Inner classes without name are called anonymous inner classes.

Without Lambda

**public** **static** **void** main(String[] args) {

Runnable r = **new** Runnable() {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In myThread" + i);

}

}

};

Thread t = **new** Thread(r);

t.start();

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In main" + i);

}

}

In above the implementation class has no name.

With Lambda

**public** **static** **void** main(String[] args) {

Runnable r = () -> {

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In myThread" + i);

}

};

Thread t = **new** Thread(r);

t.start();

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In main" + i);

}

}

Directly pass Runnable to thread

**public** **static** **void** main(String[] args) {

Thread t = **new** Thread(() -> {

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In myThread" + i);

}

});

t.start();

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In main" + i);

}

}

We cannot use lambda expression for every anonymous inner class.

Thread t = **new** Thread(() -> {

**for** (**int** i = 0; i < 100; i++) {

System.***out***.println("In myThread" + i);

}

});

In above method, the anonymous inner class (Runnable) has only one abstract method (run), so we can use it as lambda expression. What if an anonymous inner class has more than one abstract class?

Inner classes that contain only one abstract method

|  |  |  |  |
| --- | --- | --- | --- |
| **class** Test {    } | **abstract** **class** Test {    } | **interface** Test {  **public** **void** m1();  **public** **void** m2();  **public** **void** m3();  } | **interface** Test {  **public** **void** m1();  } |
| Test t =**new** Test() {    }; | Test t =**new** Test() {    }; | Test t =**new** Test() {  **public** **void** m1() {}  **public** **void** m2() {}  **public** **void** m3() {}    }; | Test t =**new** Test() {  **public** **void** m1() {}    }; |
| anonymous inner class that implements concrete class | anonymous inner class that implements abstract class | anonymous inner class that implements an interface with more than one abstract methods | anonymous inner class that implements an interface with more only one abstract methods |
| anonymous inner class can extend concrete class, but lambda expression cannot extent concrete class | anonymous inner class can extend abstract class, but lambda expression cannot extent abstract class | anonymous inner class can contain multiple methods, but lambda expression cannot use those interfaces | This is the only way we can replace anonymous inner class with lambda expression |

Another difference with this variable

|  |  |
| --- | --- |
|  |  |
| **public** **class** ThisNoLambda {  String var = "MainClass";  **public** **static** **void** main(String[] args) {  ThisNoLambda t = **new** ThisNoLambda();  t.m2();  }  **public** **void** m2() {  MyInterface myInterface = **new** MyInterface() {  String var = "anonymous inner class";  **public** **void** m1() {  System.***out***.println(**this**.var);  System.***out***.println(var);  }  };  myInterface.m1();  }  }  **interface** MyInterface {  **public** **void** m1();  } | **public** **class** ThisLambda {  String var = "MainClass";  **public** **static** **void** main(String[] args) {  ThisLambda t = **new** ThisLambda();  t.m2();  }  **public** **void** m2() {  MyInterface2 myInterface = () -> {  String var = "anonymous inner class";  System.***out***.println(**this**.var);  System.***out***.println(var);  };  myInterface.m1();  }  }  **interface** MyInterface2 {  **public** **void** m1();  } |
| Output:  anonymous inner class  anonymous inner class | Output:  MainClass  anonymous inner class |
| We can declare instance variable.  Inside anonymous inner class, this refers to current inner class | We cannot declare instance variables in lambda expressions. They act as local variables only (so this. Refers to main class variable and not lambda variable [see var in above example])  Its not possible to declare instance variable inside lambda expression |
| This refers to inner class | This refers to outer |

|  |  |  |
| --- | --- | --- |
|  | Anonymous inner class | Lambda |
| 1 | It’s a class without name | It’s a function without name |
| 2 | It can extend abstract and concrete class | It can’t extend abstract and concrete class |
| 3 | Can implement an interface that contains one or more abstract methods | It can implement interface with single abstract method (Functional interface) |
| 4 | We can declare instance variable inside anonymous inner class | We cannot define instance variables; they are just local variables |
| 5 | Can be instantiated | Cannot be instantiated |
| 6 | this refers to current anonymous inner class object but not outer class object | Inside lambda lambda expression, this refers current outer class object, i.e enclosing class object |
| 7 | anonymous inner class is best choice if we want to handle interface with multiple abstract method | Lambda expression is best choice if we want to handle interface with single abstract method (Functional interface) |
| 8 | For anonymous inner class, at the time of compilation a separate .class file will be generated | For the lambda expression at the time of compilation no separate .class file will be generated |
| 9 | Memory will be allocated on demand whenever we are creating an object | Lambda expression will reside in permanent memory of JVM (method area) |

**public** **class** LambdaTest {

**public** **static** **void** main(String[] args) {

LambdaTest l = **new** LambdaTest();

l.m2();

}

**public** **void** m2() {

**int** i = 100;

MyInterface3 mi = () -> {

**int** j = 20;

System.***out***.println(i);

System.***out***.println(j);

// i=60; // Compile time exception Local variable i defined in an enclosing scope must be final or effectively final

/\*

The local variable which is referenced from lambda expression is effectively final (even you don't mark it as final)

It is implicitly final

\*\*/

};

mi.m1();

}

}

**interface** MyInterface3 {

**public** **void** m1();

}

Advantages of lambda expressions

Lambda expressions makes programmers life easy (see below)

1. We can enable functional programming in Java
2. We can reduce length of code so that readability will be improved
3. We can resolve complexities of anonymous inner classes until some context (single abstract method)
4. We can handle procedures/functions just like values

**interface** Calculator {

**public** **int** add(**int** i, **int** j);

}

Calculator d = (a, b) -> {

**return** a + b;

};

System.***out***.println(d.add(12, 34));

1. We can handle procedures/functions as arguments

Thread t = **new** Thread(r1 = () -> {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread");

}

});

1. Easier to use updated API’s (Stream API) and libraries
2. Enable support for parallel programming (takes advantages of multicore processors)