Started on	Tuesday, 25 May 2021, 12:41
State	Finished
Completed on	Tuesday, 25 May 2021, 12:49
Time taken	7 mins 37 secs
Grade	**75.00** out of 140.00 (**54**%)

Question 1

Correct

Mark 10.00 out of 10.00

Here is the code again from the slide Autoboxing and Equality:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
a.put("c", 130); // put ints into the map
b.put("c", 130);
```

What is the compile-time type of the expression `130`?

After executing `a.put("c", 130)`, what is the runtime type that is used to represent the value `130` in the map?

What is the compile-time type of `a.get("c")`?

Select one:

[✓ a. int, Integer, Integer]

b. int, Integer, int

c. Integer, int, int

d. int, int, Integer

e. Integer, int, Integer

f. Integer, Integer, int

Your answer is correct.

30 is an integer literal, so its compile-time type is `int`.

In the `Map<String, Integer>`, the keys are `String`s and the values are `Integer`s. So when 130 is placed in the map, it is automatically *boxed* up into a fresh `Integer` object.

The `get()` operation for a `Map<K, V>` returns values of type `V`, so for a `Map<String, Integer>`, the type would be `Integer`.

The correct answer is: int, Integer, Integer

Question 2

Correct

Mark 10.00 out of 10.00

Here is the code again from the slide Autoboxing and Equality:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
a.put("c", 130); // put ints into the map
b.put("c", 130);
```

After this code executes, what would `a.get("c").equals(b.get("c"))` return?

What would `a.get("c") == b.get("c")` return?

Select one:

[✓ a. true, false]

b. false, true

c. true, true

d. false, false

Your answer is correct.

Both `get()` calls return an `Integer` object representing 130. Since `equals()` is correctly implemented for the (immutable) `Integer` type, it returns `true` for those two values.

The `get()` calls return *distinct* `Integer` objects, so they are not referentially equal. `==` returns `false`.

This is the surprising pitfall: if you have in your mind that the `Map` contains `int` values, you will be surprised by the behavior of `get()`, because it returns an `Integer` instead. Most of the time you can use `Integer` interchangeably with `int`, but not when it comes to equality operators like `==` and `equals`.

The correct answer is: true, false

Question 3

Correct Mark 10.00 out of 10.00

Here is the code again from the slide Autoboxing and Equality:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
a.put("c", 130); // put ints into the map
b.put("c", 130);
```

Now suppose you assign the `get()` results to `int` variables:

```
int i = a.get("c");
int j = b.get("c");
boolean isEqual = (i == j);
```

Is there an error with that code, or if not, what is the value of `isEqual`?

Select one:

[✓ a. true]

b. false

c. compile error

d. runtime error

Your answer is correct.

The assignments automatically *unbox* the `Integer` objects into `int` values, both 130. Those primitive `int` values are both 130, so `==` now returns `true`.

Behavior differences like this make autoboxing/unboxing bugs hard to spot and easy to introduce. Another reason they can be tricky: if we asked these same questions with 127 instead of 130, the answers would be different! For the integers from -128 to 127, the boxed `Integer` objects come from a pool **that is reused every time, and the objects will be ==.**

The correct answer is: true

Question 4

Partially correct

Mark 5.00 out of 10.00

For this code that starts a thread:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello! It's me, a thread!");
    }
}).start();
```

Put the following events in the order that they occur.

1. start() is called ✗
2. a Thread object is created ✓
3. run() is called ✗
4. a Runnable object is created ✗
5. "Hello! It's me, a thread!" is printed ✓
6. run() returns ✓

Your answer is partially correct.

The expression `new Runnable() { ... }` creates a new object that implements `Runnable`, which will be passed as a parameter to `new Thread()`. Note especially that the code inside the anonymous class *is not executed yet*. It won't be executed until its `run()` method is called.

Once we have the `Runnable` object, the next thing that happens is the call to `new Thread()`, which creates a new `Thread` object.

Then `start()` is called on that new `Thread` object.

The thread then starts, and `Thread.start()` calls `run()` on the `Runnable` object.

Inside the body of `run()`, the `println` statement executes.

Finally, the `run()` method returns, and the thread finishes.

You have correctly selected 3.
The correct answer is:
For this code that starts a thread:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello! It's me, a thread!");
    }
}).start();
```

Put the following events in the order that they occur.

1. [a Runnable object is created]
2. [a Thread object is created]
3. [start() is called]
4. [run() is called]
5. ["Hello! It's me, a thread!" is printed]
6. [run() returns]

Question 5

Incorrect

Mark 0.00 out of 10.00

When you run a Java program (for example, using the Run button in IntelliJ), how many processes and threads are created at first?

Select one:

- a. one process and one thread
- b. one process and zero thread
- c. zero process and one thread
- d. one process for each class, and one thread for each class in the program
- [✓ e. one process, and one thread for each class in the program]
- f. one process for each class in the program, and one thread
- g. zero process and zero thread

Your answer is incorrect.

The correct answer is: one process and one thread

Question 6

Incorrect

Mark 0.00 out of 10.00

Suppose we run main in this program, which contains bugs:

```
public class Moirai {
    public static void main(String[] args) {
        Thread clotho = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        clotho.start();
        new Thread(new Runnable() {
            public void run() { System.out.println("measuring"); };
        }).start();
        new Thread(new Runnable() {
            public void run() { System.out.println("cutting"); };
        });
    }
}
```

How many new Thread objects are created?

Select one:

- a. 3
- [✓ b. 2]
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6

Your answer is incorrect.

One is assigned to variable `clotho`. The other two are not assigned to a variable.

The correct answer is: 3

Question 7

Correct

Mark 10.00 out of 10.00

Suppose we run main in this program, which contains bugs:

```
public class Moirai {
    public static void main(String[] args) {
        Thread clotho = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        clotho.start();
        new Thread(new Runnable() {
            public void run() { System.out.println("measuring"); };
        }).start();
        new Thread(new Runnable() {
            public void run() { System.out.println("cutting"); };
        });
    }
}
```

How many new threads are run?

Select one:

- a. 3
- [✓ b. 2]
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6

Your answer is correct.

The code calls `start` on the first two threads. But the third thread is never `start`ed, so it will not run.

The correct answer is: 2

Question 8

Incorrect

Mark 0.00 out of 10.00

Suppose we run main in this program, which contains bugs:

```
public class Moirai {
    public static void main(String[] args) {
        Thread clotho = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        clotho.start();
        new Thread(new Runnable() {
            public void run() { System.out.println("measuring"); };
        }).start();
        new Thread(new Runnable() {
            public void run() { System.out.println("cutting"); };
        });
    }
}
```

What is the maximum number of threads that might be running at the same time?

Select one:

- a. 3
- [✓ b. 2]
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6

Your answer is incorrect.

The initial thread running `main` plus the two new threads that were started. The reason we have to say “might” here is because different interleaving may mean that we don’t always reach this maximum; for example, the first new thread might finish running before the second one even starts.

The correct answer is: 3

Question 9

Incorrect

Mark 0.00 out of 10.00

Suppose we run main in this program, which demonstrates two common bugs:

```
public class Parcae {
    public static void main(String[] args) {
        Thread nona = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        nona.run();
        Runnable decima = new Runnable() {
            public void run() { System.out.println("measuring"); };
        };
        decima.run();
        // ...
    }
}
```

How many new Thread objects are created (not counting the main thread) ?

Select one:

- a. 3
- [✓ b. 2]
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6

Your answer is incorrect.

We create only one `Thread`, assigned to variable `nona`.

The correct answer is: 1

Question 10

Incorrect

Mark 0.00 out of 10.00

Suppose we run main in this program, which demonstrates two common bugs:

```
public class Parcae {
    public static void main(String[] args) {
        Thread nona = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        nona.run();
        Runnable decima = new Runnable() {
            public void run() { System.out.println("measuring"); };
        };
        decima.run();
        // ...
    }
}
```

How many new threads are run?

Select one:

- a. 3
- b. 2
- [✓ c. 1]
- d. 0
- e. 4
- f. 5
- g. 6

Your answer is incorrect.

The line `nona.run()` is a bug: calling `Thread.run()` does not run the code in a new concurrent thread. It uses the same thread, the initial thread running `main`.

And we call `run()` on `Runnable decima`, which also uses the same thread. Perhaps the author meant to create a new `Thread` with that `Runnable` instead of running it directly.

Never call `run()` on a `Thread`, or on a `Runnable` that you created for a thread. Instead, always make a `new Thread()` with an instance of your `Runnable`, and call `start()` on the thread to start it. `Thread` will take care of calling `run()` on your `Runnable` from the new thread.

The correct answer is: 0

Question 11

Partially correct

Mark 2.50 out of 10.00

Suppose we run main in this program, which contains bugs:

```
public class Moirai {
    public static void main(String[] args) {
        Thread clotho = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        clotho.start();
        new Thread(new Runnable() {
            public void run() { System.out.println("measuring"); };
        }).start();
        new Thread(new Runnable() {
            public void run() { System.out.println("cutting"); };
        });
    }
}
```

Which of the following is a possible output from this program?

Select one or more:

- [ ] i. measuring
- [ ] ii. measuring
  spinning
- [ ] iii. spinning
- [✓] iv. spinning
  measuring
  cutting
- [ ] v. cutting
- [ ] vi. spinning
  cutting
- [✓] vii. spinning
  measuring

Your answer is partially correct.

The third thread is never started, so `cutting` will never be printed.

The order of the other outputs depends on whether the first thread runs `println` before or after the second.

Note that `main()` may very well return while the two threads it created are still running. This ends the main thread of the program, but it does not stop the entire process. In Java, the process continues running until all running threads have exited, unless `System.exit()` is called to force the proces to exit.

You have correctly selected 1.
The correct answers are: measuring
spinning, spinning
measuring

Question 12

Correct Mark 10.00 out of 10.00

Suppose we run main in this program, which demonstrates two common bugs:

```
public class Parcae {
    public static void main(String[] args) {
        Thread nona = new Thread(new Runnable() {
            public void run() { System.out.println("spinning"); };
        });
        nona.run();
        Runnable decima = new Runnable() {
            public void run() { System.out.println("measuring"); };
        };
        decima.run();
        // ...
    }
}
```

Which of the following is a possible output from this program?

Select one or more:

- [ ] i. measuring

- [ ] ii. spinning
- [ ] iii. measuring
  spinning
- [✓] iv. spinning
  measuring

Your answer is correct.

There is only one thread running in this program, and only one possible output. Both `nona.run()` and `decima.run()` run their code in the current thread, the initial thread running `main`.

The correct answer is: spinning
measuring

Question 13

Correct

Mark 10.00 out of 10.00

Consider the following code:

```
private static int x = 1;

public static void methodA() {
  x *= 2;
  x *= 3;
}

public static void methodB() {
  x *= 5;
}
```

Suppose methodA and methodB run **sequentially**, i.e. first one and then the other.

What is the final value of x?

Select one:

- a. 1
- b. 2
- c. 5
- [✓ d. 30]
- e. 6
- f. 10
- g. 150

Your answer is correct.

If `methodA` runs first, then it sets `x` to 1×2×3 = 6, and then `methodB` runs and sets `x` to 6×5 = 30. Since multiplication is commutative, we get the same result if `methodB` runs before `methodA`.

The correct answer is: 30

Question 14

Partially correct

Mark 7.50 out of 10.00

Consider the following code:

```
private static in

public static voi
  x *= 2;
  x *= 3;
}

public static void methodB() {
  x *= 5;
}
```

Now suppose methodA and methodB run **concurrently**, so that their instructions might **interleave** arbitrarily, exhibiting **a race condition**.

Which of the following are ***all*** possible final values of x?

Select one or more:

- i. 1
- ii. 2
- [✓ iii. 5]
- [✓ iv. 6]
- v. 10
- [✓ vi. 30]
- vii. 150

Your answer is partially correct.

30 is still a possible final value, because `methodA` might manage to run completely before `methodB` even starts. So the sequential execution (one before the other) is still always possible when two modules run concurrently. But it's not the only possible result.

One way to get 5 as the final result is if `methodB` first reads the value of `x` (1), and then `methodA` interleaves, running completely (setting `x` to 6). Then `methodB` finishes its computation using the original value of `x` that it read (1), setting `x` to 5 and overwriting `methodA`'s effect.

A similar argument allows us to get 6 as the final result: `methodA` first reads the value of `x`, then `methodB` runs completely, and then `methodA` computes `x =` 6 and overwrites what `methodB` did.

To get 10 as the final result, `methodA` might first successfully multiply by 2 and store that back to `x`. `methodB` then starts, reading the value of x (now 2) and starting its multiplication. Then `methodA` runs again, multiplying `x` by 3 and storing it back (6). Finally `methodB` finishes, using a stale value of `x`, and storing back 2×5 = 10, overwriting the result of `methodA`'s second multiplication.

We can't get 1 as a result, because `methodA` and `methodB` will compute *some* new value of `x` and store it back.

We also can't get 2 as a result, because the final value of `x` will come from either `methodA` or `methodB`'s last computation, which is either a multiplication by 3 or a multiplication by 5.

We also can't get 150, because none of the multiplications will produce a value that large.

You have correctly selected 3.
The correct answers are: 5, 6, 10, 30

Finish review

◂ Lab 12 Recording

Jump to...

Lab Exercise 12.1 HASet SIZEC