1. will fail to compile with the error message: local variables referenced from a lambda expression must be final or effectively final.

String name = getUserName();

name = formatUserName(name);

button.addActionListener(event -> System.out.println("hi " + name));

1. Operator '&#x002B;' cannot be applied to java.lang.Object, java.lang.Object.

BinaryOperator add = (x, y) -> x + y;

1. Is it thread-safe?

@NotThreadSafe

public class ListHelper<E> {

public List<E> list =

Collections.synchronizedList(new ArrayList<E>());

...

public synchronized boolean putIfAbsent(E x) {

boolean absent = !list.contains(x);

if (absent)

list.add(x);

return absent;

}

}

@ThreadSafe

public class ListHelper<E> {

public List<E> list =

Collections.synchronizedList(new ArrayList<E>());

...

public boolean putIfAbsent(E x) {

synchronized (list) {

boolean absent = !list.contains(x);

if (absent)

list.add(x);

return absent;

}

}

}

**Example 3-5. Not printing out artist names due to lazy evaluation**

allArtists.stream()

.filter(artist -> {

System.out.println(artist.getName());

return artist.isFrom("London");

});

**Example 3-6. Printing out artist names**

long count = allArtists.stream()

.filter(artist -> {

System.out.println(artist.getName());

return artist.isFrom("London");

})

.count();

Finding the shortest track with streams

List<Track> tracks = asList(new Track("Bakai", 524),

new Track("Violets for Your Furs", 378),

new Track("Time Was", 451));

Track shortestTrack = tracks.stream()

.min(Comparator.comparing(track -> track.getLength()))

.get();

assertEquals(tracks.get(1), shortestTrack);

ConcurrentModificationException can arise in single-threaded code as well; this happens when

objects are removed from the collection directly rather than through Iterator.remove.

An example of legacy collections code that uses loops to perform a task and refactor it into a stream-based implementation.

*Legacy code finding names of tracks over a minute in length*

**public** Set<String> findLongTracks(List<Album> albums) {

Set<String> trackNames = **new** HashSet<>();

**for**(Album album : albums) {

**for** (Track track : album.getTrackList()) {

**if** (track.getLength() > 60) {

String name = track.getName();

trackNames.add(name);

}

}

}

**return** trackNames;

}

*Refactor*

**public** Set<String> findLongTracks(List<Album> albums) {

**return** albums.stream()

.flatMap(album -> album.getTracks())

.filter(track -> track.getLength() > 60)

.map(track -> track.getName())

.collect(toSet());

}

*Example 4-5. A method that could be dispatched to one of two methods*

overloadedMethod("abc");

*Example 4-6. Two methods that are overloaded*

**private void** overloadedMethod(Object o) {

System.out.print("Object");

}

**private void** overloadedMethod(String s) {

System.out.print("String");

}

*Example 4-7. Another overloaded method call*

overloadedMethod((x, y) -> x + y);

**private interface IntegerBiFunction extends** BinaryOperator<Integer> {

}

**private void** overloadedMethod(BinaryOperator<Integer> lambda) {

System.out.print("BinaryOperator");

}

**private void** overloadedMethod(IntegerBiFunction lambda) {

System.out.print("IntegerBinaryOperator");

}

*Example 4-9. A compile failure due to overloaded methods*

overloadedMethod((x) -> **true**);

**private interface IntPredicate** {

**public boolean** test(**int** value);

}

**private void** overloadedMethod(Predicate<Integer> predicate) {

System.out.print("Predicate");

}

**private void** overloadedMethod(IntPredicate predicate) {

System.out.print("IntPredicate");

}

In this case, javac will fail to compile the example,

complaining that the lambda expression is an ambiguous method call: IntPredicate

doesn’t extend any Predicate, so the compiler isn’t able to infer that it’s more specific.

**public interface Jukebox** {

**public default** String rock() {

**return** "... all over the world!";

}

}

**public interface Carriage** {

**public default** String rock() {

**return** "... from side to side";

}

}

**public class MusicalCarriage implements** Carriage, Jukebox {

}

**public class MusicalCarriage**

**implements** Carriage, Jukebox {

@Override

**public** String rock() {

**return** Carriage.super.rock();

}

}

Take a look at the Artists domain class in Example 4-26, which represents a group

of artists. Your assignment is to refactor the getArtist method in order to return

an Optional<Artist>. It contains an element if the index is within range and is an

empty Optional otherwise. Remember that you also need to refactor the getAr

tistName method, and it should retain the same behavior.

*Example 4-26. The Artists domain class, which represents more than one Artist*

**public class Artists** {

**private** List<Artist> artists;

**public** Artists(List<Artist> artists) {

**this**.artists = artists;

}

**public** Artist getArtist(**int** index) {

**if** (index < 0 || index >= artists.size()) {

indexException(index);

}

**return** artists.get(index);

}

**private void** indexException(**int** index) {

**throw new** IllegalArgumentException(index +

" doesn't correspond to an Artist");

}

**public** String getArtistName(**int** index) {

**try** {

Artist artist = getArtist(index);

**return** artist.getName();

} **catch** (IllegalArgumentException e) {

**return** "unknown";

}

}

}

**Basic Question for Beginners**

* Convert a list of strings into their uppercase equivalents.

public static List<String> allToUpperCase(List<String> words) {

return words.stream()

.map(string -> string.toUpperCase())

.collect(Collectors.<String>toList());

}

@Test

**public void** multipleWordsToUppercase() {

List<String> input = Arrays.asList("a", "b", "hello");

List<String> result = Testing.allToUpperCase(input);

assertEquals(asList("A", "B", "HELLO"), result);

}