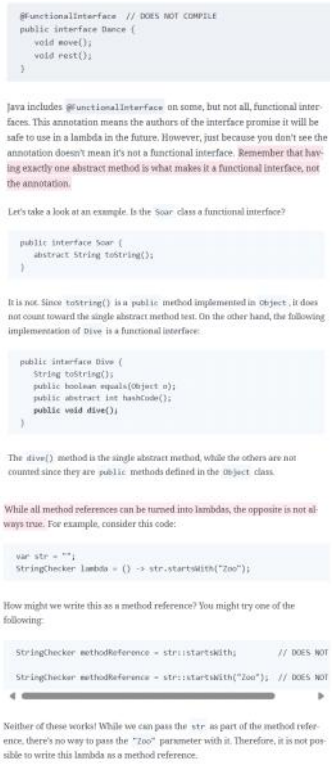
# Chapter8

Friday, October 17, 2025 9:21 AM

## Functional Interfaces



```
@FunctionalInterface  // DOES NOT COMPILE
public interface Dance {
   void move();
   void rest();
}
```

Java includes `@FunctionalInterface` on some, but not all, functional interfaces. This annotation means the authors of the interface promise it will be safe to use in a lambda in the future. However, just because you don't see the annotation doesn't mean it's not a functional interface. Remember that having exactly one abstract method is what makes it a functional interface, not the annotation.

Let's take a look at an example. Is the `Soar` class a functional interface?

```
public interface Soar {
   abstract String toString();
}
```

It is not. Since `toString()` is a `public` method implemented in `Object`, it does not count toward the single abstract method test. On the other hand, the following implementation of `Dive` is a functional interface:

```
public interface Dive {
   String toString();
   public boolean equals(Object o);
   public abstract int hashCode();
   public void dive();
}
```

The `dive()` method is the single abstract method, while the others are not counted since they are `public` methods defined in the `Object` class.

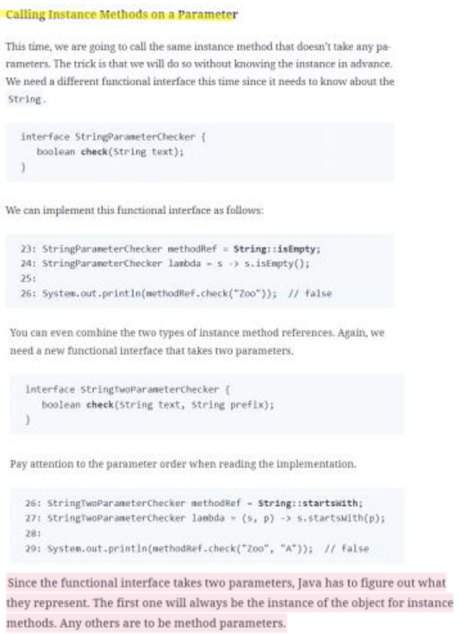
While all method references can be turned into lambdas, the opposite is not always true. For example, consider this code:

```
var str = "";
StringChecker lambda = () -> str.startsWith("Zoo");
```

How might we write this as a method reference? You might try one of the following:

```
StringChecker methodReference = str::startsWith;         // DOES NOT

StringChecker methodReference = str::startsWith("Zoo");  // DOES NOT
```

Neither of these works! While we can pass the `str` as part of the method reference, there's no way to pass the `"Zoo"` parameter with it. Therefore, it is not possible to write this lambda as a method reference.



### Calling Instance Methods on a Parameter

This time, we are going to call the same instance method that doesn't take any parameters. The trick is that we will do so without knowing the instance in advance. We need a different functional interface this time since it needs to know about the `String`.

```
interface StringParameterChecker {
   boolean check(String text);
}
```

We can implement this functional interface as follows:

```
23: StringParameterChecker methodRef = String::isEmpty;
24: StringParameterChecker lambda = s -> s.isEmpty();
25:
26: System.out.println(methodRef.check("Zoo"));  // false
```

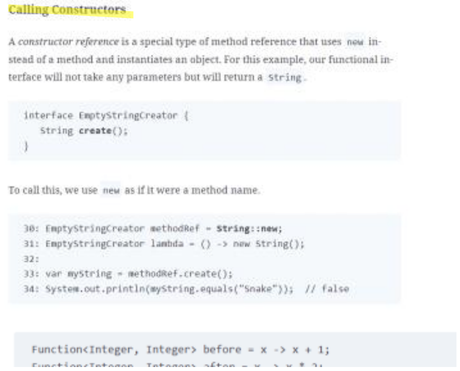
You can even combine the two types of instance method references. Again, we need a new functional interface that takes two parameters.

```
interface StringTwoParameterChecker {
   boolean check(String text, String prefix);
}
```

Pay attention to the parameter order when reading the implementation.

```
26: StringTwoParameterChecker methodRef = String::startsWith;
27: StringTwoParameterChecker lambda = (s, p) -> s.startsWith(p);
28:
29: System.out.println(methodRef.check("Zoo", "A"));  // false
```

Since the functional interface takes two parameters, Java has to figure out what they represent. The first one will always be the instance of the object for instance methods. Any others are to be method parameters.



### Calling Constructors

A *constructor reference* is a special type of method reference that uses `new` instead of a method and instantiates an object. For this example, our functional interface will not take any parameters but will return a `String`.

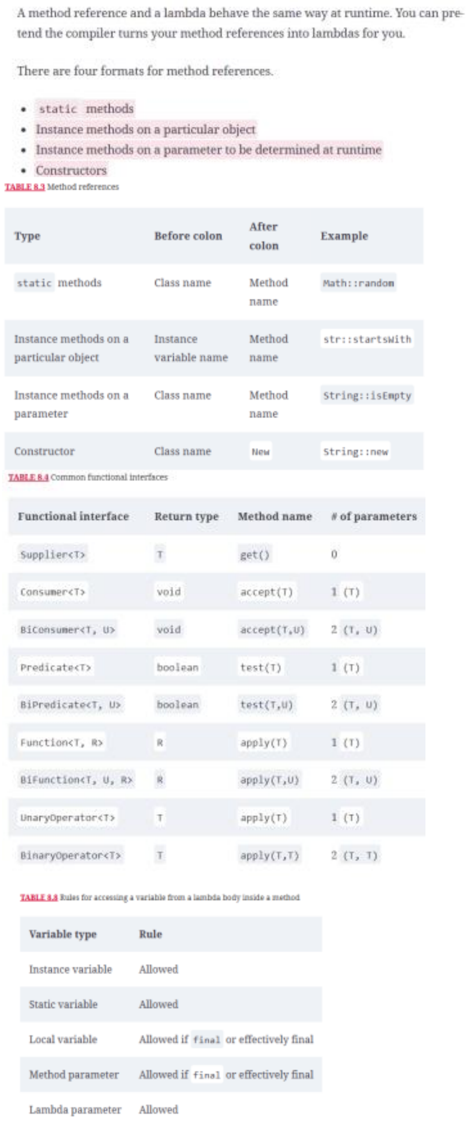
```
interface EmptyStringCreator {
   String create();
}
```

To call this, we use `new` as if it were a method name.

```
30: EmptyStringCreator methodRef = String::new;
31: EmptyStringCreator lambda = () -> new String();
32:
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
```

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;
```

## RULES



A method reference and a lambda behave the same way at runtime. You can pretend the compiler turns your method references into lambdas for you.

There are four formats for method references.

- `static` methods
- Instance methods on a particular object
- Instance methods on a parameter to be determined at runtime
- Constructors

TABLE 8.3 Method references

| Type | Before colon | After colon | Example |
|---|---|---|---|
| `static` methods | Class name | Method name | `Math::random` |
| Instance methods on a particular object | Instance variable name | Method name | `str::startsWith` |
| Instance methods on a parameter | Class name | Method name | `String::isEmpty` |
| Constructor | Class name | `New` | `String::new` |

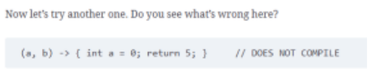
TABLE 8.4 Common functional interfaces

| Functional interface | Return type | Method name | # of parameters |
|---|---|---|---|
| `Supplier<T>` | `T` | `get()` | 0 |
| `Consumer<T>` | `void` | `accept(T)` | 1 `(T)` |
| `BiConsumer<T, U>` | `void` | `accept(T,U)` | 2 `(T, U)` |
| `Predicate<T>` | `boolean` | `test(T)` | 1 `(T)` |
| `BiPredicate<T, U>` | `boolean` | `test(T,U)` | 2 `(T, U)` |
| `Function<T, R>` | `R` | `apply(T)` | 1 `(T)` |
| `BiFunction<T, U, R>` | `R` | `apply(T,U)` | 2 `(T, U)` |
| `UnaryOperator<T>` | `T` | `apply(T)` | 1 `(T)` |
| `BinaryOperator<T>` | `T` | `apply(T,T)` | 2 `(T, T)` |

TABLE 8.5 Rules for accessing a variable from a lambda body inside a method

| Variable type | Rule |
|---|---|
| Instance variable | Allowed |
| Static variable | Allowed |
| Local variable | Allowed if `final` or effectively final |
| Method parameter | Allowed if `final` or effectively final |
| Lambda parameter | Allowed |

## Tips

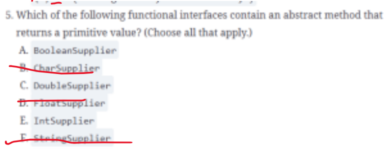


Now let's try another one. Do you see what's wrong here?

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

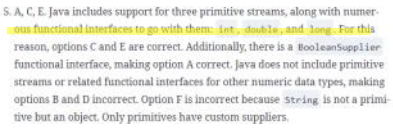
## EXAM

1.A
2.c
3.A,C
4.A,F
5.A,C,~~D~~ E



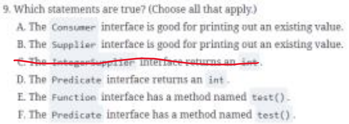
5. Which of the following functional interfaces contain an abstract method that returns a primitive value? (Choose all that apply.)

A. `BooleanSupplier`
~~B. `CharSupplier`~~
C. `DoubleSupplier`
~~D. `FloatSupplier`~~
E. `IntSupplier`
~~F. `StringSupplier`~~



5. A, C, E. Java includes support for three primitive streams, along with numerous functional interfaces to go with them: `int`, `double`, and `long`. For this reason, options C and E are correct. Additionally, there is a `BooleanSupplier` functional interface, making option A correct. Java does not include primitive streams or related functional interfaces for other numeric data types, making options B and D incorrect. Option F is incorrect because `String` is not a primitive but an object. Only primitives have custom suppliers.

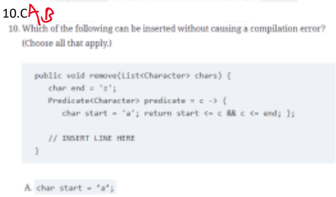
6.A,C
7.E
8.E
9.A,~~C~~,F



9. Which statements are true? (Choose all that apply.)

A. The `Consumer` interface is good for printing out an existing value.
B. The `Supplier` interface is good for printing out an existing value.
~~C. The `IntegerSupplier` interface returns an `int`.~~
D. The `Predicate` interface returns an `int`.
E. The `Function` interface has a method named `test()`.
F. The `Predicate` interface has a method named `test()`.

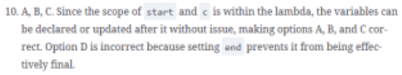
10.C A,B



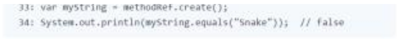
10. Which of the following can be inserted without causing a compilation error? (Choose all that apply.)

```
public void remove(List<Character> chars) {
   char end = 'z';
   Predicate<Character> predicate = c -> {
      char start = 'a'; return start <= c && c <= end; };

   // INSERT LINE HERE
}
```

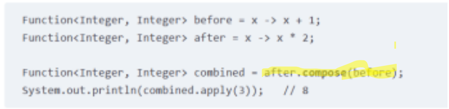
A. `char start = 'a';`



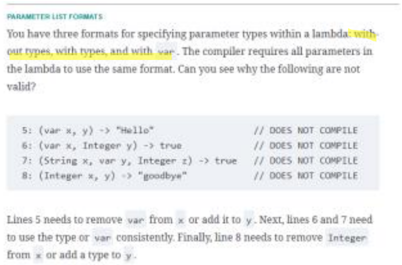
10. A, B, C. Since the scope of `start` and `c` is within the lambda, the variables can be declared or updated after it without issue, making options A, B, and C correct. Option D is incorrect because setting `end` prevents it from being effectively final.



```
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
```



```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3));   // 8
```

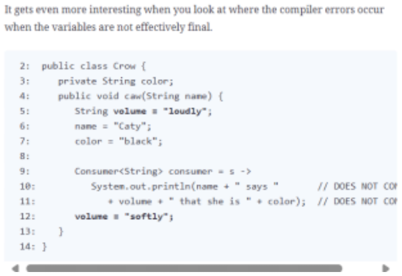


PARAMETER LIST FORMATS

You have three formats for specifying parameter types within a lambda: without types, with types, and with `var`. The compiler requires all parameters in the lambda to use the same format. Can you see why the following are not valid?

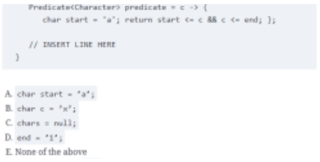
```
5: (var x, y) -> "Hello"                 // DOES NOT COMPILE
6: (var x, Integer y) -> true            // DOES NOT COMPILE
7: (String x, var y, Integer z) -> true  // DOES NOT COMPILE
8: (Integer x, y) -> "goodbye"           // DOES NOT COMPILE
```

Lines 5 needs to remove `var` from `x` or add it to `y`. Next, lines 6 and 7 need to use the type or `var` consistently. Finally, line 8 needs to remove `Integer` from `x` or add a type to `y`.



It gets even more interesting when you look at where the compiler errors occur when the variables are not effectively final.

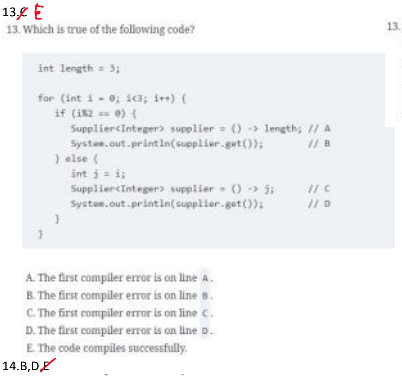
```
2:  public class Crow {
3:     private String color;
4:     public void caw(String name) {
5:        String volume = "loudly";
6:        name = "Caty";
7:        color = "black";
8:
9:        Consumer<String> consumer = s ->
10:          System.out.println(name + " says "        // DOES NOT CO
11:             + volume + " that she is " + color);  // DOES NOT CO
12:       volume = "softly";
13:    }
14: }
```



```
   Predicate<Character> predicate = c -> {
      char start = 'a'; return start <= c && c <= end; };

   // INSERT LINE HERE
}
```

A. `char start = 'a';`
B. `char c = 'x';`
C. `chars = null;`
D. `end = '1';`
E. None of the above

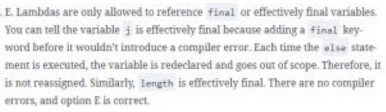
11.D
12.a
13.~~C~~ E



13. Which is true of the following code?

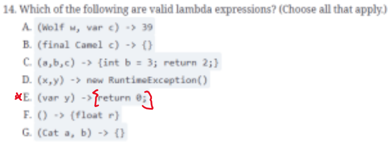
```
int length = 3;

for (int i = 0; i<3; i++) {
   if (i%2 == 0) {
      Supplier<Integer> supplier = () -> length; // A
      System.out.println(supplier.get());        // B
   } else {
      int j = i;
      Supplier<Integer> supplier = () -> j;      // C
      System.out.println(supplier.get());        // D
   }
}
```

A. The first compiler error is on line `A`.
B. The first compiler error is on line `B`.
C. The first compiler error is on line `C`.
D. The first compiler error is on line `D`.
E. The code compiles successfully.



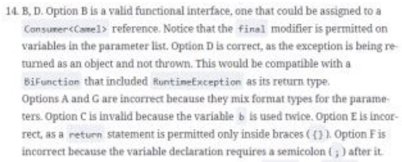
13. E. Lambdas are only allowed to reference `final` or effectively final variables. You can tell the variable `j` is effectively final because adding a `final` keyword before it wouldn't introduce a compiler error. Each time the `else` statement is executed, the variable is redeclared and goes out of scope. Therefore, it is not reassigned. Similarly, `length` is effectively final. There are no compiler errors, and option E is correct.

14.B,D,~~E~~



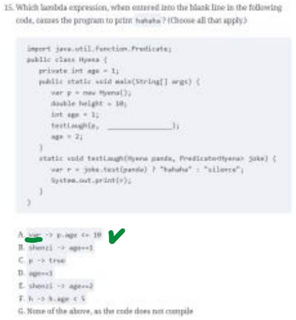
14. Which of the following are valid lambda expressions? (Choose all that apply.)

A. `(Wolf w, var c) -> 39`
B. `(final Camel c) -> {}`
C. `(a,b,c) -> {int b = 3; return 2;}`
D. `(x,y) -> new RuntimeException()`
×E. `(var y) -> {return 0;}`
F. `() -> {float r}`
G. `(Cat a, b) -> {}`



14. B, D. Option B is a valid functional interface, one that could be assigned to a `Consumer<Camel>` reference. Notice that the `final` modifier is permitted on variables in the parameter list. Option D is correct, as the exception is being returned as an object and not thrown. This would be compatible with a `BiFunction` that included `RuntimeException` as its return type.
Options A and G are incorrect because they mix format types for the parameters. Option C is invalid because the variable `b` is used twice. Option E is incorrect, as a `return` statement is permitted only inside braces (`{}`). Option F is incorrect because the variable declaration requires a semicolon (`;`) after it.

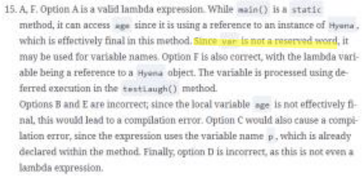
15.F A



15. Which lambda expression, when entered into the blank line in the following code, causes the program to print `hahaha`? (Choose all that apply.)

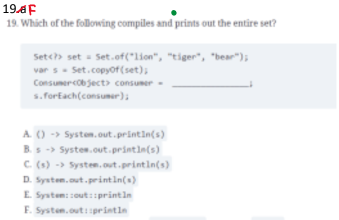
```
import java.util.function.Predicate;
public class Hyena {
   private int age = 1;
   public static void main(String[] args) {
      var p = new Hyena();
      double height = 10;
      int age = 1;
      testLaugh(p, _______________);
      age = 2;
   }
   static void testLaugh(Hyena panda, Predicate<Hyena> joke) {
      var r = joke.test(panda) ? "hahaha" : "silence";
      System.out.print(r);
   }
}
```

A. `var -> p.age <= 10` ✓
B. `shenzi -> age==1`
C. `p -> true`
D. `age==1`
E. `shenzi -> age==2`
F. `h -> h.age < 5`
G. None of the above, as the code does not compile



15. A, F. Option A is a valid lambda expression. While `main()` is a `static` method, it can access `age` since it is using a reference to an instance of `Hyena`, which is effectively final in this method. Since `var` is not a reserved word, it may be used for variable names. Option F is also correct, with the lambda variable being a reference to a `Hyena` object. The variable is processed using deferred execution in the `testLaugh()` method.
Options B and E are incorrect; since the local variable `age` is not effectively final, this would lead to a compilation error. Option C would also cause a compilation error, since the expression uses the variable name `p`, which is already declared within the method. Finally, option D is incorrect, as this is not even a lambda expression.

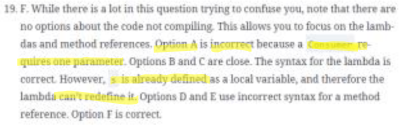
16.c
17.c
18.b,f,g
19.~~a~~F



19. Which of the following compiles and prints out the entire set?

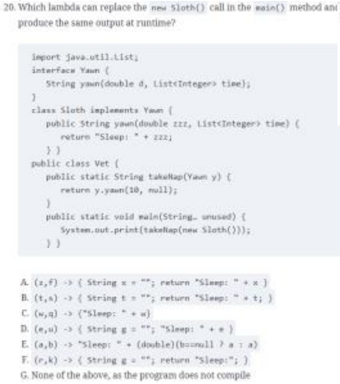
```
Set<?> set = Set.of("lion", "tiger", "bear");
var s = Set.copyOf(set);
Consumer<Object> consumer = _______________;
s.forEach(consumer);
```

A. `() -> System.out.println(s)`
B. `s -> System.out.println(s)`
C. `(s) -> System.out.println(s)`
D. `System.out.println(s)`
E. `System::out::println`
F. `System.out::println`



19. F. While there is a lot in this question trying to confuse you, note that there are no options about the code not compiling. This allows you to focus on the lambdas and method references. Option A is incorrect because a `Consumer` requires one parameter. Options B and C are close. The syntax for the lambda is correct. However, `s` is already defined as a local variable, and therefore the lambda can't redefine it. Options D and E use incorrect syntax for a method reference. Option F is correct.

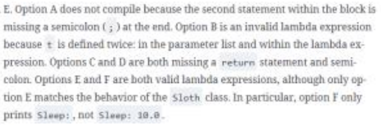
20.~~g~~E



20. Which lambda can replace the `new Sloth()` call in the `main()` method and produce the same output at runtime?

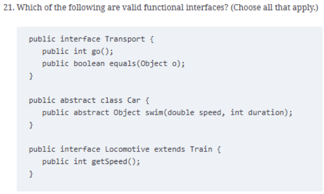
```
import java.util.List;
interface Yawn {
   String yawn(double d, List<Integer> time);
}
class Sloth implements Yawn {
   public String yawn(double zzz, List<Integer> time) {
      return "Sleep: " + zzz;
   } }
public class Vet {
   public static String takeNap(Yawn y) {
      return y.yawn(10, null);
   }
   public static void main(String... unused) {
      System.out.print(takeNap(new Sloth()));
   } }
```

A. `(z,f) -> { String x = ""; return "Sleep: " + x }`
B. `(t,s) -> { String t = ""; return "Sleep: " + t; }`
C. `(w,q) -> {"Sleep: " + w}`
D. `(e,u) -> { String g = ""; "Sleep: " + e }`
E. `(a,b) -> "Sleep: " + (double)(b==null ? a : a)`
F. `(r,k) -> { String g = ""; return "Sleep:"; }`
G. None of the above, as the program does not compile



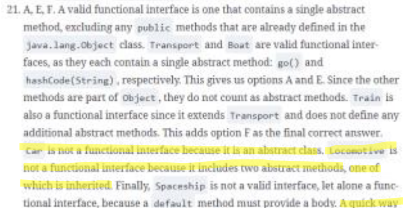
20. E. Option A does not compile because the second statement within the block is missing a semicolon (`;`) at the end. Option B is an invalid lambda expression because `t` is defined twice: in the parameter list and within the lambda expression. Options C and D are both missing a `return` statement and semicolon. Options E and F are both valid lambda expressions, although only option E matches the behavior of the `Sloth` class. In particular, option F only prints `Sleep:`, not `Sleep: 10.0`.

21e,~~d~~,f A

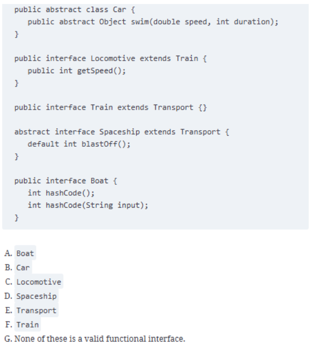


21. Which of the following are valid functional interfaces? (Choose all that apply.)

```
public interface Transport {
   public int go();
   public boolean equals(Object o);
}

public abstract class Car {
   public abstract Object swim(double speed, int duration);
}

public interface Locomotive extends Train {
   public int getSpeed();
}
```

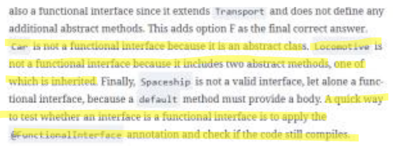


21. A, E, F. A valid functional interface is one that contains a single abstract method, excluding any `public` methods that are already defined in the `java.lang.Object` class. `Transport` and `Boat` are valid functional interfaces, as they each contain a single abstract method: `go()` and `hashCode(String)`, respectively. This gives us options A and E. Since the other methods are part of `Object`, they do not count as abstract methods. `Train` is also a functional interface since it extends `Transport` and does not define any additional abstract methods. This adds option F as the final correct answer. `Car` is not a functional interface because it is an abstract class. `Locomotive` is not a functional interface because it includes two abstract methods, one of which is inherited. Finally, `Spaceship` is not a valid interface, let alone a functional interface, because a `default` method must provide a body. A quick way



```
public abstract class Car {
   public abstract Object swim(double speed, int duration);
}

public interface Locomotive extends Train {
   public int getSpeed();
}

public interface Train extends Transport {}

abstract interface Spaceship extends Transport {
   default int blastOff();
}

public interface Boat {
   int hashCode();
   int hashCode(String input);
}
```

A. `Boat`
B. `Car`
C. `Locomotive`
D. `Spaceship`
E. `Transport`
F. `Train`
G. None of these is a valid functional interface.



also a functional interface since it extends `Transport` and does not define any additional abstract methods. This adds option F as the final correct answer. `Car` is not a functional interface because it is an abstract class. `Locomotive` is not a functional interface because it includes two abstract methods, one of which is inherited. Finally, `Spaceship` is not a valid interface, let alone a functional interface, because a `default` method must provide a body. A quick way to test whether an interface is a functional interface is to apply the `@FunctionalInterface` annotation and check if the code still compiles.