Day_14_OOPJ_Sanket_Shalukar

Friday, September 12, 2025 10:14 AM

Topics that are in the day 14

Generics

Reflection

Inner classes

Multithreading

Generics:

- **Definition**

  Generics allow classes, interfaces, and methods to be written with type parameters so the same code can work with different data types safely.

- **Need**

  They provide type safety, remove explicit type casting, allow code reusability, and enable compile-time error checking.

- **Type Parameters**

  Commonly used symbols include T (Type), E (Element), K (Key), V (Value).

- **Generic Classes & Methods**

  A single class or method can be written once and used for different data types.

- **Collections & Generics**

  Java Collections Framework uses generics (e.g., `List`, `Set`, `Map`) to enforce type safety and avoid runtime errors.

- **Advantages**

  Generics improve type safety, code reusability, readability, maintainability, and prevent `ClassCastException`.

Synatax of Generics:

Class Classname <T> {
}

Parameters

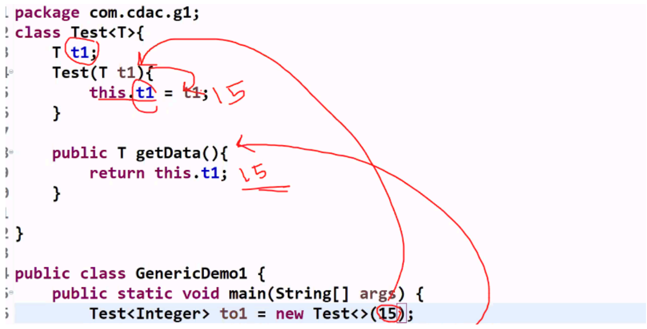
- In **Generics**, we use **type parameters** inside < >.
- These are **just names/letters** (not keywords).
- They act as **placeholders** for actual data types.

- **Example:**
  - <T> → means "some Type" (like `Integer`, `String`, etc.).
  - <E> → means "Element" (commonly used in collections like `List<E>`).
  - `<K, V>` → means "Key and Value" (commonly used in `Map<K,V>`).

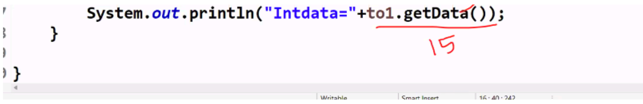
Common Naming Conventions

- T → Type
- E → Element
- K → Key
- V → Value
- N → Number
- S, U, V → second, third, fourth type parameters

Code Examples: For Generics



```
package com.cdac.g1;
class Test<T>{
    T t1;
    Test(T t1){
        this.t1 = t1;
    }

    public T getData(){
        return this.t1;
    }

}

public class GenericDemo1 {
    public static void main(String[] args) {
        Test<Integer> to1 = new Test<>(15);
```



```
        System.out.println("Intdata="+to1.getData());
    }
}
}
```

Reflection :

- **Definition**

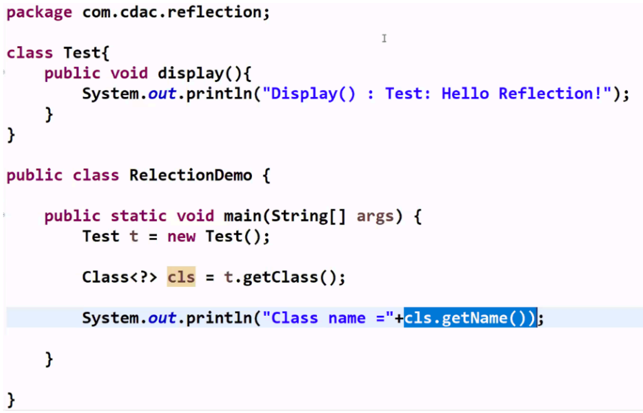
  Reflection is a feature that allows program to inspect and modify classes, methods fields and constructors at runtime, without knowing their names at compile time.

- **Package :**

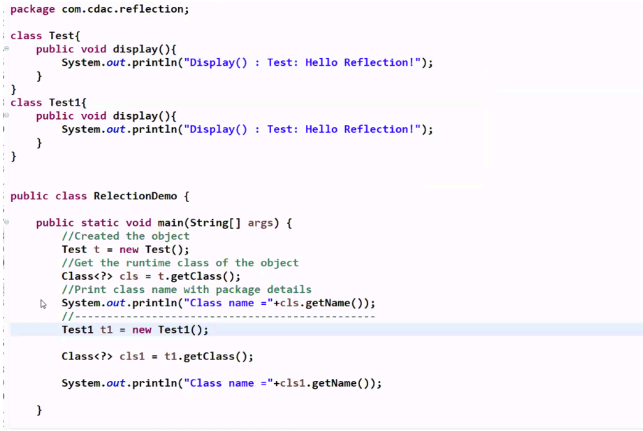
  It is inside the lang package. We need to use "**java.lang.reflect"** to use reflection

- **Key Classes/Interfaces**

  Class : Represents a class at runtime.
  Method: Represents a method of a class.
  Fields : Represents a fields(variables) of a class.
  Constructor: Represents a constructor of a class.

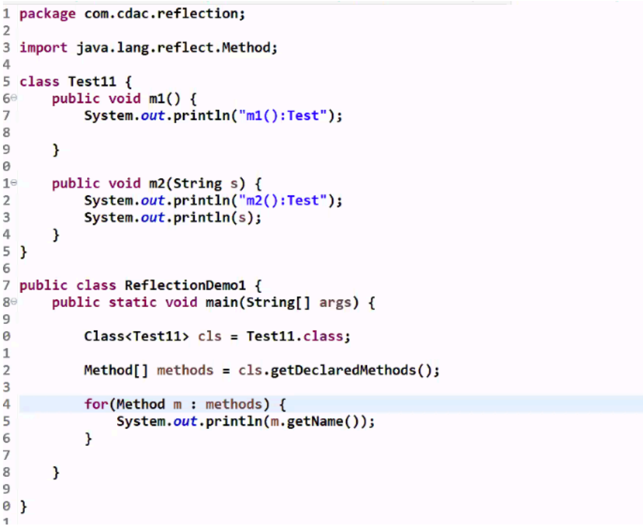


```
package com.cdac.reflection;

class Test{
    public void display(){
        System.out.println("Display() : Test: Hello Reflection!");
    }
}

public class RelectionDemo {

    public static void main(String[] args) {
        Test t = new Test();

        Class<?> cls = t.getClass();

        System.out.println("Class name ="+cls.getName());

    }

}
```



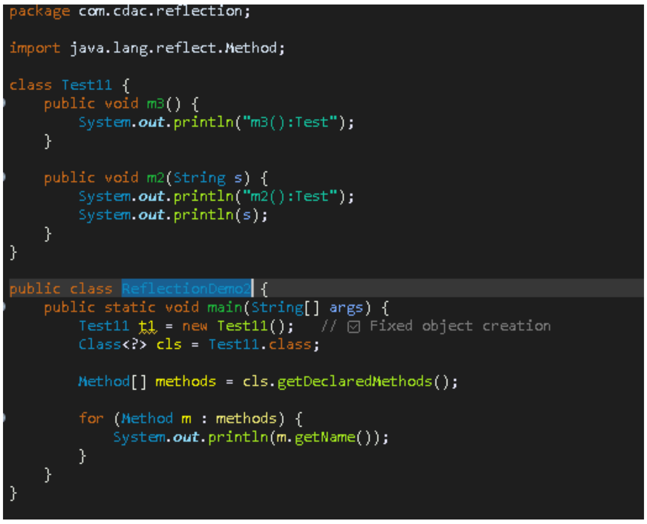
```
package com.cdac.reflection;

class Test{
    public void display(){
        System.out.println("Display() : Test: Hello Reflection!");
    }
}
class Test1{
    public void display(){
        System.out.println("Display() : Test: Hello Reflection!");
    }
}


public class RelectionDemo {

    public static void main(String[] args) {
        //Created the object
        Test t = new Test();
        //Get the runtime class of the object
        Class<?> cls = t.getClass();
        //Print class name with package details
        System.out.println("Class name ="+cls.getName());
        //---------------------------------------------
        Test1 t1 = new Test1();

        Class<?> cls1 = t1.getClass();

        System.out.println("Class name ="+cls1.getName());

    }
```

Second Example:



```
package com.cdac.reflection;

import java.lang.reflect.Method;

class Test11 {
    public void m1() {
        System.out.println("m1():Test");

    }

    public void m2(String s) {
        System.out.println("m2():Test");
        System.out.println(s);
    }
}

public class ReflectionDemo1 {
    public static void main(String[] args) {

        Class<Test11> cls = Test11.class;

        Method[] methods = cls.getDeclaredMethods();

        for(Method m : methods) {
            System.out.println(m.getName());
        }

    }

}
```

Example 3



```java
package com.cdac.reflection;

import java.lang.reflect.Method;

class Test11 {
    public void m3() {
        System.out.println("m3():Test");
    }

    public void m2(String s) {
        System.out.println("m2():Test");
        System.out.println(s);
    }
}

public class ReflectionDemo2 {
    public static void main(String[] args) {
        Test11 t1 = new Test11();   // ☑ Fixed object creation
        Class<?> cls = Test11.class;

        Method[] methods = cls.getDeclaredMethods();

        for (Method m : methods) {
            System.out.println(m.getName());
        }
    }
}
```

Inner classes :

Definition :

An **inner class** in Java is a **class defined within another class**. It is a way to logically group classes that are only used in one place, increasing encapsulation and readability. Inner classes can access the members (including private members) of the outer class directly.

Key Points:

1. Inner classes exist **inside the body of another class**.
2. They can be **associated with an instance of the outer class** or **be static** (not tied to an instance).
3. Inner classes are often used to **implement helper classes, event handlers, or callbacks**.
4. By keeping a class inside another, you can **hide it from other classes**, which improves **encapsulation**.
5. Inner classes have **access to all members** (fields and methods) of the outer class, including private ones.

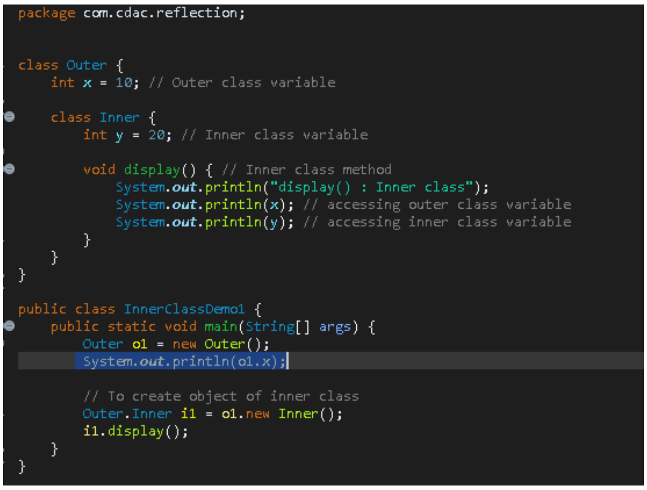
There are different types of inner classes (you don't need code for this, just the concept):

- **Non-static (instance) inner class:** Belongs to an instance of the outer class.
- **Static nested class:** Belongs to the outer class itself, not instances.
- **Local inner class:** Defined inside a method.
- **Anonymous inner class:** Defined without a name, usually for one-time use.

1. Regular class is a class defined inside another class
2. Regular inner class / (Non-static nasted class);
3. Define inside

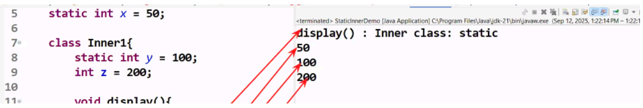
Code Examples: for Inner class

Example no 1



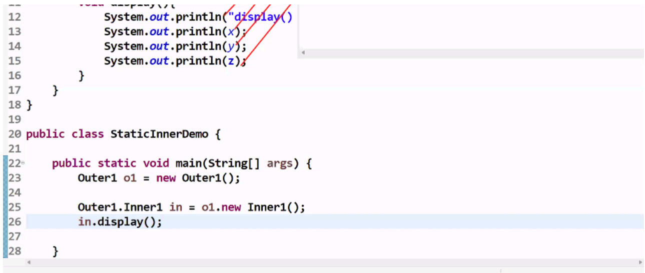
```java
package com.cdac.reflection;


class Outer {
    int x = 10; // Outer class variable

    class Inner {
        int y = 20; // Inner class variable

        void display() { // Inner class method
            System.out.println("display() : Inner class");
            System.out.println(x); // accessing outer class variable
            System.out.println(y); // accessing inner class variable
        }
    }
}

public class InnerClassDemo1 {
    public static void main(String[] args) {
        Outer o1 = new Outer();
        System.out.println(o1.x);

        // To create object of inner class
        Outer.Inner i1 = o1.new Inner();
        i1.display();
    }
}
```

Example 2 - (Static Inner class)



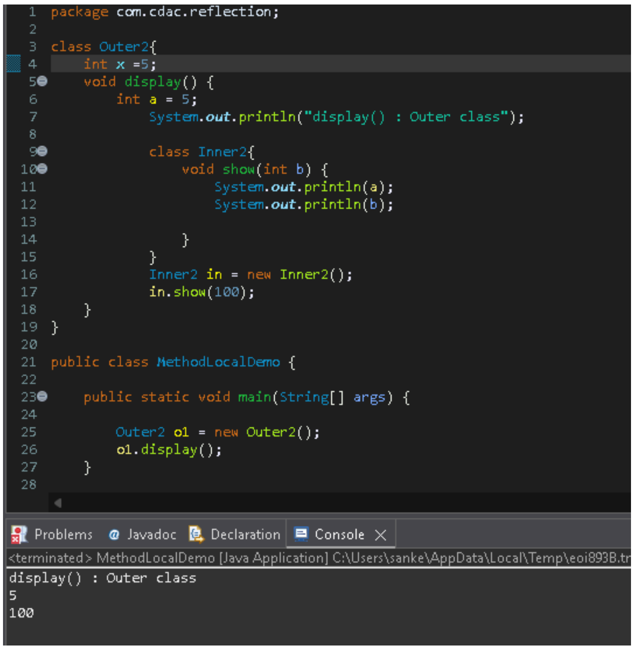
```java
    static int x = 50;

    class Inner1{
        static int y = 100;
        int z = 200;

        void display(){
```

```
<terminated> StaticInnerDemo [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Sep 12, 2025, 1:22:14 PM – 1:22:1
display() : Inner class: static
50
100
200
```



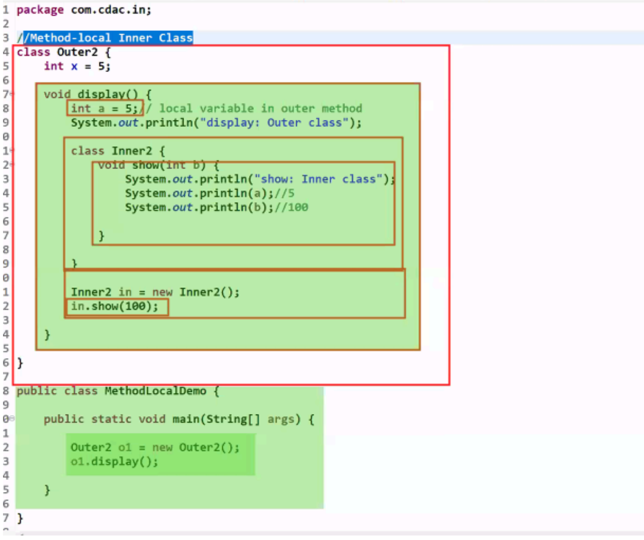
```java
        void display(){
            System.out.println("display()
            System.out.println(x);
            System.out.println(y);
            System.out.println(z);
        }
    }
}

public class StaticInnerDemo {

    public static void main(String[] args) {
        Outer1 o1 = new Outer1();

        Outer1.Inner1 in = o1.new Inner1();
        in.display();

    }
```

Example 2 (Method Local Demo)



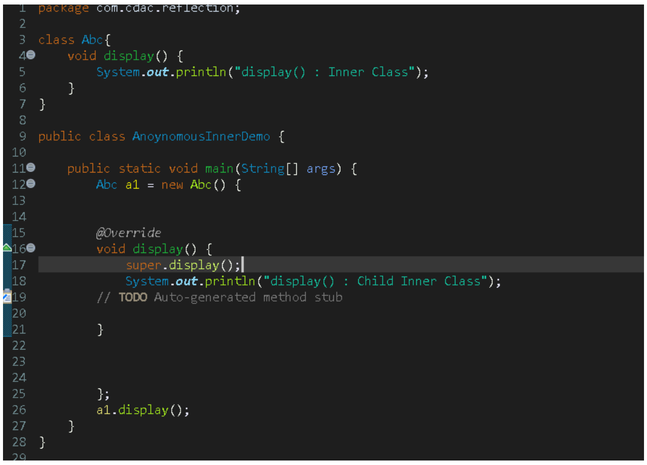
```java
package com.cdac.reflection;

class Outer2{
    int x =5;
    void display() {
        int a = 5;
            System.out.println("display() : Outer class");

            class Inner2{
                void show(int b) {
                    System.out.println(a);
                    System.out.println(b);

                }
            }
            Inner2 in = new Inner2();
            in.show(100);
    }
}

public class MethodLocalDemo {

    public static void main(String[] args) {

        Outer2 o1 = new Outer2();
        o1.display();
    }

```

```
<terminated> MethodLocalDemo [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B.tr
display() : Outer class
5
100
```

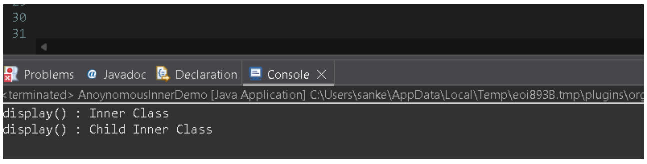


```java
package com.cdac.in;

//Method-local Inner Class
class Outer2 {
    int x = 5;

    void display() {
        int a = 5;// local variable in outer method
        System.out.println("display: Outer class");

        class Inner2 {
            void show(int b) {
                System.out.println("show: Inner class");
                System.out.println(a);//5
                System.out.println(b);//100

            }

        }

        Inner2 in = new Inner2();
        in.show(100);

    }

}

public class MethodLocalDemo {

    public static void main(String[] args) {

        Outer2 o1 = new Outer2();
        o1.display();

    }

}
```

Example 3 (AnoynomousInnerDemo)

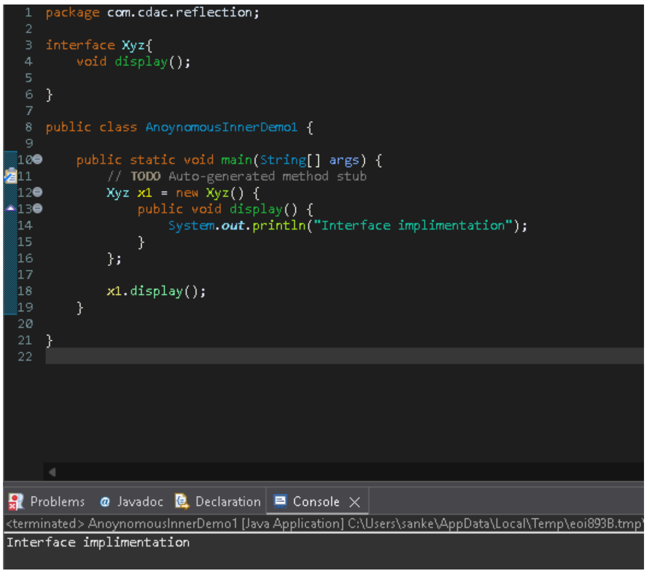


```java
package com.cdac.reflection;

class Abc{
    void display() {
        System.out.println("display() : Inner Class");
    }
}

public class AnoynomousInnerDemo {

    public static void main(String[] args) {
        Abc a1 = new Abc() {


        @Override
        void display() {
            super.display();
            System.out.println("display() : Child Inner Class");
        // TODO Auto-generated method stub

        }



        };
        a1.display();
    }
}
```



```
<terminated> AnoynomousInnerDemo [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B.tmp\plugins\org
display() : Inner Class
display() : Child Inner Class
```

Example 3 (AnoynomousInner1Demo)



```java
package com.cdac.reflection;

interface Xyz{
    void display();

}

public class AnoynomousInnerDemo1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Xyz x1 = new Xyz() {
            public void display() {
                System.out.println("Interface implimentation");
            }
        };

        x1.display();
    }

}
```

```
<terminated> AnoynomousInnerDemo1 [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B.tmp\
Interface implimentation
```

Boilerplate Code :

Boilerplate code means the **repetitive, standard code** you need to write again and again, even if it adds little value.

Examples in Java:

- Writing `getters` and `setters` for every class field.
- Overriding `toString()`, `equals()`, `hashCode()` again and again.
- Writing anonymous inner classes just to pass a simple function (before lambdas).

Why is it bad?

- Makes code long, harder to read.
- Easy to introduce errors when copying/pasting.
- Doesn't add much to business logic.

How Java reduces boilerplate?

- **Lambdas** → reduce anonymous inner class code.
- **Records (Java 14+)** → automatically create constructor, getters, toString, equals, hashCode.
- **Annotations (e.g., Lombok's @Getter, @Setter)** → reduce repeated code.

Lambda Expression :

- **Definition**

  1. A **lambda expression** is an anonymous function (no name, no return type declaration, no modifiers) used to provide the implementation of a functional interface in a concise way.
  2. A **lambda expression** in Java is basically a **short way to write a method** (especially for functional interfaces — interfaces with only **one abstract method**).

     Instead of writing a whole class and method, you can pass a function **as an argument**.

  - It is a concise way to represent anonymous function (methods without names)
  - Java 8 for
  - Used to implement functional interfaces (interface with single abstract method)

- **Uses :**

  Code readability and reduce boilerplate code.

  @Getter
  @Setter
  @constructor

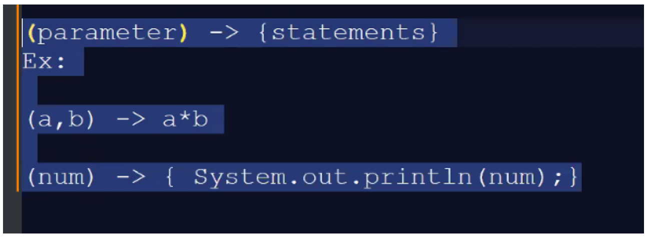
- **Syntax :**

  (parameter) -> expression
  Or
  (parameter) -> {statement}

- **Functional Interface:**
    1. Required for Lambda functions
    2. It contains exactly one abstract method
    3. We can also create custom functional interface @FunctionalInterface
    4. It is a part of java.lang package.

- **Example :**

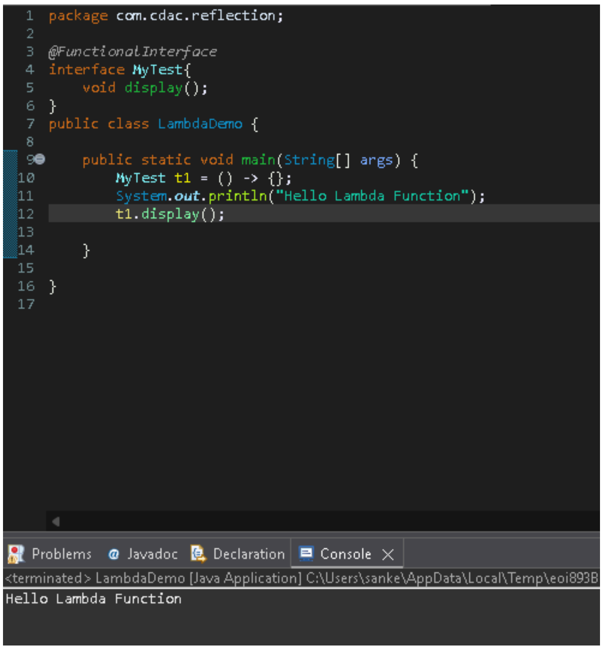
```
@Functional Interface
Interface MyTest{
        Void display();
}
```



```
(parameter) -> {statements}
Ex:

(a,b) -> a*b

(num) -> { System.out.println(num);}
```

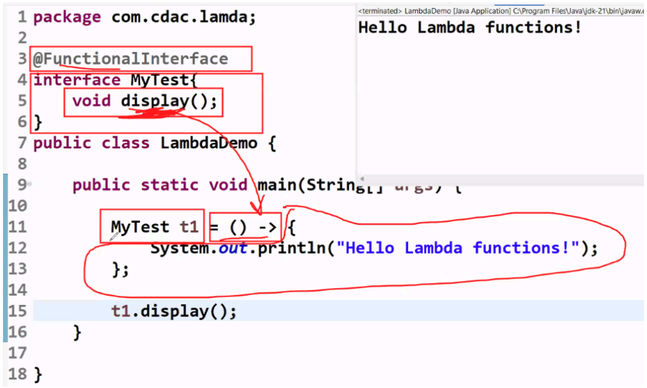
Code Examples: For lambda expression

1) Lambda Function Demo



```
package com.cdac.reflection;

@FunctionalInterface
interface MyTest{
    void display();
}
public class LambdaDemo {

    public static void main(String[] args) {
        MyTest t1 = () -> {};
        System.out.println("Hello Lambda Function");
        t1.display();

    }

}
```

```
<terminated> LambdaDemo [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B
Hello Lambda Function
```

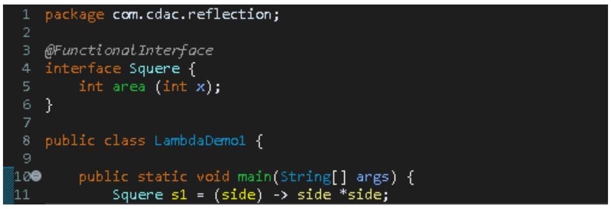


```
package com.cdac.lamda;

@FunctionalInterface
interface MyTest{
    void display();
}
public class LambdaDemo {

    public static void main(String[] args) {

        MyTest t1 = () -> {
            System.out.println("Hello Lambda functions!");
        };

        t1.display();
    }

}
```

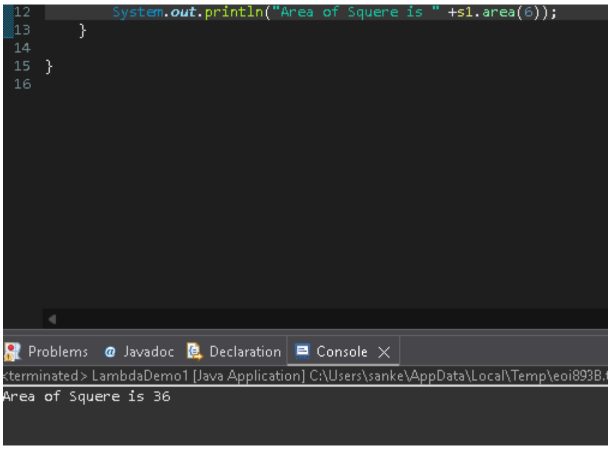
```
<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.e
Hello Lambda functions!
```

Example no 2

LambdaDemo1

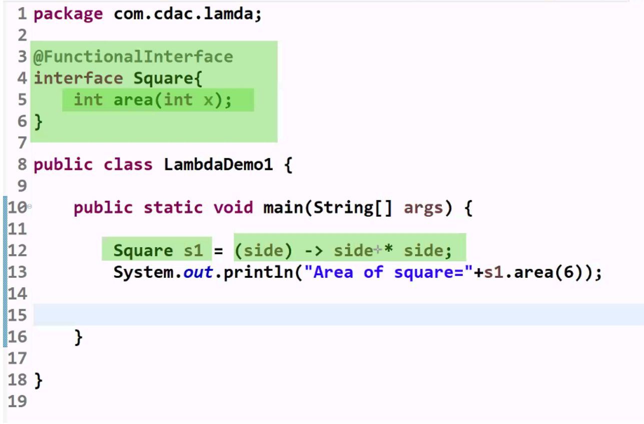


```
package com.cdac.reflection;

@FunctionalInterface
interface Squere {
    int area (int x);
}

public class LambdaDemo1 {

    public static void main(String[] args) {
        Squere s1 = (side) -> side *side;
```



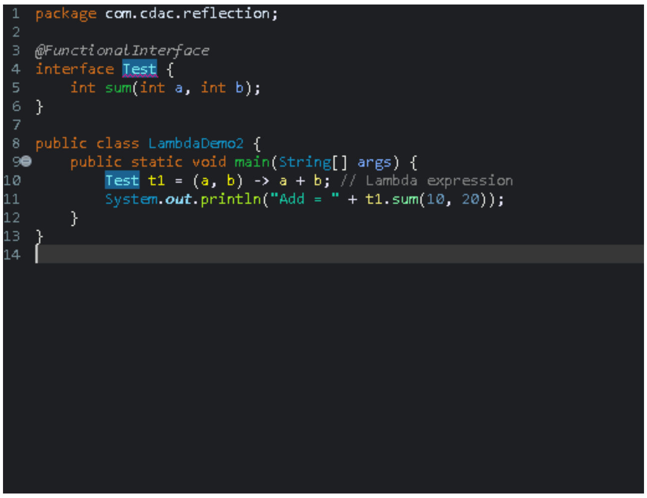
```
            System.out.println("Area of Squere is " +s1.area(6));
    }

}
```

```
<terminated> LambdaDemo1 [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B.
Area of Squere is 36
```



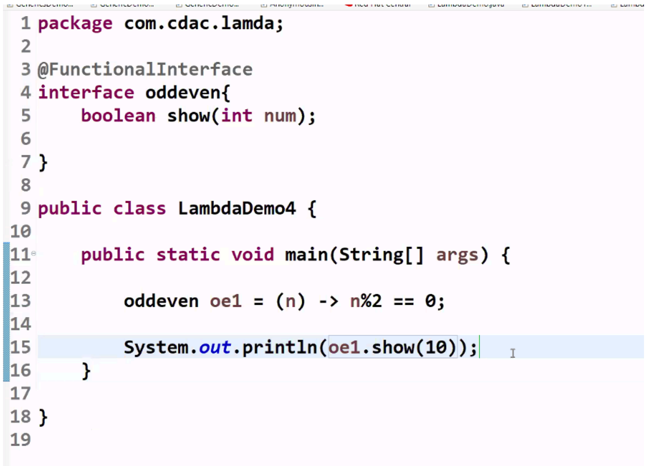
```
package com.cdac.lamda;

@FunctionalInterface
interface Square{
    int area(int x);
}

public class LambdaDemo1 {

    public static void main(String[] args) {

        Square s1 = (side) -> side * side;
        System.out.println("Area of square="+s1.area(6));


    }

}
```

Example 3

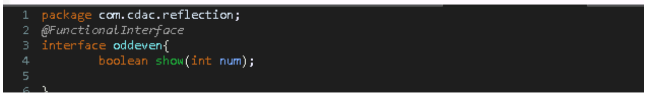


```
package com.cdac.reflection;

@FunctionalInterface
interface Test {
    int sum(int a, int b);
}

public class LambdaDemo2 {
    public static void main(String[] args) {
        Test t1 = (a, b) -> a + b; // Lambda expression
        System.out.println("Add = " + t1.sum(10, 20));
    }
}
```

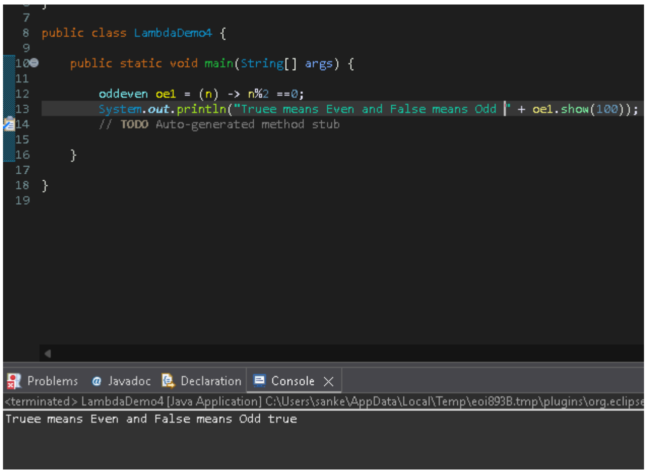
Example 4



```
package com.cdac.lamda;

@FunctionalInterface
interface oddeven{
    boolean show(int num);

}

public class LambdaDemo4 {

    public static void main(String[] args) {

        oddeven oe1 = (n) -> n%2 == 0;

        System.out.println(oe1.show(10));
    }

}
```



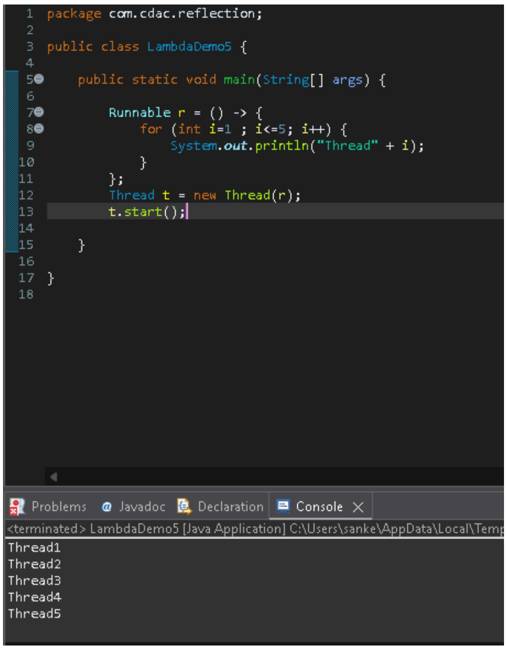
```
package com.cdac.reflection;
@FunctionalInterface
interface oddeven{
        boolean show(int num);

}
```



```
    }

public class LambdaDemo4 {

    public static void main(String[] args) {

        oddeven oe1 = (n) -> n%2 ==0;
        System.out.println("Truee means Even and False means Odd " + oe1.show(100));
        // TODO Auto-generated method stub

    }

}
```

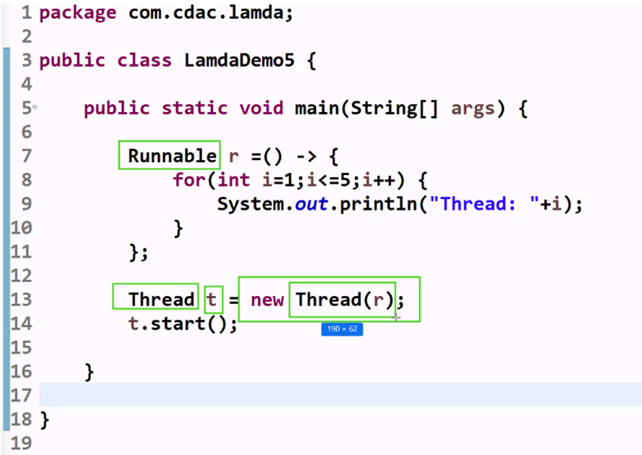
```
<terminated> LambdaDemo4 [Java Application] C:\Users\sanke\AppData\Local\Temp\eoi893B.tmp\plugins\org.eclipse
Truee means Even and False means Odd true
```

Example 5

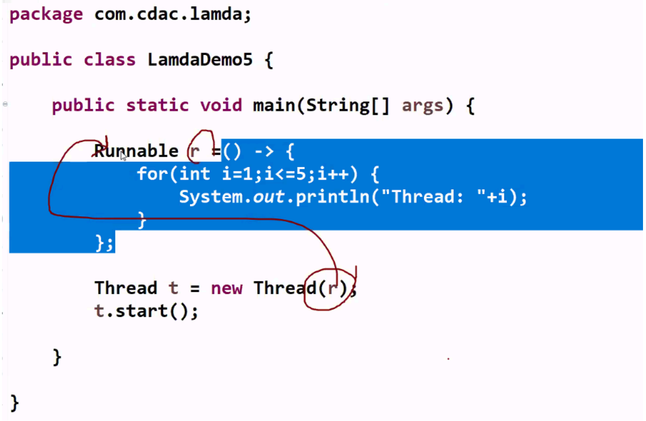


```
package com.cdac.reflection;

public class LambdaDemo5 {

    public static void main(String[] args) {

        Runnable r = () -> {
            for (int i=1 ; i<=5; i++) {
                System.out.println("Thread" + i);
            }
        };
        Thread t = new Thread(r);
        t.start();

    }

}
```

```
<terminated> LambdaDemo5 [Java Application] C:\Users\sanke\AppData\Local\Temp
Thread1
Thread2
Thread3
Thread4
Thread5
```



```
package com.cdac.lamda;

public class LamdaDemo5 {

    public static void main(String[] args) {

        Runnable r =() -> {
            for(int i=1;i<=5;i++) {
                System.out.println("Thread: "+i);
            }
        };

        Thread t = new Thread(r);
        t.start();

    }

}
```



```
package com.cdac.lamda;

public class LamdaDemo5 {

    public static void main(String[] args) {

        Runnable r =() -> {
            for(int i=1;i<=5;i++) {
                System.out.println("Thread: "+i);
            }
        };

        Thread t = new Thread(r);
        t.start();

    }

}
```

- `r.run()` → **just a method call (no multithreading).**
- `t.start()` → **tells JVM to create a new thread and then calls `run()` on that new thread.**

Multithreading :

Definition

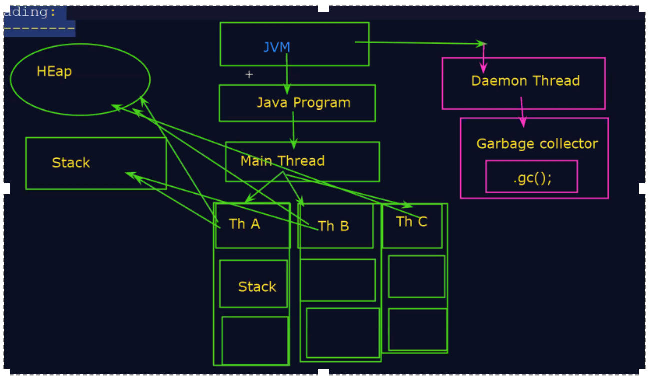
- Multithreading is the ability of a program to execute multiple threads concurrently. A thread is the smallest unit of execution within a process.

Short Explanation

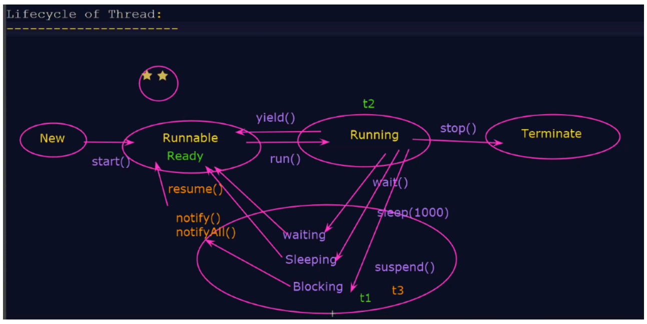
- A process is a program in execution, and each process can contain one or more threads.
- Multithreading means running two or more threads at the same time within the same program.
- Each thread runs independently but shares the same memory of the process.
- It improves performance and makes better use of CPU resources.
-

Key Points

- Threads run concurrently (sometimes in true parallel on multi-core CPUs).
- They share resources like memory, but run independently.
- Useful for tasks like downloading files, handling multiple users, animations, background tasks, etc.
- Helps in achieving faster execution and responsive programs.
- But needs proper synchronization to avoid issues like race conditions



Lifecycle of Thread :



Thread Lifecycle in Java :

A thread in Java goes through different states during its lifetime. These states are defined in the Thread.State enum.

1. New (Created)
The thread object is created using new Thread().
It is not yet started.
Method: start() (used to move it to Runnable state).

2. Runnable
After calling start(), the thread is ready to run and waiting for CPU scheduling.
Method: run() (contains the code to be executed, but JVM calls it, not us directly).

3. Running
When the thread scheduler picks the thread from Runnable, it goes to Running state.
Only one thread runs at a time per CPU core.

Method: No direct method; happens automatically when the scheduler selects the thread.

4. Waiting / Timed Waiting / Sleeping / Blocked

Threads can temporarily stop execution:
sleep(ms) → thread pauses for given time.
join() → current thread waits until another finishes.
wait() → thread waits until notified.
Blocked → waiting to acquire a lock (synchronization).

5. Terminated (Dead)

Once the run() method finishes, the thread enters terminated state.
It cannot be restarted.
Method: No method to restart; must create a new thread object.

Important Methods in Thread Lifecycle

1. start() → begins execution, moves thread to Runnable.
2. run() → contains task code (executed when thread starts).
3. sleep(ms) → pauses thread for some time.
4. join() → waits for another thread to finish.
5. wait() / notify() / notifyAll() → used in synchronization.
6. yield() → hints to scheduler to pause current thread and give chance to others.
7. interrupt() → interrupts a sleeping or waiting thread.
8. isAlive() → checks if a thread is still active.