LAMBDA EXPRESSIONS

Lambda Expression is an anonymous function which accepts a set of input parameters

and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

A Lambda Expression contains 3 parts:

1. Parameter List

A Lambda Expression can contain zero or one or more parameters. It is optional.

2. Lambda Arrow Operator

"->" is known as Lambda Arrow operator. It separates the parameters list and body.

3. Lambda Expression Body

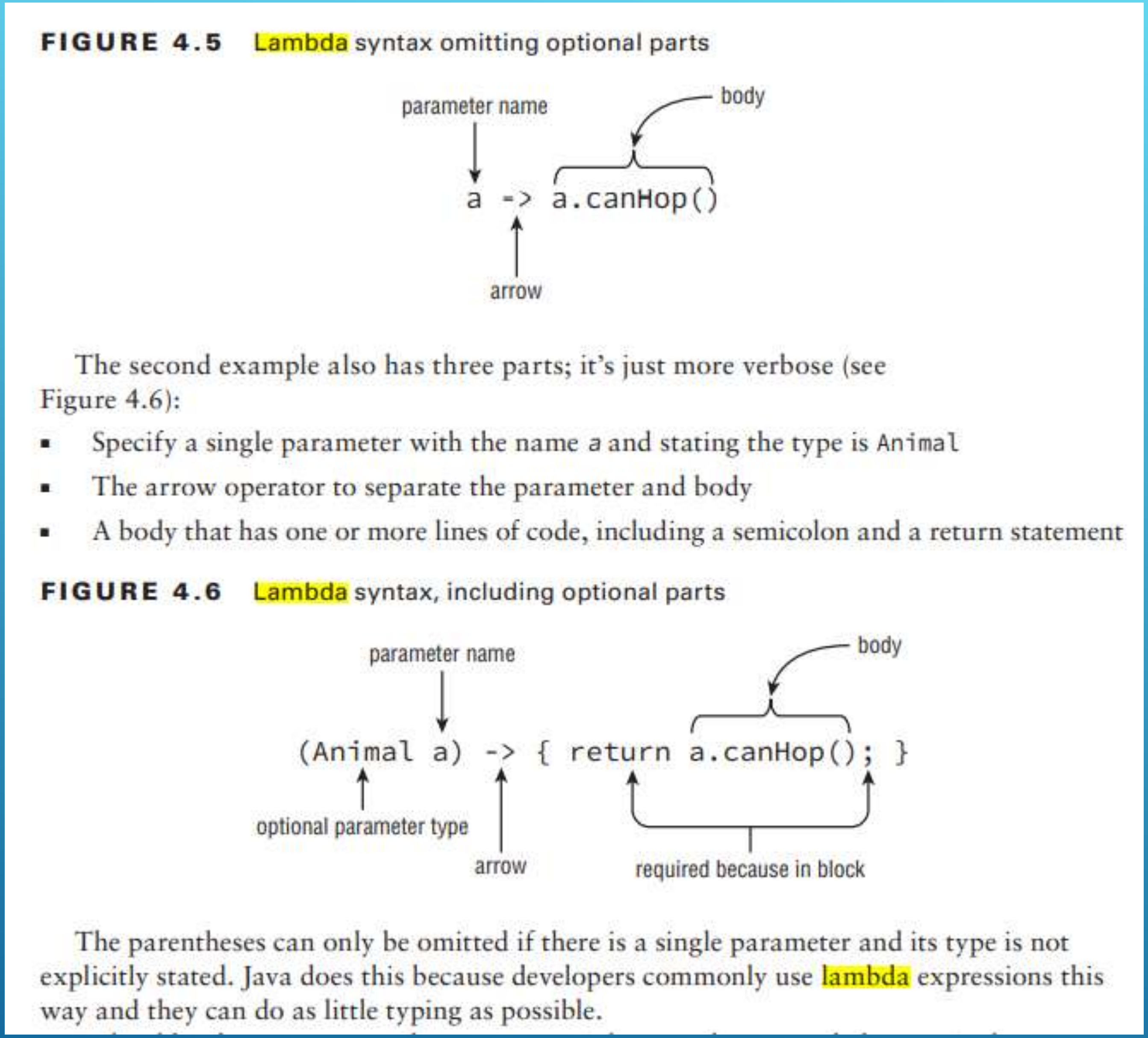
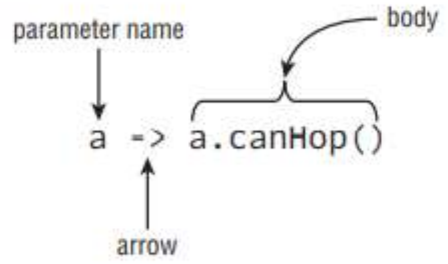


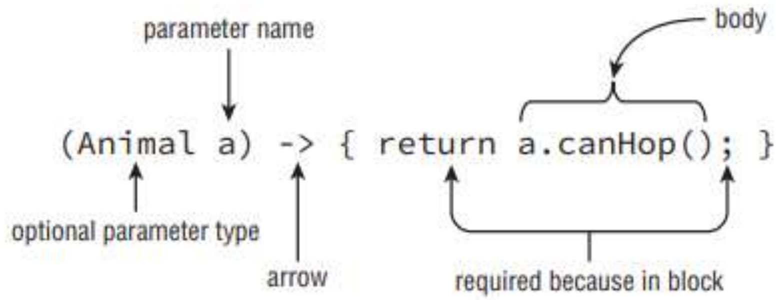
FIGURE 4.5 Lambda syntax omitting optional parts



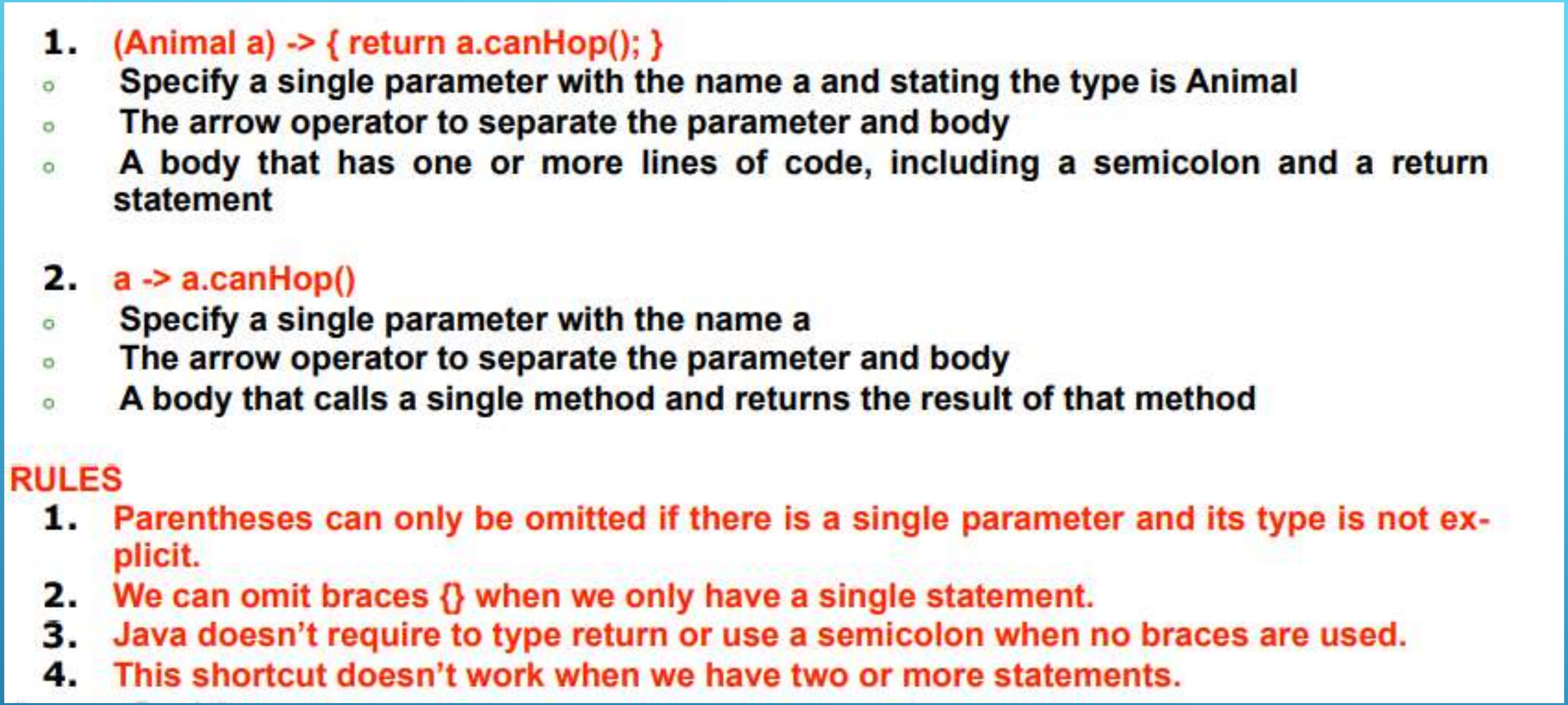
The second example also has three parts; it's just more verbose (see Figure 4.6):

- Specify a single parameter with the name *a* and stating the type is `Animal`
- The arrow operator to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a return statement

FIGURE 4.6 Lambda syntax, including optional parts


parameter name
body
(Animal a) -> { return a.canHop(); }
optional parameter type
arrow
required because in block

The parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do as little typing as possible.

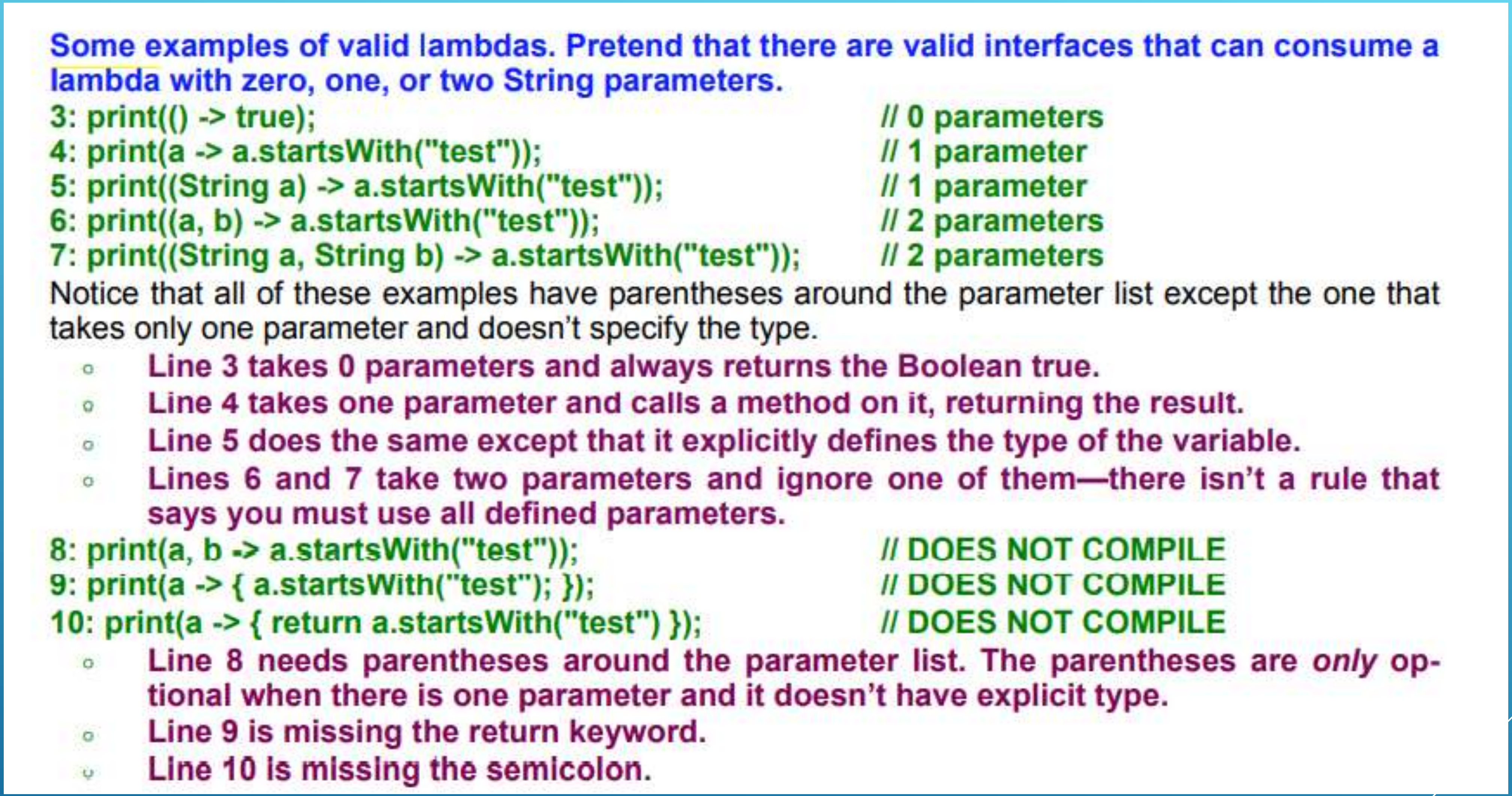


1. (Animal a) -> { return a.canHop(); }
   - Specify a single parameter with the name a and stating the type is Animal
   - The arrow operator to separate the parameter and body
   - A body that has one or more lines of code, including a semicolon and a return statement

2. a -> a.canHop()
   - Specify a single parameter with the name a
   - The arrow operator to separate the parameter and body
   - A body that calls a single method and returns the result of that method

RULES

1. Parentheses can only be omitted if there is a single parameter and its type is not explicit.
2. We can omit braces {} when we only have a single statement.
3. Java doesn’t require to type return or use a semicolon when no braces are used.
4. This shortcut doesn’t work when we have two or more statements.



Some examples of valid lambdas. Pretend that there are valid interfaces that can consume a lambda with zero, one, or two String parameters.

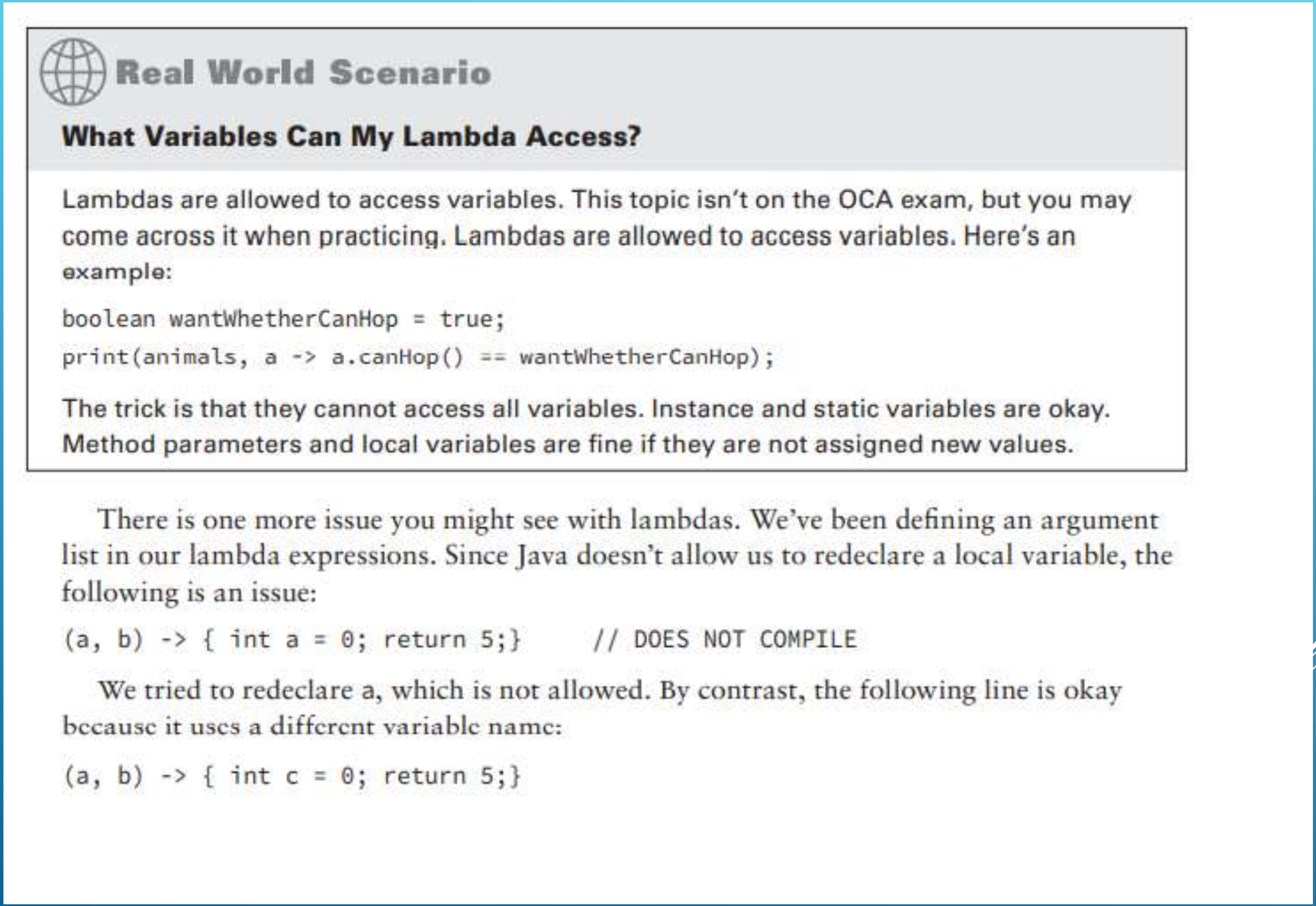
```
3: print(() -> true);                                   // 0 parameters
4: print(a -> a.startsWith("test"));                    // 1 parameter
5: print((String a) -> a.startsWith("test"));           // 1 parameter
6: print((a, b) -> a.startsWith("test"));               // 2 parameters
7: print((String a, String b) -> a.startsWith("test")); // 2 parameters
```

Notice that all of these examples have parentheses around the parameter list except the one that takes only one parameter and doesn't specify the type.

- **Line 3 takes 0 parameters and always returns the Boolean true.**
- **Line 4 takes one parameter and calls a method on it, returning the result.**
- **Line 5 does the same except that it explicitly defines the type of the variable.**
- **Lines 6 and 7 take two parameters and ignore one of them—there isn't a rule that says you must use all defined parameters.**

```
8: print(a, b -> a.startsWith("test"));                 // DOES NOT COMPILE
9: print(a -> { a.startsWith("test"); });               // DOES NOT COMPILE
10: print(a -> { return a.startsWith("test") });        // DOES NOT COMPILE
```

- **Line 8 needs parentheses around the parameter list. The parentheses are *only* optional when there is one parameter and it doesn't have explicit type.**
- **Line 9 is missing the return keyword.**
- **Line 10 is missing the semicolon.**



Real World Scenario

What Variables Can My Lambda Access?

Lambdas are allowed to access variables. This topic isn't on the OCA exam, but you may come across it when practicing. Lambdas are allowed to access variables. Here's an example:

```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```

The trick is that they cannot access all variables. Instance and static variables are okay. Method parameters and local variables are fine if they are not assigned new values.

There is one more issue you might see with lambdas. We've been defining an argument list in our lambda expressions. Since Java doesn't allow us to redeclare a local variable, the following is an issue:

```
(a, b) -> { int a = 0; return 5;}     // DOES NOT COMPILE
```

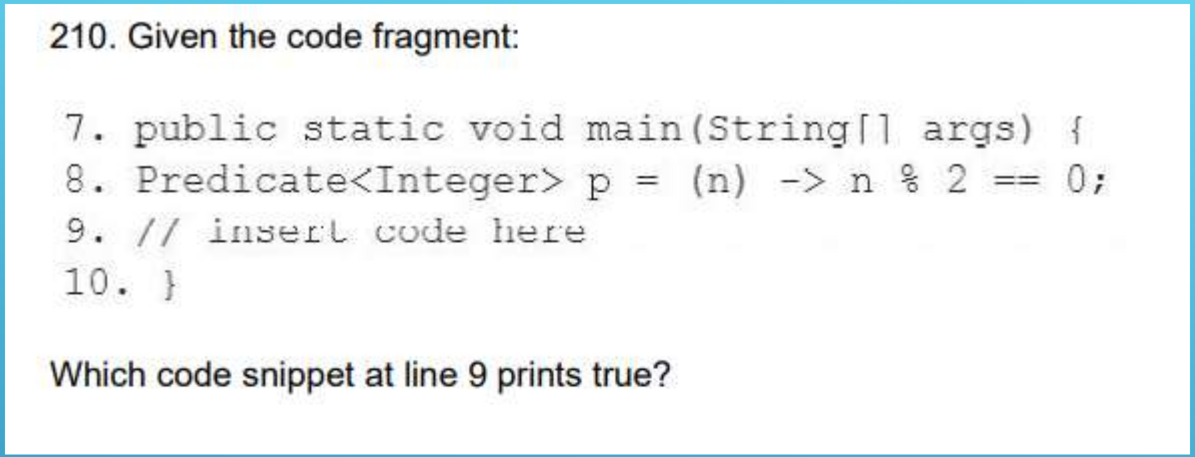
We tried to redeclare a, which is not allowed. By contrast, the following line is okay because it uses a different variable name:

```
(a, b) -> { int c = 0; return 5;}
```

NOTES

◦ It provides a clear and concise way to represent one method interface using an expression.
◦ An interface which has only one abstract method is called **Functional interface**.
◦ The Lambda expression is used to provide the implementation of Functional interface.
◦ It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation.
◦ Java provides an annotation @FunctionalInterface, which is used to declare an interface as functional interface.
◦ A functional interface is an interface that contains exactly one abstract method. It may contain zero or more default methods and/or static methods. Because a functional interface contains exactly one abstract method, you can omit the name of that method when you implement it using a lambda expression. For example, consider the following interface

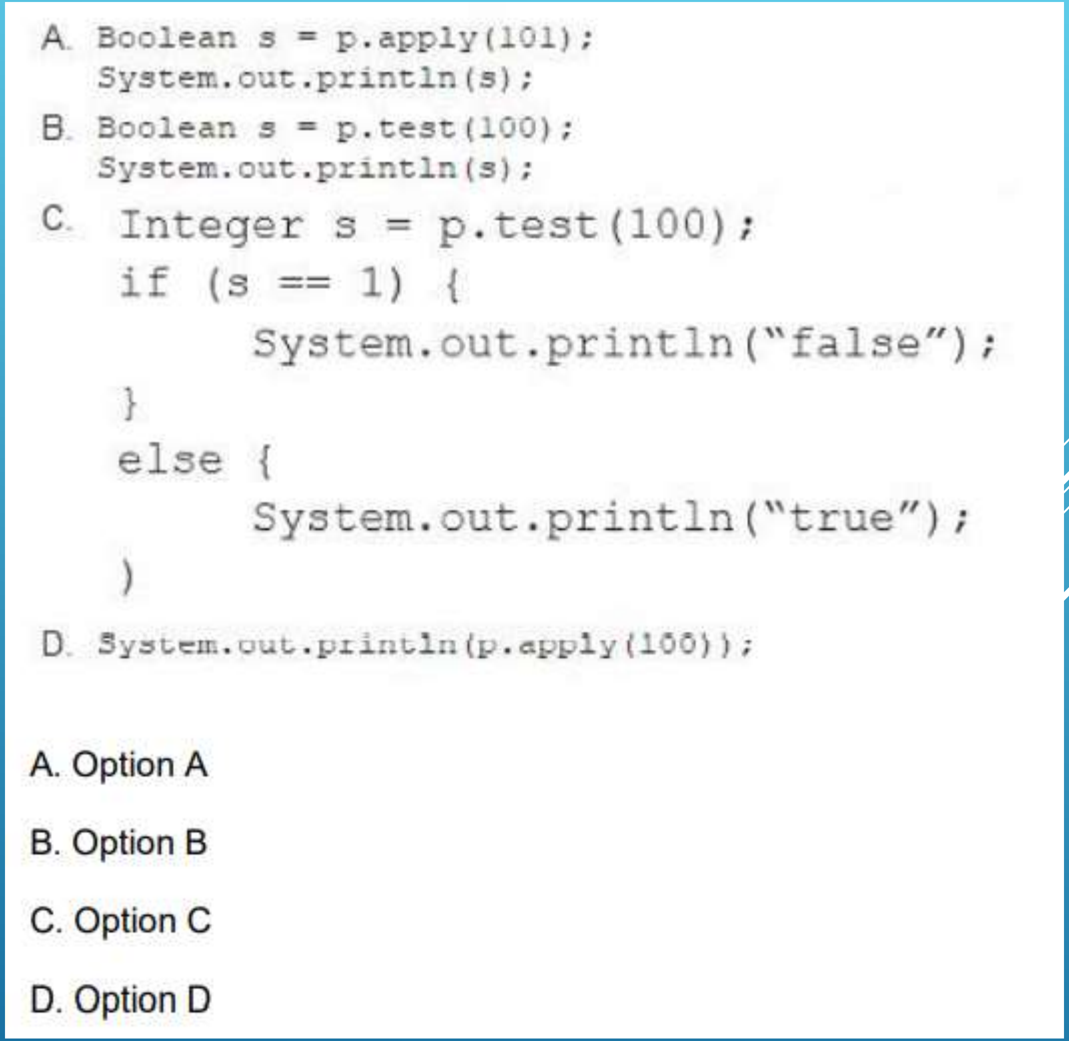
```
interface Predicate {

        boolean test(T t);

}
```



210. Given the code fragment:

```
7. public static void main(String[] args) {
8. Predicate<Integer> p = (n) -> n % 2 == 0;
9. // insert code here
10. }
```

Which code snippet at line 9 prints true?



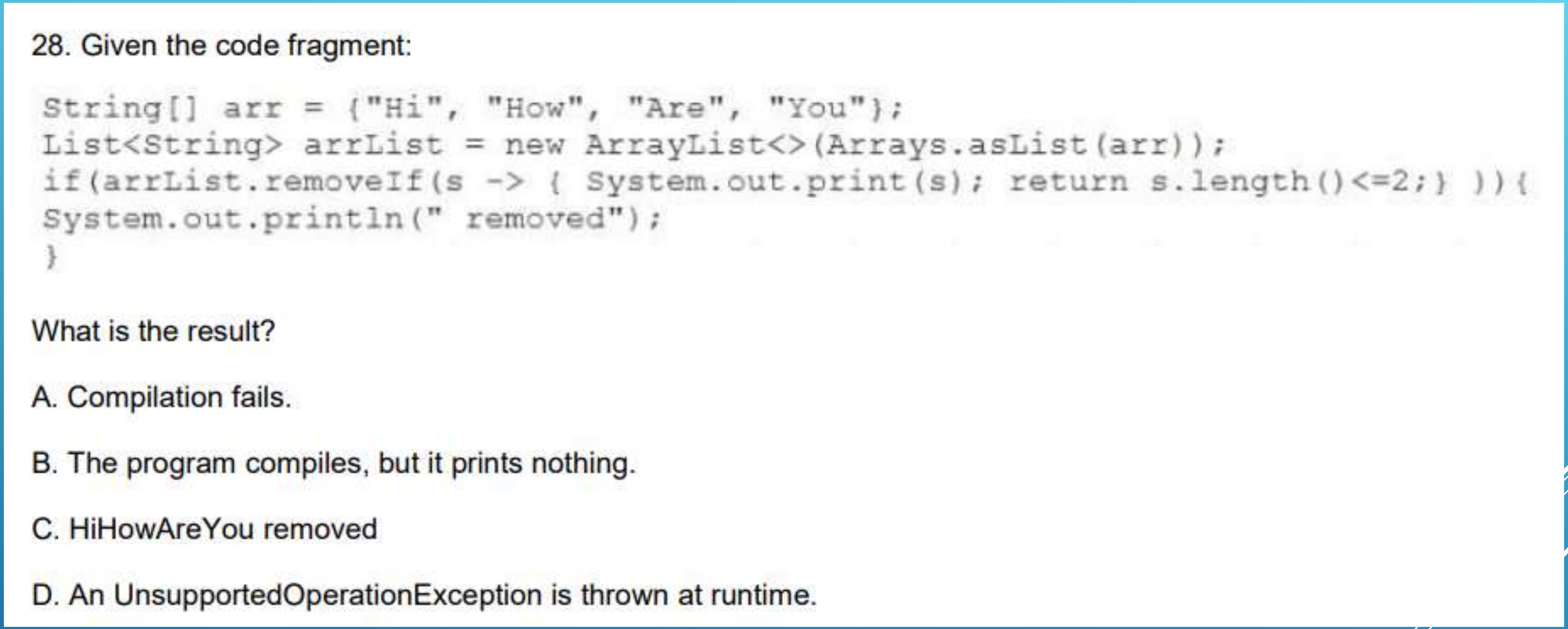
A.
```
Boolean s = p.apply(101);
System.out.println(s);
```
B.
```
Boolean s = p.test(100);
System.out.println(s);
```
C.
```
Integer s = p.test(100);
if (s == 1) {
      System.out.println("false");
}
else {
      System.out.println("true");
}
```
D.
```
System.out.println(p.apply(100));
```

A. Option A

B. Option B

C. Option C

D. Option D



28. Given the code fragment:

```
String[] arr = {"Hi", "How", "Are", "You"};
List<String> arrList = new ArrayList<>(Arrays.asList(arr));
if(arrList.removeIf(s -> { System.out.print(s); return s.length()<=2;} )){
System.out.println(" removed");
}
```

What is the result?

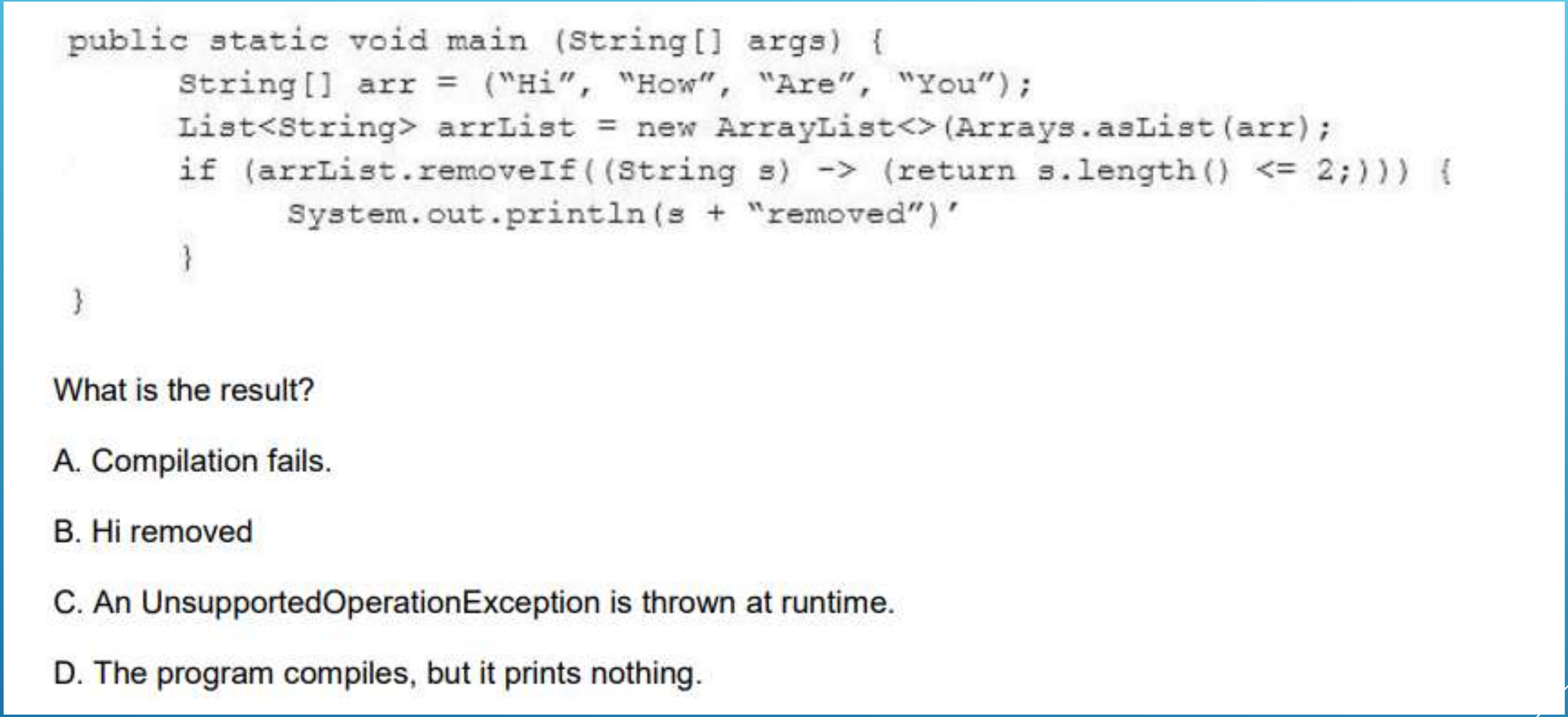
A. Compilation fails.

B. The program compiles, but it prints nothing.

C. HiHowAreYou removed

D. An UnsupportedOperationException is thrown at runtime.

43. Given the code fragment:



```
public static void main (String[] args) {
      String[] arr = (“Hi”, “How”, “Are”, “You”);
      List<String> arrList = new ArrayList<>(Arrays.asList(arr);
      if (arrList.removeIf((String s) -> (return s.length() <= 2;))) {
            System.out.println(s + “removed”)’
      }
}
```

What is the result?

A. Compilation fails.

B. Hi removed

C. An UnsupportedOperationException is thrown at runtime.

D. The program compiles, but it prints nothing.

57. Given the code fragments:

Person.java:

```
public class Person {
    String name;
    int age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Test.java:

```
public static void checkAge(List<Person> list, Predicate<Person> predicate) {
    for (Person p : list) {
        if (predicate.test(p)) {
            System.out.println(p.name + " ");
        }
    }
}

public static void main(String[] args) {
    List<Person> iList = Arrays.asList(new Person("Hank", 45),
                                       new Person("Charlie", 40),
                                       new Person("Smith", 38));
    //line n1
}
```

Which code fragment, when inserted at line n1, enables the code to print Hank?

A

```
checkAge (iList, ( ) -> p. get Age ( ) > 40);
```

B

```
checkAge(iList, Person p -> p.getAge( ) > 40);
```

C

```
checkAge (iList, p -> p.getAge ( ) > 40);
```

D

```
checkAge(iList, (Person p) -> { p.getAge() > 40; });
```

136. Given the code fragment:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Tech");
arrayList.add("Expert");
arrayList.set(0, "Java");
arrayList.forEach (a -> a.concat("Forum"));
arrayList.replaceAll (s -> s.concat("Group"));
System.out.println(arrayList);
```

What is the result?

A. [JavaForum, ExpertForum]

B. [JavaGroup, ExpertGroup]

C. [JavaForumGroup, ExpertForumGroup]

D. [JavaGroup, TechGroup ExpertGroup]