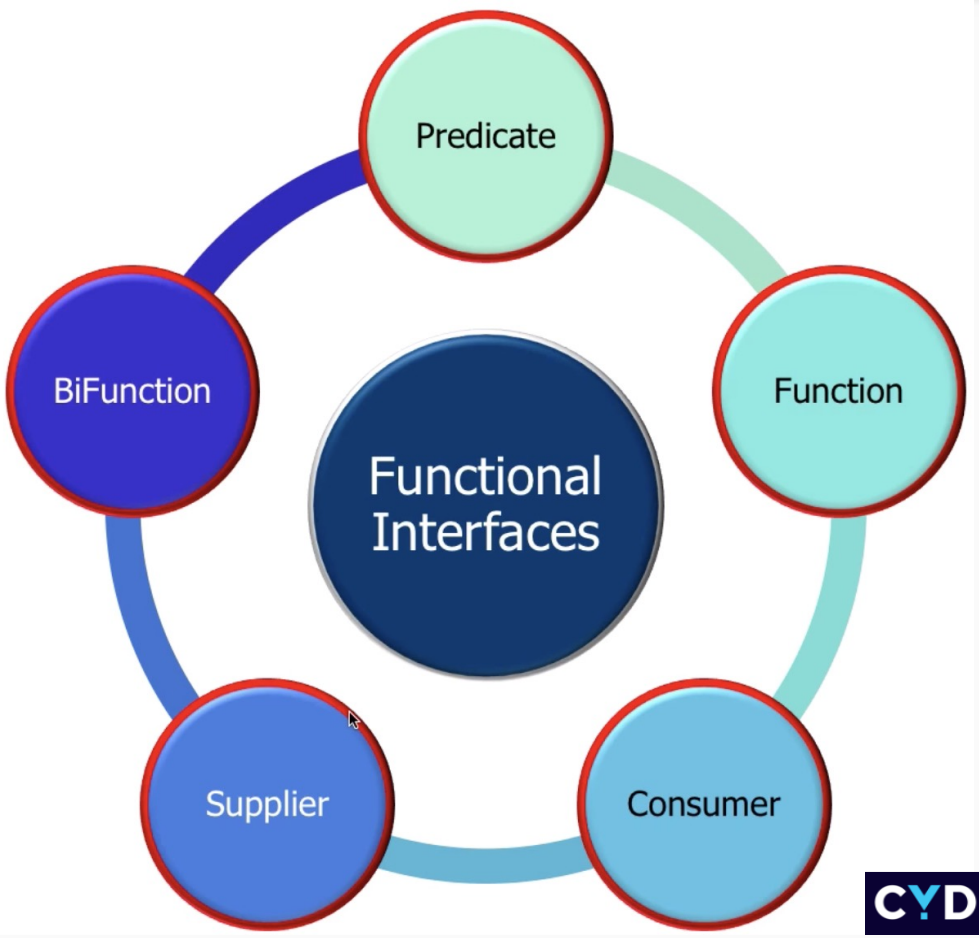
CYDEO

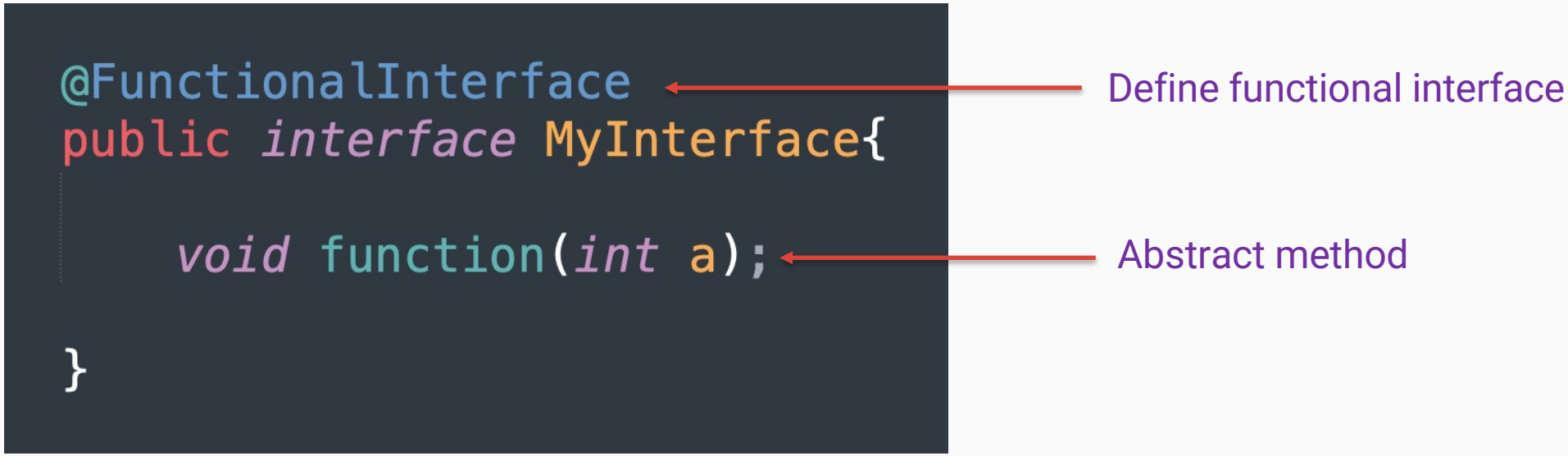
Functional Interface & Lambda Expression

Functional Interface (SAM)

- Known as SAM (Single Abstract Method) interface
- There is only one abstract method in the interface
- Effectively acts as a function
- @FunctionalInterface annotation is applicable (Optional)

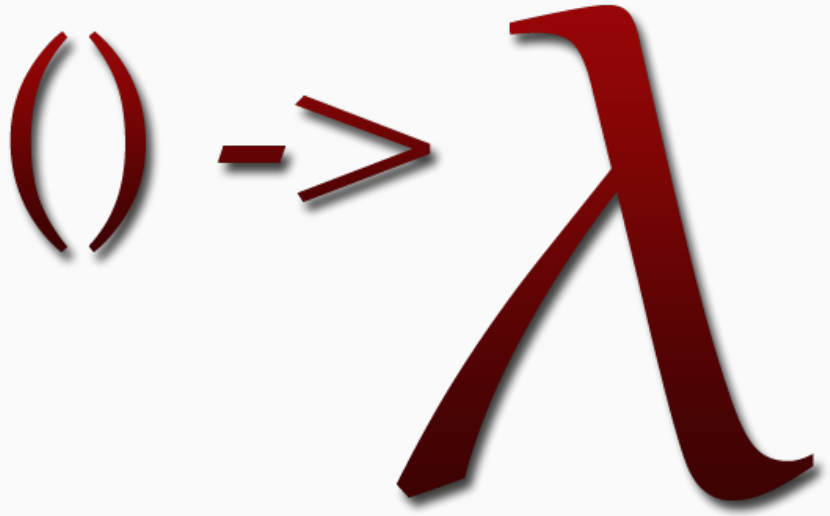


Functional Interface

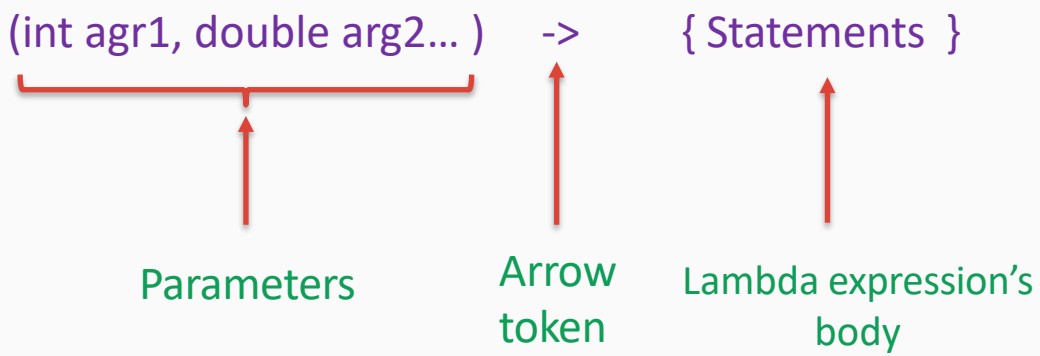


Lambda Expression

- A function with no name and an identifier
- Can be defined in the place where they are needed
- Expresses the instances of a functional Interface
- Can be assigned to the instance of functional interface



Syntax of Lambda Expression



Syntax	Description
( ) -> { statements }	Takes no argument and executes the given statement(s) in lambda expression's body
(Parameters) -> { statements }	Takes argument(s) and executes the given statements in lambda expression's body

Custom Functional interface Example

```java
@FunctionalInterface
public interface MyInterface{

    String test(String s1, String s2);

}
```

Abstract method

```java
public class Test{

    public static void main(String[] args) {

        MyInterface longestStr = (s1, s2) -> {
            if(s1.length() > s2.length())
                return s1;
            else
                return s2;
        };


        String str1 = longestStr.test("Java", "Wooden Spoon");

    }

}
```

Lambda Expression:
Implementation of functional interface' abstract method

CYDEO

Build In Functional Interfaces:

- Predicate
- Consumer
- Function
- BiPredicate
- BiConsumer
- BiFunction

Build in Functional Interface: Predicate

- Represents a function which takes one argument (any object) and returns boolean

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

    /**
     * Evaluates this predicate on the giv
     *
     * @param t the input argument
     * @return {@code true} if the input a
     * otherwise {@code false}
     */

    boolean test(T t);
```

Abstract method

Predicate Interface Example

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

    Predicate<Integer> isOdd = n ->  n % 2 != 0;

    boolean r1 = isOdd.test(100);

}
```

Lambda Expression: Implementation of predicate interface's abstract method

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

    Predicate<String> isPalindrome = s -> {

        String reverse = "";
        for(int i = s.length()-1; i >=0; i--){
            reverse += s.charAt(i);
        }

       return s.equalsIgnoreCase(reverse);
    };

    boolean r2 = isPalindrome.test("Wooden Spoon");

}
```

Lambda Expression: Implementation of predicate interface's abstract method

CYDEO

Build in Functional Interface: Consumer

- Represents a function which takes one argument (any object) and does not return a value

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

    /**
     * Performs this operation on th
     *
     * @param t the input argument
     */
    void accept(T t);
```

Abstract method

Consumer Interface Example

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

    Consumer<int[]> printEach = list -> {
        for (int each : list) {
            System.out.println(each);
        }
    };

    int[] numbers = {1,2,3,4,5};

    printEach.accept(numbers);

}
```

Lambda Expression:
Implementation of Consumer interface's abstract method

Build in Functional Interface: Function

- Represents a function which takes one argument (any object) and return a value (any object)

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

    /**
     * Applies this function to the g
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

Abstract method

Function Interface Example

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

    Function<int[], List<Integer>> convertToList = (arr) -> {
        List<Integer> result = new ArrayList<>();
        for (int each : arr) {
            result.add(each);
        }

        return result;
    };

    int[] numbers = {1,2,3,4,5};
    List<Integer> list = convertToList.apply(numbers);

}
```

Lambda Expression:
Implementation of function interface's abstract method

CYDEO

Build in Functional Interface: BiPredicate

- Represents a function which takes two arguments (any objects) and returns boolean

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    /** Evaluates this predicate on the
     * @param t the first input argument
     * @param u the second input argumen
     * @return {@code true} if the input
     * otherwise {@code false}*/

    boolean test(T t, U u);
```

Abstract method

BiPredicate Interface Example

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

    BiPredicate<int[], Integer> contains = (arr, n) -> {

        for (int each : arr) {
            if(each == n){
                return true;
            }
        }

        return false;
    };


    int[] numbers = {1,2,3,4,5};
    boolean r1 = contains.test(numbers, 6);

}
```

Lambda Expression:
Implementation of BiPredicate interface's abstract method

Build in Functional Interface: BiConsumer

- Represents a function which takes two argument (any object) and does not return a value

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    /**
     * Performs this operation on the
     *
     * @param t the first input argume
     * @param u the second input argum

    void accept(T t, U u);
```

Abstract method

BiConsumer Interface Example

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

    BiConsumer<String, Integer> printMultipleTimes = (s, n) -> {

        for (int i = 0; i < n; i++) {
            System.out.println(s);
        }

    };

    printMultipleTimes.accept("Wooden Spoon", 10);

}
```

Lambda Expression:
Implementation of
BiConsumer interface's
abstract method

Build in Functional Interface: BiFunction

- Represents a function which takes two arguments (any objects) and return a value (any object)

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    /**
     * Applies this function to the give
     *
     * @param t the first function argu
     * @param u the second function argu
     * @return the function result*/

    R apply(T t, U u);
```

Abstract method

BiFunction Interface Example

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

    BiFunction<int[], int[], List<Integer>> mergeArray = (arr1, arr2)->{
        List<Integer> list = new ArrayList<>();

        for (int each : arr1)
            list.add(each);

        for (int each : arr2)
            list.add(each);

        return list;

    };

   int[] n1 = {1,2,3,4};
   int[] n2 = {5,6,7};

   List<Integer> list = mergeArray.apply(n1, n2);

}
```

Lambda Expression:
Implementation of
BiPredicate interface's
abstract method

Stream

Stream

- Stream is not a data structure
- Stream is a method that takes inputs from a data structure (Array & Collection)
- Stream is unable to change the data structure

```
int[] numbers = {1,1,2,2,3,3,4,4};

Arrays.stream(numbers).distinct();
```

```
Set<String> set = new HashSet<>();

set.stream().map( n -> n.toUpperCase() );
```

```
List<Integer> list = new ArrayList<>();

list.stream().filter( p -> p%2==0);
```

Methods of Stream

- After calling the stream() function from an Array/Collection, we can access to the methods of stream

Method	Method	Methods
distinct()	collect()	toArray()
skip()	limit()	map()
filter()	count()	forEach()
allMatch()	anyMatch()	nonMatch()