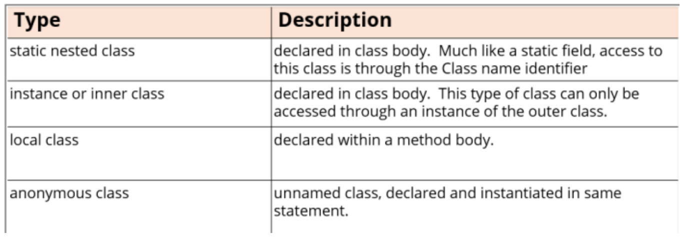
Nested Classes:

- A class usually contains attributes and methods. However, 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.

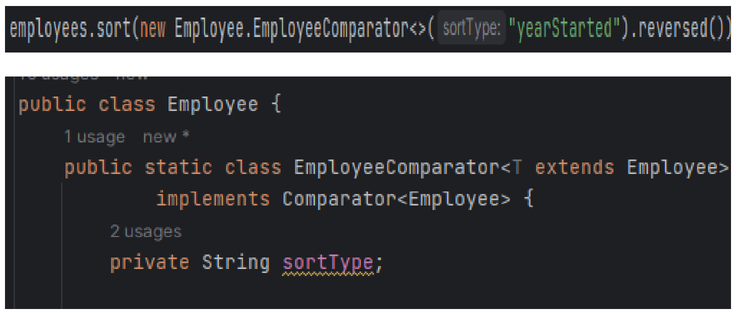
Note : Only static nested classes were allowed to have static methods before JDK 16. Now all four types of nested classes can have static members of any type, including static methods.



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.

Static Nested Classes:

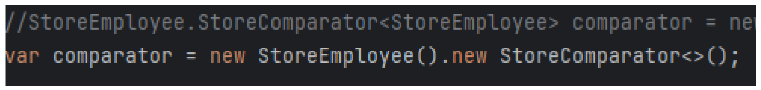
- It is a class enclosed in the structure of another class declaring it as static.
- To access this class externally, it requires the outer class name as a part of qualifying name. Here the static nested class is Employee Comparator. In order to access it, we require the outer class name that is Employee.
- The major advantage of static nested class is being to able to access private attributes of the outer class.
- Similarly, the outer class can access any attributes of the nested static class, including private attributes.
- Starting master classes doesn't have access to the instance variables and methods of their enclosing class.



```
employees.sort(new Employee.EmployeeComparator<>( sortType: "yearStarted").reversed())
```

```
public class Employee {
    1 usage  new *
    public static class EmployeeComparator<T extends Employee>
            implements Comparator<Employee> {
        2 usages
        private String sortType;
```

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



```
//StoreEmployee.StoreComparator<StoreEmployee> comparator = ne
var comparator = new StoreEmployee().new StoreComparator<>();
```

Static nested classes can be used to logically group related functionality together. And also improve encapsulation (Nested static classes can access private members of their enclosing class, providing better encapsulation and access control. This allows the utility methods to interact with private members of the enclosing class, which might not be accessible from outside the class).

Lets take an example

- The PatientRecord class represents patient information such as patientId, name, age, and diagnosis.
- We have a nested static class DataUtils within PatientRecord to encapsulate utility methods for data validation and processing. These methods include isValidAge, isValidDiagnosis, and formatName.
- By grouping these utility methods within a nested static class, we improve the organization and clarity of our code. This is particularly useful because the PatientRecord class may have other non-static members and methods related to patient information, and we want to separate the utility methods from the rest of the class's functionality.

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

- Unlike started nested classes, inner classes have access to the instance variables and methods of the enclosing class.
- To create an instance of an inner class, you first must have an instance of the enclosed class. From that instance you call .new keyword followed by the inner class name and the parenthesis taking any constructor arguments.

Inner classes are useful when you need to represent a relationship where one class is part of another class or when you want to encapsulate related functionality within the scope of an instance of the enclosing class.

nested static classes and inner classes are useful when you need to define classes that are closely related to an existing class and benefit from access to its members or when you want to logically group related classes together within a single class file.

Local Classes: classes are also inner classes, but they are directly declared in the code block, usually a method body.

- Since they are declared in a method body, they don't have access modifiers. They're only accessible to that method body while it's executing.
- However, like an inner class, local classes also have access to all the fields and methods on their enclosing class. Additionally they can also access local variables and method arguments that are final or effectively final

Final vs effectively final: Once you declare a variable final, you cannot assign a different value once these are initialized. These are explicitly declared final variables. If you assign a value to them and never change after that. These are called as effectively final variables.

Why do we need to have local classes?

If the functionality provided by the class is specific to certain methods and is not relevant outside of that context. We can use a local class to keep the implementation details close to where they are used. Here we are limiting the visibility to only the code within a particular method. This will help us to prevent unintended usage of a class elsewhere in the code base.

Local classes are meant to be localized implementation specific to particular method or context, whereas interfaces are designed to be global contracts that can be implemented by various classes. The reason you cannot define an interface inside a method (local interface) is because interfaces are meant to be contracts that can be implemented by multiple classes. Placing an interface inside a method would restrict its visibility and make it inaccessible to other parts of code, defeating the purpose of interfaces. Whereas local classes, including local interfaces, can be useful for encapsulating functionality within a specific method scope. They are tightly coupled with the method in which they are defined and are not meant to be accessed from outside that method. Local classes can access variables and parameters from the enclosing method, which can be useful for certain scenarios where you need to encapsulate logic within a specific context.

Anonymous Class: Anonymous Class is a local class that doesn't have a name. It is never created with a class declaration, but always instantiated as a part of expression. Anonymous class are replaced by lambda expression starting JDK 8.

Anonymous classes can extend the class or implement the interface. They're often used for one time implementation of interfaces or abstract classes.

Usually anonymous class is created inline without being assigned to a variable, and its purpose is to provide the implementation of the abstract method of an interface

- We can create anonymous class either by extending a class or by implementing an interface. We only use it in cases where we only need a Single object that implements interface, we can do it in one single statement and get the object we need for one time use.