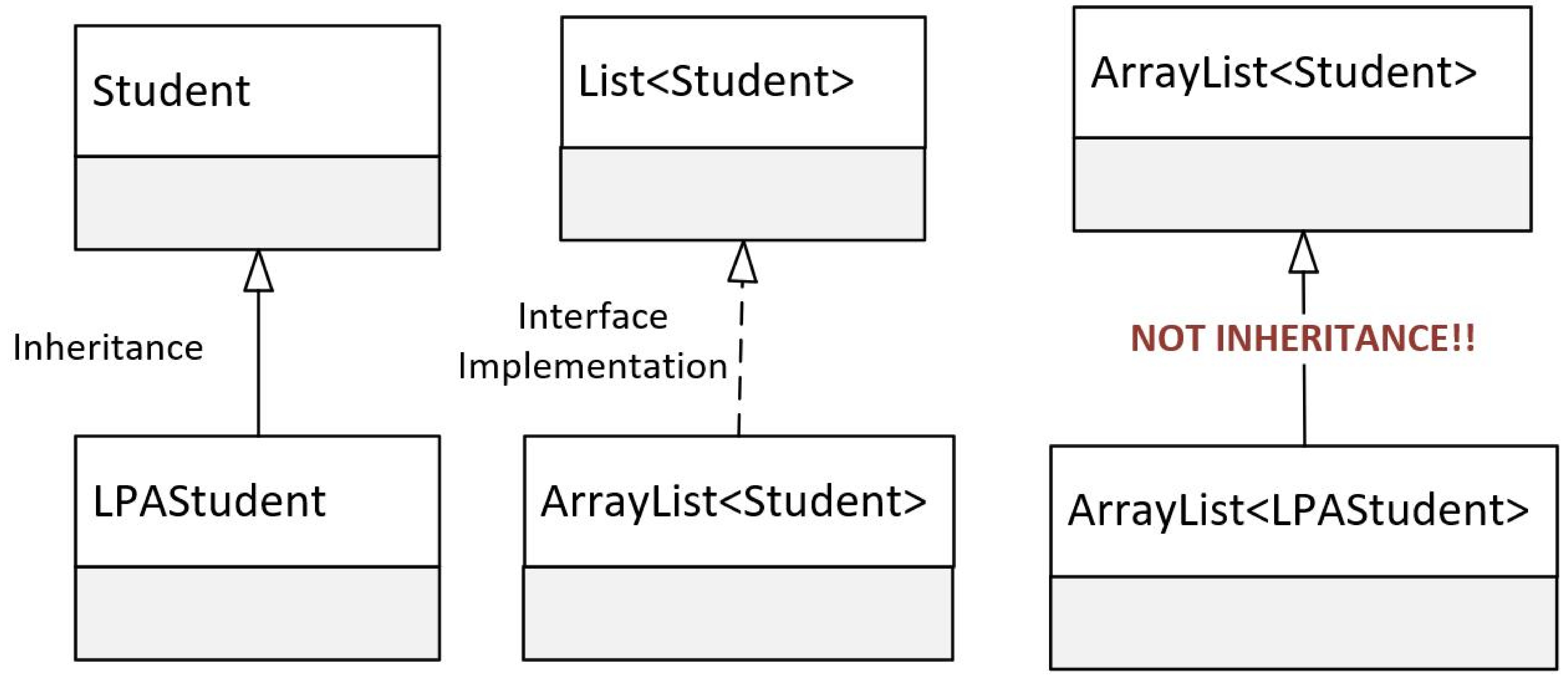
This isn't inheritance

Surely, if an LPAStudent is a Student, a List of LPAStudent is ultimately a List of Student.

It's very natural to assume that a method that takes a List with Students should accept a List with LPAStudents, because LPAStudent is a Student after all.

But that's not how it works.

When used as reference types, a container of one type has no relationship to the same container of another type, even if the contained types do have a relationship.



Limitation of a reference of generic class with a list argument

When you declare a variable or method parameter with:

- List<**Student**>

Only List subtypes with Student elements can be assigned to this variable or method argument.

You can't assign a list of Student subtypes to this!

The generic method

For a method, type parameters are placed after any modifiers and before the method's return type.

The type parameter can be referenced in method parameters, as the method return type, and in the method code block, much as we have seen how a class's type parameter be used.

A generic method can be used for collections with type arguments, as we just saw, to allow for variability of the elements in the collection, without using a raw version of the collection.

```
public <T> String myMethod(T input) {
    return input.toString();
}
```

The generic method

A generic method can be used for static methods on a generic class, because static methods can't use class type parameters.

A generic method can be used on a non-generic class, to enforce type rules on a specific method.

The generic method type parameter is separate from a generic class type parameter.

In fact, if you've used T for both, the T declared on the method means a different type, than the T for the class.

```java
public <T> String myMethod(T input) {
    return input.toString();
}
```

Type Parameters, Type Arguments and using a Wildcard

A **type parameter** is a generic class, or generic method's declaration of the type.

In both of these examples, T is said to be the type parameter.

You can bind a type parameter with the use of the **extends** keyword, to specify an **upper bound**.

Generic class	Generic Method
`public class Team<T> {}`	`public <T> void doSomething(T t) {}`

Type Parameters, Type Arguments and using a Wildcard

A **type argument** declares the type to be used, and is specified in a type reference, such as a local variable reference, method parameter declaration, or field declaration.

In this example, BaseballPlayer is the type argument for the Team class.

Generic class

```
Team<BaseballPlayer> team = new Team<>();
```

Type Parameters, Type Arguments and using a Wildcard

A **wildcard** can only be used in a **type argument**, not in the type parameter declaration.

A wildcard is represented by the **?** character.

A wildcard means the type is **unknown**.

For this reason, a wildcard **limits what you can do**, when you specify a type this way.

List declaration using a wildcard

```
List<?> unknownList;
```

Type Parameters, Type Arguments and using a Wildcard

A wild card can't be used in an instantiation of a generic class.

The code shown here is invalid.

Invalid! You can’t use a wildcard in an instantiation expression

```
var myList = new ArrayList<?>();
```

Type Parameters, Type Arguments and using a Wildcard

A wildcard can be unbounded, or alternately specify either an upper bound or lower bound.

You **can't specify both** an **upper** bound and a **lower** bound, in the same declaration.

Argument	Example	Description
unbounded	`List<?>`	A List of any type can be passed or assigned to a List using this wildcard.
upper bound	`List<? extends Student>`	A list containing any type that is a Student or a **sub type** of Student can be assigned or passed to an argument specifying this wildcard..
lower bound	`List<? super LPAStudent>`	A list containing any type that is an LPAStudent or a **super type** of LPAStudent, so in our case, that would be Student AND Object.

Type Erasure

Generics exist to enforce tighter type checks at compile time.

The compiler transforms a generic class into a typed class, meaning the byte code or class file contains no type parameters.

Everywhere a type parameter is used in a class, it gets replaced with either the type Object, if no upper bound was specified, or the upper bound type itself.

This transformation process is called type erasure, because the T parameter (or S, U, V), is erased or replaced with a true type.

Why is this important?

Understanding how type erasure works for overloaded methods is important.

Using Multiple types to declare an Upper Bound

You can use multiple types to set a more restrictive upper bound with the use of an ampersand between types.

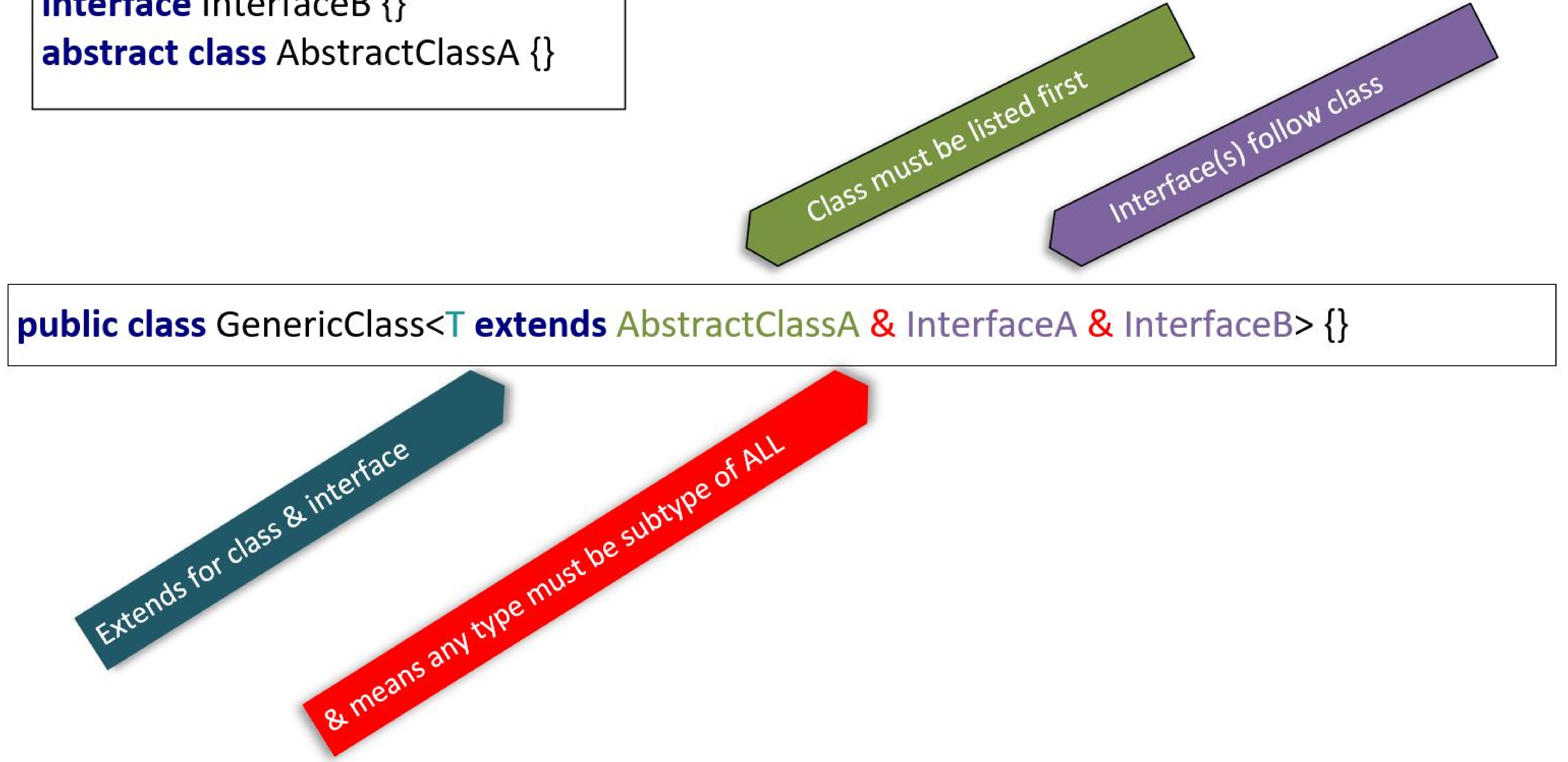
The conditions require a type argument to implement all interfaces declared, and to be a subtype of any class specified.

You can extend only one class at most, and zero to many interfaces.

You use extends for either a class or an interface or both.

If you do extend a class as well as an interface or two, the class must be the first type listed.

```java
interface InterfaceA {}
interface InterfaceB {}
abstract class AbstractClassA {}
```


Class must be listed first
Interface(s) follow class
Extends for class & interface
& means any type must be subtype of ALL

```java
public class GenericClass<T extends AbstractClassA & InterfaceA & InterfaceB> {}
```

Putting it all together, Final Section Challenge

In this challenge, I want you to start with some of the code I just talked about in the last video.

Be sure to start with the Student and LPAStudent classes, as well as the QueryItem interface and QueryList class.

Putting it all together, Final Section Challenge

In this challenge, you'll want to do the following items:

- Change QueryList to extend ArrayList, removing the items field.
- Add a student id field to the Student class, and implement a way to compare Students, so that students are naturally ordered by a student id.
- Implement at least one other mechanism for comparing Students by course or year, or for LPAStudents, by percent complete.

Putting it all together, Final Section Challenge

In this challenge, you'll want to do the following items:

- Override the matchFieldValue method on the LPAStudent class, so that you return students not matched on percent complete equal to a value, but on percent less than or equal to a submitted value. Note: an LPAStudent should be searchable by the same fields as Student as well.
- Run your code for 25 random students selecting students who have completed less than or equal to 50% of their course, and print out the list, sorted in at least two ways. First by using List.sort with the Comparator.naturalOrder() comparator, and then using your own Comparator, so first by student id, as well as one of the other ways you selected.
- And be sure to have some fun!