
Java

The Java programming language is a common choice for developing cross-platform applications and systems. Additionally, it has attracted more attention because of the popularity of Android mobile phones in recent years. Therefore, many companies have requirements on their candidates' Java proficiency level.

Java Keywords

Similar to C++ and C#, there are many interview questions on Java keywords or concepts. One of the most frequently met questions is: What are the uses of `finalize`, `finally`, and `final` in Java? The `finalize` method is related to the garbage collector in Java. Java is a managed programming language, and its runtime periodically reclaims memory occupied by objects that are not referenced by others. If an object is holding some resources besides memory, such as files and network connections, we might want to make sure these resources are released before the object is destroyed. Java provides a mechanism called finalization to handle such situations. We define specific actions in the method of `finalize` that will occur when an object is about to be reclaimed by the garbage collector.

The keyword `finally` is related to the exception handling mechanism. The `finally` block always executes when the corresponding `try` block exists. This ensures that the `finally` block is executed even if an unexpected exception occurs. Usually, the `finally` block contains code to clean up resources.

The keyword `final` in the Java programming language is used in several different contexts to define an entity that cannot be modified later:

- No classes can derive from `final` classes.
- A `final` method cannot be overridden by subclasses.
- A `final` variable can only be initialized once, either via an assignment statement at the point of declaration or a constructor. Some `final` variables, named blank `final` variables, are not initialized at the point of declaration. A blank `final` variable of a class must be definitely assigned in every constructor of the class in which it is declared. Additionally, `final` arguments, which are declared in argument lists of methods, are similar to `final` variables.

The use for `final` variables gets more complex when they are references. If a `final` variable is a reference, it cannot be rebound to reference another object. However, the object it references is still mutable.

The piece of code in Listing 2-18 uses `final` variables. Which methods will cause compile-time errors?

Listing 2-18. *Java Code to Modify Final Fields*

```
public class WithFinal {
public final int number = 0;
public final int[] array;

public WithFinal() {
    array = new int[10];
    for(int i = 0; i < array.length; ++i)
        array[i] = i;
}

public void ModifyFinal1() {
    ++number;
}

public void ModifyFinal2() {
    for(int i = 0; i < array.length; ++i)
        array[i] += i;
}

public void ModifyFinal3() {
    array = new int[10];
       for(int i = 0; i < array.length; ++i)
```

```
        array[i] = i;
    }
}
```

In the class `WithFinal`, there are two data fields. The field `number` is a primitive while the field `array` is a reference and it is also a blank `final` variable.

Blank `final` variables can be initialized in constructors, so the constructor method in the class `WithFinal` has no problems. It tries to modify the constant primitive `number` inside the method `ModifyFinal1`, which is not allowed by the Java compiler. The method `ModifyFinal2` modifies the object referenced by `array`, but it does not modify the reference itself. It is allowed to do so. Inside the method `ModifyFinal3`, a new array is created and it is rebound to the constant reference `array`, so it raises compiling errors.

Data Containers

The Java programming language provides many useful data containers. Data containers can be divided into two categories: one is collection, such as `LinkedList` and `Stack`; and the other is map, such as the type `HashMap`. Both are commonly used in practical development and also frequently met in interviews. For example, the following is an interview question about `HashMap`: What is the output of the piece of code in Listing 2-19?

Listing 2-19. *Java Code with* `HashMap`

```
public class MyString {
    public MyString(String data) {
        this.data = data;
    }

    private String data;
}

public static void main(String args[]) {
    Map<String, Integer> map1 = new HashMap<String, Integer>();
    String str1 = new String("Hello World.");
    String str2 = new String("Hello World.");
    map1.put(str1, new Integer(10));
    map1.put(str2, new Integer(20));

    Map<MyString, Integer> map2 = new HashMap<MyString, Integer>();
    MyString str3 = new MyString(str1);
    MyString str4 = new MyString(str2);
    map2.put(str3, new Integer(10));
    map2.put(str4, new Integer(20));

    System.out.println(map1.get(str1));
    System.out.println(map2.get(str3));
}
```

Java checks the existence of a key in a `HashMap` via its hash code, which is returned by the method `hashCode`. The method `hashCode` is defined in the class `Object`, and it can be overridden by subclasses. When two keys have the same hash code, it calls the method `equals` to check their equality. Similar to `hashCode`, `equals` is also defined in the class `Object` and can be overridden by subclasses.

The type of key in the `map1` is `String`, which overrides the method `hashCode` and `equals`. The method `String.hashCode` returns the same hash code when two instances have the same string content. Because the contents in `str1` and `str2` are the same, they share the same hash code. When it tries to put the record with key `str2` to `map1`, a key `str1` exists already with the same hash code. The method `equals` also shows that these two keys are equal to each other because their contents are the same. Therefore, it just updates the corresponding value of the key `str1` to 20, instead of inserting a new record.

The type of key in `map2` is `MyString`, which does not override the methods `hashCode` and `equals`. It has to call the method `Object.hashCode` to compare hash codes of keys, which returns the object addresses. The keys `str3` and `str4` have different hash codes because they are two different objects and have different addresses. A new record with key `str4` is inserted into the `map2`, and the value responding to the key `str3` remains 10.

Therefore, the output of the code above contains two lines: The first line is a number 20, and the second one is a number 10.

Java has good support for threads and synchronization. There are many interesting interview problems related to multithreading programming, and the following one is an example.

Thread Scheduler

Question 4 There are three threads in a process. The first thread prints 1 1 1 …, the second one prints 2 2 2 …, and the third one prints 3 3 3 … endlessly. How do you schedule these three threads in order to print 1 2 3 1 2 3 …?

Java defines methods `wait`, `notify`, and `notifyAll` in the base class `Object`. The method `wait` is used when a thread is waiting for some condition that is typically controlled by another thread. It allows us to put a thread to sleep while waiting for the condition to change, and the thread will be wakened up when a `notify` or `notifyAll` occurs. Therefore, `wait` provides a method to synchronize activities between threads, and it is applicable to solve this problem.

A solution based on methods `wait` and `notify` is found in Listing 2-19.

Listing 2-19. *Java code to schedule threads*

```
   public class SimpleThread extends Thread {

    private int value;

    public SimpleThread(int num) {

        this.value = num;


            start();

    }


    public void run() {

        while(true) {

            synchronized(this) {

                try {

                    wait();

                } catch (InterruptedException e) {

                    throw new RuntimeException(e);

                }


                System.out.print(value + " ");

            }

        }

    }

}


public class Scheduler {

    static final int COUNT = 3;

    static final int SLEEP = 37;


    public static void main(String args[]) {

        SimpleThread threads[] = new SimpleThread[COUNT];

        for(int i = 0; i < COUNT; ++i)

            threads[i] = new SimpleThread(i + 1);


        int index = 0;

        while(true){

            synchronized(threads[index]) {

                threads[index].notify();

            }


            try {

                Thread.sleep(SLEEP);

            } catch (InterruptedException e) {

                throw new RuntimeException(e);

            }


            index = (++index) % COUNT;
```

```
        }
    }
}
```

There are four threads in the code above. The first is the main thread in the Java application, which acts as the scheduler, and it creates three printing threads and stores them into an array. The main thread awakens threads one by one according to their index in the array via the method `notify`. Once a thread wakes up, it prints a number and then sleeps again to wait for another notification.

Source Code:

004_Scheduler.java