NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 9: 26th Mar 2024

Generics

- By now, you're familiar with generalizing, conceptually, your own class design.
- You pull out what things your classes have in common, so that you can think about them generally.
- Generics allow us to create classes, to design them, in a general way, without really worrying about the specific details of the elements they might contain.
- Java's ArrayLists are an example of a generic class.
- We can use an ArrayList for any type of object, because many of the methods on that class can be applied to any type.
- It is important to see how to create generic types, and examine how to use them and when you might choose to use them.

What are Generics?

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

It also supports generic methods.

Declaring a Class vs. Declaring a generic Class

Let's look at 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 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

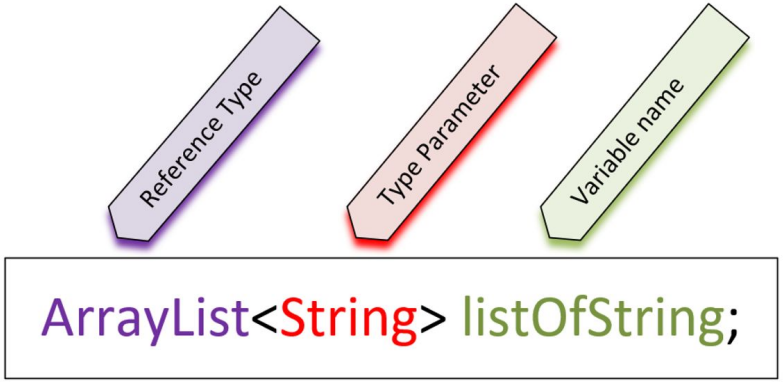
`class ITellYou {`
 	
`    private String field;`
 	
`}`
	`class YouTellMe<T> {`
	
	`    private T field;`
	
	`}`

Using a generic class as a reference type

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

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

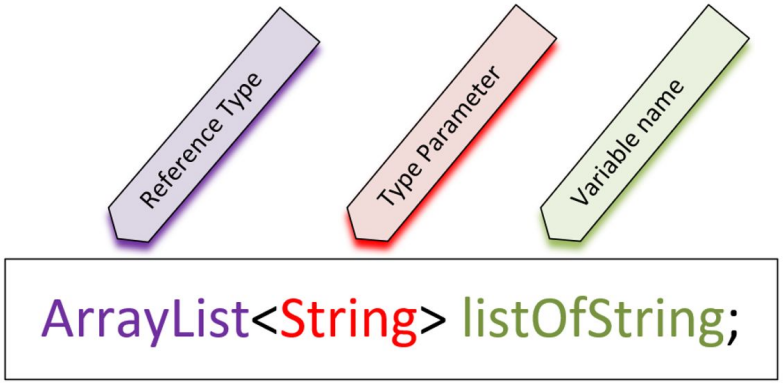
The reference type is ArrayList, the type parameter (or parameterized type) is String, which is declared in angle brackets, and listOfString is the variable name.



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.

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

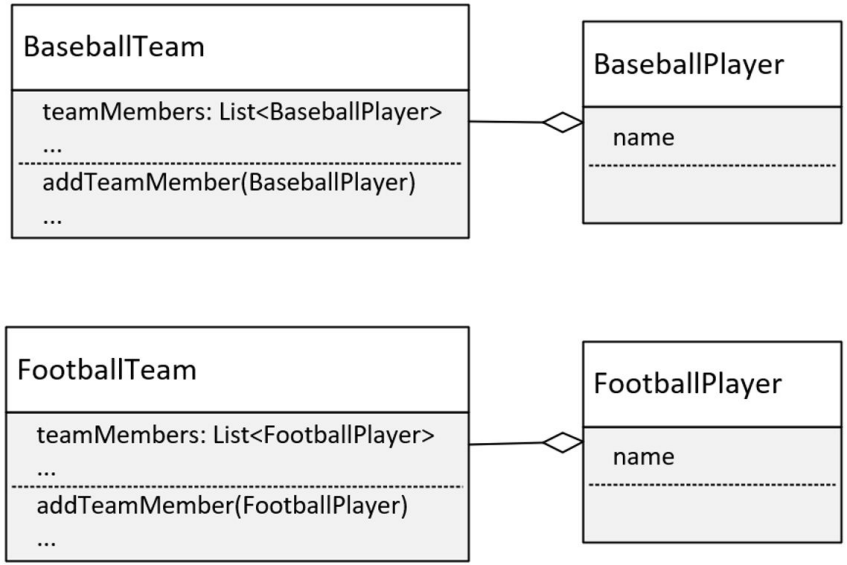


Solution 1: Duplicate code

We could copy and paste the BaseballTeam, and rename everything for FootballTeam, and create a FootballPlayer.

This means you'd have to make sure any changes you made 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.



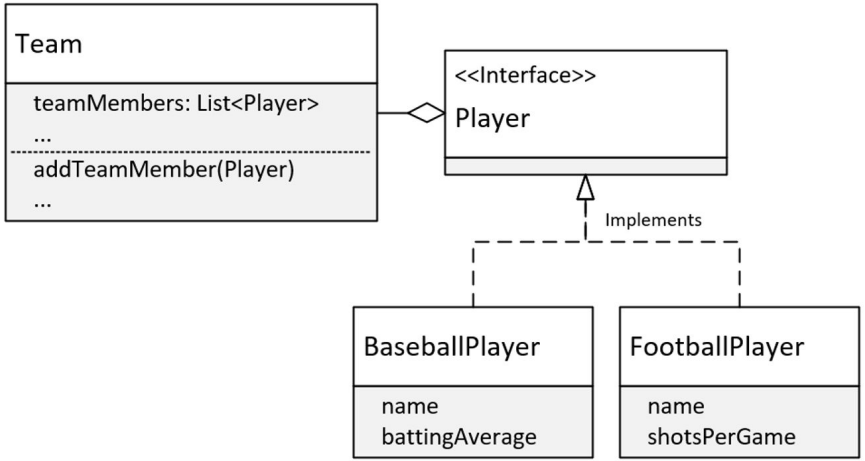
Solution 2: Use a Player Interface or abstract class to support different types of players

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

Here,the Team Class has the members are a List of Players.

Player is an interface, and have BaseballPlayer and FootballPlayer implementing that interface.

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



Generic Type Parameters

This is the one way to declare a generic class, that is to include a type parameter in the angle brackets.

```
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 let me encourage you to stick to this convention.

Generic Type Parameters

You can have more than one type parameter, so we could do T1, T2, T3.

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

But again 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.

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

```
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.

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.

Now 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.

Interfaces used for sorting

Since interfaces and generic classes are covered, let us look at one of the common interfaces used in detailed, i.e., Comparable.

For an array, we can simply call **Arrays.sort**, and pass it an array, but 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 implement this interface.

Comparable Interface

The interface declaration in 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 mean, when implemented.

resulting Value	Meaning
zero	`0 == 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.

You'll 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.

We'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> {`
 	
`    int compare(T o1, T o2);`
`}`
	`public interface Comparable<T> {`
	
	`    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.

Nesting classes (or types) within another class (or type)

A class can contain other types within the class body, such as other classes, interfaces, enums, and records.

These are called nested types, or nested classes.

You might want to use nested classes when your classes are tightly coupled, meaning their functionality is interwoven.

Nested Classes

The four different types of nested classes you can use in Java are: the static nested class, the inner class, and the local and anonymous classes.

Type	Description
static nested class	declared in class body. Much like a static field, access to this class is through the Class name identifier
instance or inner class	declared in class body. This type of class can only be accessed through an instance of the outer class.
local class	declared within a method body.
anonymous class	unnamed class, declared and instantiated in same statement.

Important Restrictions for nested classes were removed in JDK16

Before JDK16, only static nested classes were allowed to have static methods.

As of JDK16, all four types of nested classes can have static members of any type, including static methods.

Static Nested Class

The static nested class is a class enclosed in the structure of another class, declared as static.

This means the class, if accessed externally, requires the outer class name as part of the qualifying name.

This class has the advantage of being able to access private attributes on the outer class.

The enclosing class can access any attributes on the static nested class, also including private attributes.

Inner Classes

Inner classes are non-static classes, declared on an enclosing class, at the member level.

Inner classes can have any of the four valid access modifiers.

An inner class has access to instance members, including private members, of the enclosing class.

Instantiating an inner class from external code, is a bit tricky, and I'll cover that shortly.

As of JDK16, static members of all types are supported on inner classes.

Inner Classes

To create an instance of an inner class, you first must have an instance of the Enclosing Class.

From that instance you call .new, followed by the inner class name and the parentheses, taking any constructor arguments.

This definitely looks strange the first time you see it.

```
EnclosingClass outerClass = new EnclosingClass();
EnclosingClass.InnerClass innerClass = outerClass.new InnerClass();
```

Local Classes

Local classes are inner classes, but declared directly in a code block, usually a method body.

Because of that, **they don't have access modifiers,** and are only accessible to that method body while it's executing.

Like an inner class, they have access to all fields and methods on the enclosing class.

They can also access local variables and method arguments, that are final or effectively final.

Local Class's 'Captured Variables'

When you create an instance of a local class, referenced variables used in the class, from the enclosing code, are 'captured'.

This means a copy is made of them, and the copy is stored with the instance.

This is done because the instance is stored in a different memory area, than the local variables in the method.

For this reason, if a local class uses local variables, or method arguments, from the enclosing code, these must be final or effectively final.

Final Variables and Effectively Final

The code sample on this slide shows:

A method parameter, called methodArgument in the doThis method, declared as final.

And a local variable, in the method block, Field30, also declared with the key word final.

In both these cases, this means you can't assign a different value, once these are initialized.

```
class ShowFinal {

    private void doThis(final int methodArgument) {

        final int Field30 = 30;
    }
}
```

These are explicitly final, and any of these could be used in a local class, because of this.

Effectively Final

In addition to explicitly final variables, you can also use **effectively final** variables in your local class.

A local variable or a method argument are effectively final, if a value is assigned to them, and then never changed after that.

Effectively final variables can be used in a local class.

Additional Local Types

As of JDK 16, you can also create a local record, interface and enum type, in your method block.

These are all implicitly static types, and therefore aren't inner classes, or types, but static nested types.

The record was introduced in JDK16.

Prior to that release, there was no support for a local interface or enum in a method block either.

Anonymous Classes

An anonymous class is a local class that doesn't have a name.

All the nested classes we've looked at so far have been created with a class declaration.

The anonymous class is never created with a class declaration, but it's always instantiated as part of an expression.

Anonymous classes are used a lot less, since the introduction of Lambda Expressions in JDK 8.

But there are still some use cases where an anonymous class might be a good solution.

Anonymous class creation

An anonymous class is instantiated and assigned in a single statement.

The new keyword is used followed by any type.

This is NOT the type of the class being instantiated.

It's the super class of the anonymous class, or it's the interface this anonymous class will implement.

```
var c4 = new Comparator<StoreEmployee>() {};
```

Anonymous class creation

In the first example on this slide, the anonymous unnamed class will implement the Comparator interface.

```
var c4 = new Comparator<StoreEmployee>() {};
```

In the second example on this slide, the anonymous class extends the Employee class, meaning it's a subclass of Employee.

```
var e1 = new Employee {};
```

In both cases, it's important to remember the semi-colon after the closing bracket, because this is an expression, not a declaration.

The Local and Anonymous Class Challenge

First, you need to create a record named Employee, that contains First Name, Last Name, and hire date.

Set up a list of Employees with various names and hire dates in the main method.

Set up a new method that takes this list of Employees as a parameter.

Create a local class to wrap this class, (pass Employee to the constructor and include a field for this) and add some calculated fields, such as full name, and years worked.

The Local and Anonymous Class Challenge

Create a list of employees using your local class.

Create an anonymous class to sort your local class employees, by full name, or years worked.

Print the sorted list.

Hint: Here is another review of a date function, which should help you with calculating years worked.

```
int currentYear = LocalDate.now().getYear();
```