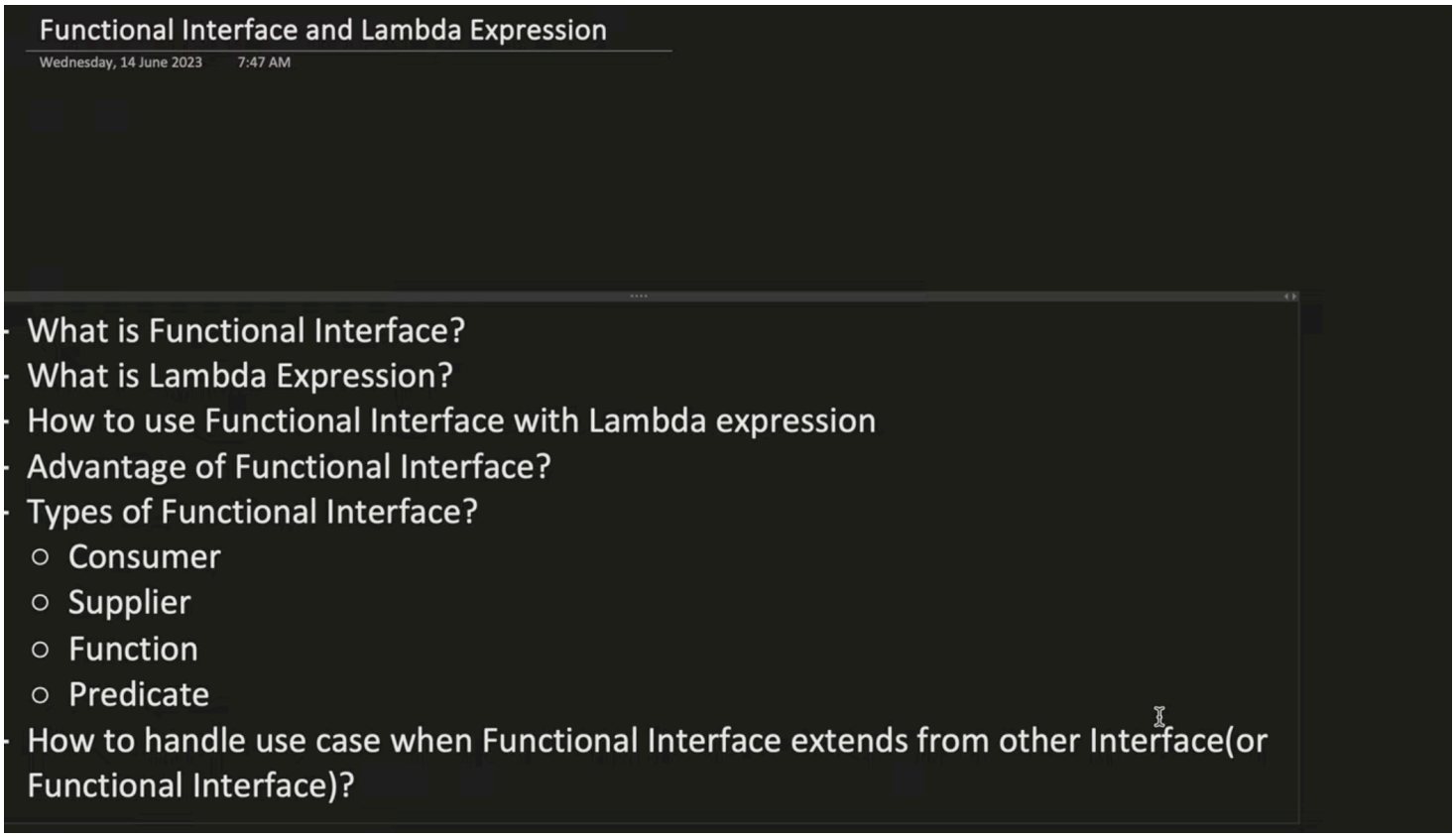
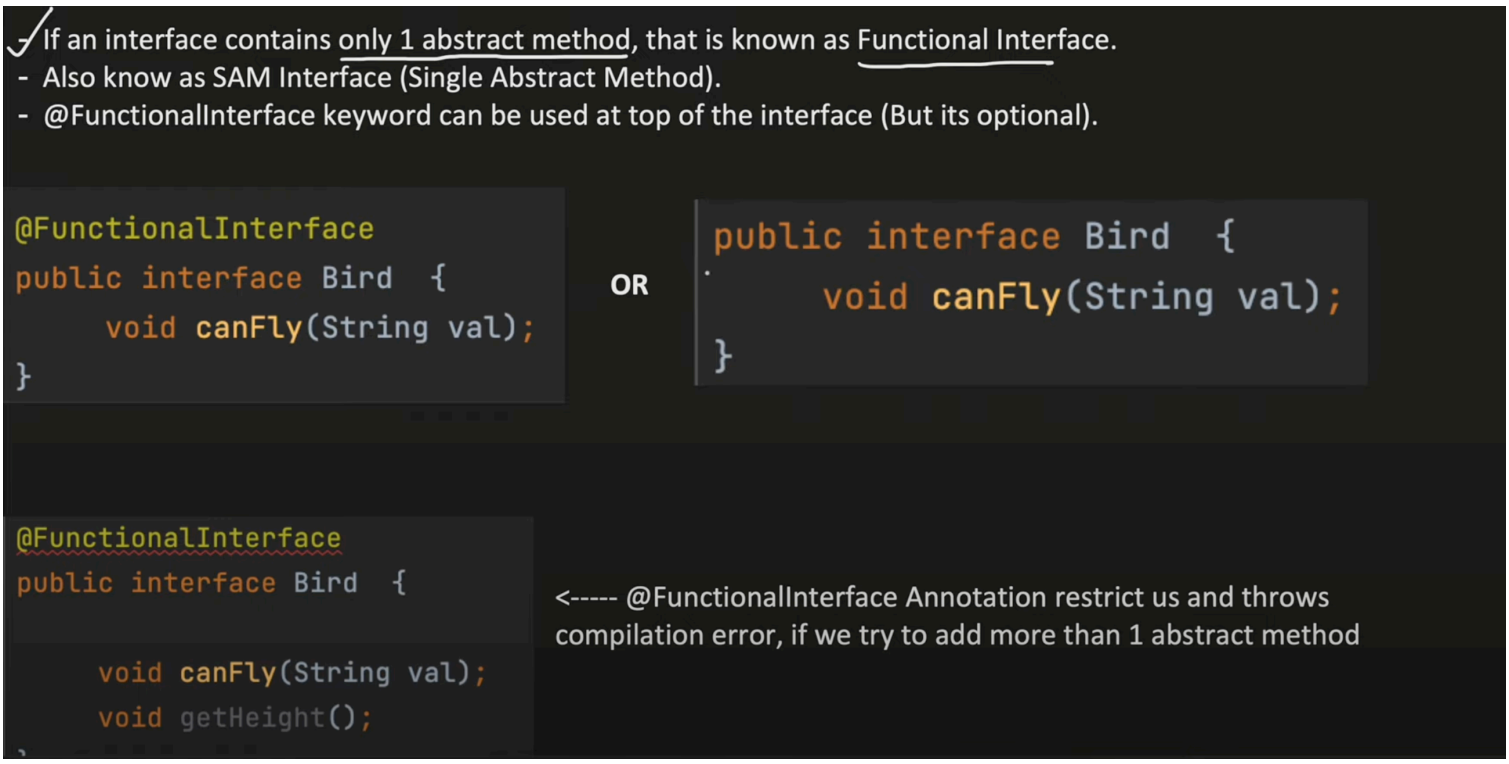
Java: Functional Interfaces,Lamba expressions Depth





```java
@FunctionalInterface
public interface Bird {

    void canFly(String val);

    default void getHeight(){
        //default method implementation
    }

    static void canEat(){
        //my static method implementation
    }

    String toString(); //Object class method
```

<----- In Functional Interface, only 1 abstract method is allowed, but we c
other methods like default, static method or Methods inherited from the
Class

What is Lambda Expression:

- Lambda expression is a way to implement the Functional Interface.

Before going into further into Lambda expression, lets first see:

Different Ways to Implements the Functional Interface:

1. **Using "implements"**
2. **Using "anonymous Class"**

Using "implements"

```java
@FunctionalInterface
public interface Bird  {

    void canFly(String val);

}
```

```java
public class Eagle implements Bird{

    @Override
    public void canFly(String val) {
        System.out.println("Eagle Bird Implementation");
    }
}
```

```
Bird eagleObject = new Eagle();
```

Using "anonymous Class"

```java
@FunctionalInterface
public interface Bird  {

    void canFly(String val);

}
```

```java
public class Main {

    public static void main(String args[]){

        Bird eagleObject = new Bird() {
            @Override
            public void canFly(String val) {
                System.out.println("Eagle Bird Imple
            }
        };
```

3

Using "Lambda Expression"

```java
@FunctionalInterface
public interface Bird  {

    void canFly(String val);

}
```

```java
public class Main {

    public static void main(String args[]){

        Bird eagleObject = (String value) -> {
                System.out.println("Eagle Bird Implementation");
            };
```

Types of Functional Interface:

in built

Consumer

- Represent an operation, that accept a single input parameter and returns no result.
- Present in package: java.util.function;

Supplier

- Represent the supplier of the result. Accepts no Input parameter but produce a result
- Present in package: java.util.function;

- Present in package: java.util.function;

```java
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

```java
public class Main {

    public static void main(String args[]) {

        Consumer<Integer> loggingObject = (Integer val) -> {
            if(val>10) {
                System.out.println("Logging");
            }
        };

        loggingObject.accept( t 11);
```

- Present in package: java.util.function;

```java
@FunctionalInterface
public interface Supplier<T> {

    T get();
}
```

```java
public class Main {

    public static void main(String args[]) {

        Supplier<String> isEvenNumber = () -> "this is the data i am

        System.out.println(isEvenNumber.get());
    }
}
```

OR

OR

```
public class Main {

    public static void main(String args[]) {

        Supplier<String> isEvenNumber = () -> {
            return "this is the data i am returning";
        };

        System.out.println(isEvenNumber.get());
    }
}
```

Function

- **Represent function, that accepts one argument process it and produce a result.**
- **Present in package:** java.util.function;

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
}
```

```
public class Main {

    public static void main(String args[]) {

        Function<Integer, String> integerToString =
                (Integer num) -> {
                    String output = num.toString();
                    return output;
                };

        System.out.println(integerToString.apply( t: 64));
    }
```

Predicate

- **Represent function, that accept one argument and return the boolean.**
- **Present in package:** java.util.function;

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);
}
```

```
public class Main {

    public static void main(String args[]) {

        Predicate<Integer> isEven = (Integer val) -> {

            if(val%2 == 0){
                return true;
            } else {
                return false;
            }
        };

        System.out.println(isEven.test( t: 19));
```

Handle use case when Functional Interface extends from other Inter

---

Use Case 1: Functional Interface extending Non Functional Interface

```java
public interface LivingThing {

    public void canBreathe();
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

```java
public interface LivingThing {

    default public boolean canBreathe(){
        return true;
    }
}
```

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

Use Case 2: Interface extending Functional Interface

```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

```java
public interface Bird extends LivingThing {

    void canFly(String val);
}
```

Use Case 3: Functional Interface extending other Functional Interface

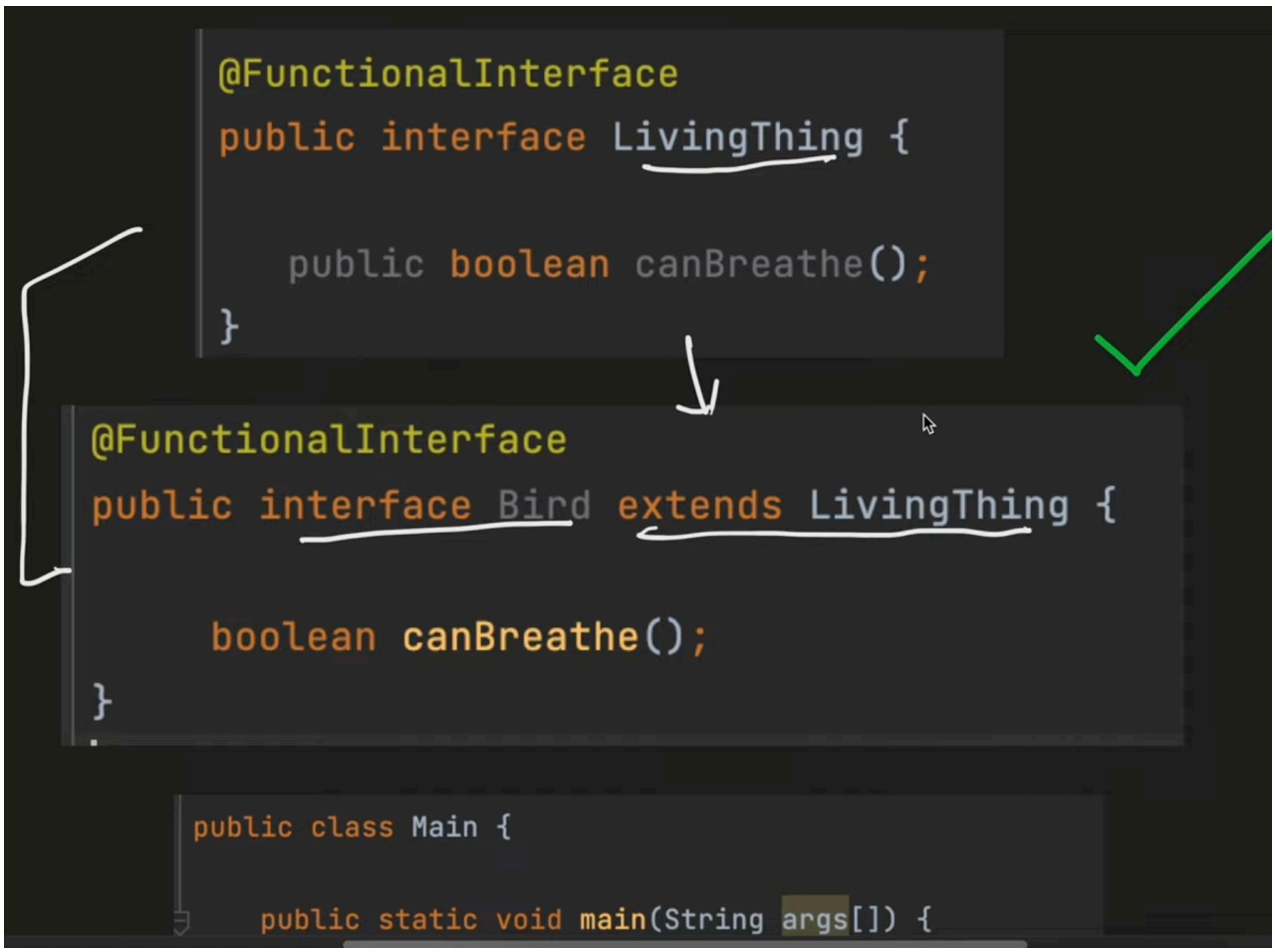
```java
@FunctionalInterface
public interface LivingThing {

    public boolean canBreathe();
}
```

1 abstract

```java
@FunctionalInterface
public interface Bird extends LivingThing {

    void canFly(String val);
}
```


@FunctionalInterface
public interface LivingThing {
public boolean canBreathe();
}
@FunctionalInterface
public interface Bird extends LivingThing {
boolean canBreathe();
}
public class Main {
public static void main(String args[]) {