CHAPTER 2

Programming Languages

There are a number of popular programming languages these days. Different companies, even different departments within the same company, prefer to use different languages. Driver developers get used to writing code in C. Many programmers who work on Linux develop applications in C++, while other programmers working on Windows prefer C#. Many cross-platform systems are developed in Java. Objective C becomes more and more popular due to sales of iPads and iPhones. Additionally, scripting languages, such as Perl and Python, are very suitable to toolkit development.

Massive books have been written about each language. It is not a goal to cover all languages in detail in this book because of the space limitation, so Coding Interviews only discusses interview questions for the four most popular programming languages: C, C++, C#, and Java[1].

C

One of the reasons why the C programming language is so popular is the flexibility of its use for memory management. Programmers have opportunities to control how, when, and where to allocate and deallocate memory. Memory is allocated statically, automatically, or dynamically in the C programming language.

Static variables and global variables are static-duration variables, which are allocated along with the executable code of the program and persist during the execution of the application. When a local variable is declared as static inside a function, it is initialized only once. Whatever values the function puts into its static local variables during one call will be present when the function is called again. The following is a typical interview question about static local variables: What is the output when the function `test` executes? (See Listing 2-1).

Listing 2-1. *C Code with Static Variables*

```
int hasStatic(int n) {
    static int x = 0;
    x += n;
    return x;
```

[1] The TIOBE company publishes the monthly *TIOBE Programming Community Index,* which is an authoritative indicator of the popularity of programming languages. Please refer to the web site `www.tiobe.com/index.php/content/paperinfo/tpci/index.html` for the latest ranking.

```
}

void test() {
    int sum = 0;
    for(int i = 1; i <= 4; ++i)
        sum += hasStatic(i);

    printf("Result of sum is %d.\n", sum);
}
```

The variable `x` in the function `hasStatic` is declared as `static`. When the function is called the first time with a parameter 1, the static variable `x` is initialized as 0, and then set to 1. When the function is called the second time with a parameter 2, it is not initialized again and its value 1 persists. It becomes 3 after execution. Similarly, its value becomes 6 and 10 after execution with parameters 3 and 4. Therefore, the result is the sum 1+2+6+10=19.

Automatic-duration variables, also called local variables, are allocated on the stack. They are allocated and deallocated automatically when the program execution flow enters and leaves their scopes. In the sample code in Listing 2-2, there is a local array `str` in the function `allocateMemory`. What is the problem with it?

Listing 2-2. *C Code to Allocate Memory*

```
char* allocateMemory() {
    char str[20] = "Hello world.";
    return str;
}

void test() {
    char* pString = allocateMemory();
    printf("pString is %s.\n", pString);
}
```

Because the array `str` is a local variable, it is deallocated automatically when the execution flow leaves the function `allocateMemory`. Therefore, the returned memory to `pString` in the function `test` is actually invalid, and its content is somewhat random.

One limitation of static-duration and automatic-duration variables is that their size of allocation is required to be compile-time constant. This limitation can be avoided by dynamic memory allocation in which memory is more explicitly, as well as more flexibly, managed. In the C programming language, the library function `malloc` is used to allocate a block of memory dynamically on the heap, where size is specified at runtime. This block of memory is accessible via a pointer that `malloc` returns. When the memory is no longer needed, the pointer should be passed to another library function, `free`, which deallocates the memory in order to avoid memory leaks.

The function `allocateMemory` in Listing 2-3 utilizes the function `malloc` to allocate memory dynamically. What is the problem in it?

Listing 2-3. *C Code to Allocate Memory*

```
void allocateMemory(char* pString, int length) {
    pString = (char*)malloc(length);
}

void test() {
    char* pString = NULL;
    allocateMemory(pString, 20);
    strcpy(pString, "Hello world.");
}
```

The parameter `pString` in the function `allocateMemory` is a pointer, and the string content it points to can be modified. However, the memory address it owns cannot be changed when the function `allocateMemory` returns. Since the input parameter `pString` of `allocateMemory` in the function `test` is `NULL`, it remains `NULL` after the execution of `allocateMemory`, and it causes the program to crash when the `NULL` address is accessed in `test`. In order to fix this problem, the first parameter of `allocateMemory` should be modified as `char** pString`.

Macros are commonly used in the C programming language and also commonly asked about in technical interviews. Since it is tricky and error-prone to use macros, unexpected results are gotten when they are misused. For example, what is the output when the function `test` in Listing 2-4 gets executed?

Listing 2-4. *C Code with Macros*

```
#define SQUARE(x) (x*x)

void test() {
    printf("Square of (3+4) is %d.\n", SQUARE(3+4));
}
```

The result of 3+4 is 7, and the square of 7 is 49. However, 49 is not the result of `SQUARE(3+4)`. Macros are substituted by the preprocessor, so `SQUARE(3+4)` becomes 3+4*3+4, which is actually 19. In order to make the result as expected, the macro should be modified to `#define SQUARE(x) ((x)*(x))`.

Even in the revised version of macro `SQUARE`, its result might look surprising if it is not analyzed carefully in some cases. For instance, what are the results of `x` and `y` after the code in Listing 2-5 is executed?

Listing 2-5. *C Code with Macros*

```
#define SQUARE(x) ((x)*(x))

void test() {
    int x = 5;
    int y = SQUARE(x++);
    printf("Result of x is %d, y is %d.\n", x, y);
}
```

It looks like that the variable `x` increases only once in the code above. However, when the macro is substituted, the variable `y` is assigned as `((x++)*(x++))`. The result of `y` is 25, and `x` becomes 7 because it increases twice.

Many coding interview questions can be solved in the C programming language. A question in the following subsection is an example, and more questions are discussed in later chapters.

Palindrome Numbers

■ **Question 1** Please implement a function that checks whether a positive number is a palindrome or not. For example, 121 is a palindrome, but 123 is not.

Converting a Number into a String

It is easy to check whether a string is a palindrome or not: We can check whether the first character and the last one are identical, and then compare the second character and the second one from the end, and so on. If the converted string is a palindrome, the original should also be a palindrome.

This solution can be implemented with the code in Listing 2-6, which converts a number into a string with the library function `sprintf`.

Listing 2-6. *C Code to Verfiy Palindrome Numbers (Version 1)*

```
/* It returns 1 when number is palindrome, otherwise returns 0. */
#define NUMBER_LENGTH 20
int IsPalindrome_solution1(unsigned int number) {
    char string[NUMBER_LENGTH];
    sprintf(string, "%d", number);

    return IsPalindrome(string);
}

int IsPalindrome(const char* const string) {
    int palindrome = 1;
    if(string != NULL) {
        int length = strlen(string);
        int half = length >> 1;

        int i;
        for(i = 0; i < half; ++ i) {
            if(string[i] != string[length - 1 - i]) {
                palindrome = 0;
                break;
            }
        }
```

```
    }

    return palindrome;
}
```

Usually, this solution is not the one expected by interviewers. One reason is that while it is intuitive, interviewers expect something innovative, and another reason is that it requires auxiliary memory to store the converted string.

Composing a Reversed Number

As we know, it is easy to get digits from right to left via the / and % operators. For example, digits in the number 123 from right to left are 3, 2, and 1. A reversed number 321 is constructed with these three digits. Let's check whether the reversed number is identical to the original one. If it is, the original number is a palindrome.

The implementation is shown in Listing 2-7.

Listing 2-7. *C Code to Verfiy Palindrome Numbers (Version 2)*

```
/* It returns 1 when number is palindrome, otherwise returns 0. */
int IsPalindrome_solution2(unsigned int number) {
    int reversed = 0;
    int copy = number;

    while(number != 0) {
        reversed = reversed * 10 + number % 10;
        number /= 10;
    }

    return (reversed == copy) ? 1 : 0;
}
```

Source Code:

```
001_PalindromeNumber.c
```

Test Cases:

- Palindrome numbers with odd length
- Palindrome numbers with even length
- Non-palindrome numbers

C++

C++ might be the most difficult programming language in many software engineers' judgment. Many interviewers believe a candidate who is proficient on C++ has the abilities needed to master other technical skills, so questions about C++ are quite popular during interviews.

Interview questions about the C++ programming language can be divided into three categories: C++ concepts, tricky C++ coding problems, and implementing a class or member function.

C++ Concepts

Questions in the first category are about C++ concepts, especially about C++ keywords. For example, what are the four keywords for type casting, and under what are scenarios would you use each of them?

The most popular questions in this category concern the keyword `sizeof`.

For instance, the following dialog is repeated again and again in many interviews:

Interviewer: There is an empty class, without any member fields and functions inside its definition. What is the result of `sizeof` for this class?

Candidate: It is 1.

Interviewer: Why not 0?

Candidate: The size seems to be 0 since the class is empty without any data. However, it should occupy some memory; otherwise, it cannot be accessed. The size of memory is decided by compilers. It occupies 1 byte in Visual Studio.

Interviewer: What is the result of `sizeof` if we add the constructor and destructor functions in the class?

Candidate: It is also 1. The addresses for functions are irrelevant to instances, and compilers do not add any data in instances of this class.

Interviewer: How about declaring the destructor function as a virtual function?

Candidate: When a C++ compiler sees a virtual function inside a class, it creates a virtual function table for the class and adds a pointer to the table in each instance. A pointer in a 32-bit machine occupies 4 bytes, so the result of `sizeof` is 4. The result on a 64-bit machine is 8 because a pointer occupies 8 bytes there.

Analyzing Execution of C++ Code

The second type of interview questions in C++ concerns analyzing execution results of some sample code. Candidates without deep understanding of C++ are prone to make mistakes because the code to be analyzed is usually quite tricky.

For example, an interviewer hands a piece of paper printed with the code in Listing 2-8 to a candidate and asks the candidate what the result is if we try to compile and execute the code: (A) Compiling error; (B) It compiles well, but crashes in execution; or (C) It compiles well, and executes smoothly with an output of 10.

Listing 2-8. *C++ Code about Copy Constructor*

```
class A {
private:
    int value;
```

```
public:
    A(int n) {
        value = n;
    }
    A(A other) {
        value = other.value;
    }
    void Print() {
         std::cout << value << std::endl;
    }
};

int main(int argc, _TCHAR* argv[]) {
    A a = 10;
    A b = a;
    b.Print();

    return 0;
}
```

The parameter in the copy constructor `A(A other)` is an instance of type `A`. When the copy constructor is executed, it calls the copy constructor itself because of the pass-by-value parameter. Since endless recursive calls cause call stack overflow, a pass-by-value parameter is not allowed in the C++ standard, and both Visual Studio and GCC report an error during compiling time. We have to modify the constructor as `A(const A& other)` to fix this problem. That is to say, the parameter in a copy constructor should be passed by a reference.

Implementing a Class or Member Function in C++

The third type of C++ interview questions is based on implementing a class or some member functions. Usually it is more difficult to write code than to read and analyze sample code. Many C++ coding interview questions focus on constructor or destructor functions as well as overloading operators. For instance, the following problem about an assignment operator is such an example.

Assignment Operator

■ **Question 2** The declaration of class `CMyString` is found in Listing 2-9. Please add an assignment operator to it.

Listing 2-9. *C++ Code for Declaration of* `CMyString`

```
class CMyString {
```

```
public:
    CMyString(char* pData = NULL);
    CMyString(const CMyString& str);
    ~CMyString(void);

private:
    char* m_pData;
};
```

Classic, but Only Suitable for Newbie Developers

When an interviewer asks a candidate to overload the assignment operator, he or she asks the following questions to check the candidate's code:

- Does it return a reference? Before the assignment operator function ends, it should return the instance itself (`*this`) as a reference. If an assignment operator is overloaded as a `void` function, it has no chance to chain multiple assignments together. Suppose that there are three instances of the `CMyString` type: `str1`, `str2`, and `str3`. They cause a compiling error at `str1=str2=str3` if the assignment operator is overloaded as a `void` function.
- Is the argument passed by a constant reference? If the argument is passed by value, a copy constructor is called, and it is a waste of time. The call of the copy constructor is avoided if the argument is passed by reference. Additionally, it should not modify the status of the input instance during assignment, so the reference should be declared as `const`.
- Is the existing memory freed? If the old memory for `m_pData` is not deallocated before it allocates new memory, memory leaks occur.
- Does it protect against self-assignment? It returns directly and does nothing if the assignment source (input instance) is identical to the assignment target (`*this`). Otherwise if `*this` is the same as the input instance, its memory will be freed and its content cannot be gotten back anymore.

Listing 2-10 is a piece of C++ code, covering all four items.

Listing 2-10. *C++ Code for Assignment Operator (Version 1)*

```
CMyString& CMyString::operator =(const CMyString &str) {
    if(this == &str)
        return *this;

    delete []m_pData;
    m_pData = NULL;

    m_pData = new char[strlen(str.m_pData) + 1];
    strcpy(m_pData, str.m_pData);

    return *this;
}
```

This is a classic solution in textbooks. A junior candidate can pass this round of interview with the code above. However, it is far from perfection. An interviewer will have more stringent requirements if the candidate is a senior C++ programmer.

Exception Safety

In the previous code, the old `m_pData` is deleted before it is renewed. If an exception is thrown while reallocating memory due to insufficient memory, `m_pData` is a `NULL` pointer and it is prone to crash the whole application. In other words, when an exception is thrown in the assignment operator function, the status of a `CMyString` instance is invalid. It breaks the requirement of exception safety: If an exception is thrown, everything in the program must remain in a valid state.

Two approaches are available to achieve the goal of exception safety. The first one is to allocate memory with the `new` operator before deleting. It makes sure the old memory is deleted only after it allocates memory successfully, and the status of `CMyString` instances is valid if it fails to allocate memory.

A better choice is to create a temporary instance to copy the input data and then to swap it with the target. The code in Listing 2-11 is based on the copy-and-swap solution.

Listing 2-11. C++ Code for Assignment Operator (Version 2)

```
CMyString& CMyString::operator =(const CMyString &str) {
    if(this != &str) {
        CMyString strTemp(str);

        char* pTemp = strTemp.m_pData;
        strTemp.m_pData = m_pData;
        m_pData = pTemp;
    }

    return *this;
}
```

In this code, a temporary instance `strTemp` is constructed first, copying data from the input `str`. Next, it swaps `strTemp.m_pData` with `this->m_pData`. Because `strTemp` is a local variable, its destructor will be called automatically when the execution flow exits this function. The memory pointed to by `strTemp.m_pData` is what was pointed to by `this->m_pData` before, so the memory of the old instance is deleted when the destructor of `strTemp` is invoked.

It allocates memory with the `new` operator in the constructor of `CMyString`. Supposing that a `bad_alloc` exception is thrown due to insufficient memory, the old instance has not been modified, and its status is still valid. Therefore, it is an exception-safe solution.

Source Code:

`002_AssignmentOperator.cpp`

Test Cases:

- Assign an instance to another

- Assign an instance to itself
- Chain multiple assignments together
- Stress tests to check whether the code contains memory leaks

C#

The C# programming language is based on Microsoft .NET, the popular platform in the Windows ecosystem. Many companies focusing on Windows development include requirements on C# skills in their job descriptions.

C# can be viewed as a managed programming language based on C++, so there are many similarities between them. It does not take much time for a C++ programmer to learn C#. It is easy to learn the similarities, but it is difficult to distinguish differences. Many interviewers like to ask questions about the confusing differences. For example, the following is a piece of dialog from an interview:

Interviewer: Types in C++ can be defined as `struct` and `class`. What are the differences between these two types?

Candidate: The default access level in a `struct` is public, while in a `class` is private.

Interviewer: How about in C#?

Candidate: It is different in C#. The default access level for both `struct` and `class` is private in C#. However, their memory allocation models are different. A `struct` is a value type, and its instances are created on the stack. A type declared as a `class` is a reference type, and its instances are created on the heap.

Similar to C++, every type in C# has a constructor at least. Additionally, there might be two different methods (finalizer and `Dispose`) used to release resources in C#. The finalizer method looks similar to the destructor in C++, which is the type name followed by a complement symbol (~). However, the time to invoke them is different. The time to invoke a destructor in C++ is fixed, but the time to invoke the finalizer in C# is determined by the runtime. The finalizer is invoked by the garbage collector, and the time is unknown to programmers.

Additionally, there is a special constructor in C#, named static constructor. It is invoked automatically when its type is executed by the runtime at the first time, and it is invoked only once. There are many interesting interview questions about the static constructor. For example: What is the output of the C# code in Listing 2-12?

Listing 2-12. *C# Code about Static Constructors*

```
class A {
    public A(string text) {
        Console.WriteLine(text);
    }
}

class B{
    static A a1 = new A("a1");
    A a2 = new A("a2");
```

```
    static B() {
        a1 = new A("a3");
    }

    public B() {
        a2 = new A("a4");
    }
}

class Program {
    static void Main(string[] args) {
        B b = new B();
    }
}
```

Before any code of the type `B` gets executed, its static constructor will be invoked first. It initializes static data fields and then executes statements inside the static constructor. Therefore, it prints a line of "a1", and then another line of "a3".

When the execution flow reaches the statement `B b = new B()`, the ordinary constructor is invoked. It initializes its data fields first and then the statements in its constructor method, so it prints a line of "a2", and then another line of "a4".

Singleton

■ **Question 3** Please design and implement a class of which we can only create a single instance.

A class follows the singleton pattern if it can have one instance at most. Design patterns are very important in object-oriented design and development, so many interviewers like questions related to patterns. Singleton is the only pattern that can be implemented in dozens of lines of code, so it is quite suitable for interviews.

Workable Only in Single-Threading Applications

When the constructor is defined as a private method, none of the code outside the class can create its instances. A static method inside the class is defined to create an instance on demand. The class `Singleton1` is implemented based on the solution in Listing 2-13.

Listing 2-13. *C# Code for Singleton (Version 1)*

```
public class Singleton1 {
    private Singleton1() {
    }
```

```
    private static Singleton1 instance = null;
    public static Singleton1 Instance {
        get {
            if (instance == null)
                instance = new Singleton1();
            return instance;
        }
    }
}
```

In this class, an instance is created only when static field `Singleton1.instance` is `null`, so it does not have the opportunity to get multiple instances.

Works with Multiple Threads but Is Inefficient

`Singleton1` works when there is only one thread, but it has problems when there are multiple threads in an application. Supposing that there are two threads concurrently reaching the `if` statement to check whether `instance` is `null`. If `instance` is not created yet, each thread will create one separately. It violates the definition of the singleton pattern when two instances are created. In order to make it work in multithreading applications, a lock is introduced as shown in the `Singleton2` in Listing 2-14.

Listing 2-14. *C# Code for Singleton (Version 2)*

```
public class Singleton2 {
    private Singleton2() {
    }

    private static readonly object syncObj = new object();

    private static Singleton2 instance = null;
    public static Singleton2 Instance {
        get {
            lock (syncObj) {
                if (instance == null)
                    instance = new Singleton2();
            }

            return instance;
        }
    }
}
```

Suppose there are two threads that are both going to create their own instances. As we know, only one thread can get the lock at a time. When one thread gets it, the other one has to wait. The first thread that gets the lock finds that `instance` is `null`, so it creates an instance. After the first thread releases the lock, the second thread gets it. Since the `instance` was already created by the first thread, the `if` statement is `false`.

An instance will not be recreated again. Therefore, it guarantees that there is one instance at most when there are multiple threads executing concurrently.

The class `Singleton2` is far from perfect. Every time `Singleton2.Instance.get` executes, it has to get and release a lock. Operations to get and release a lock are time-consuming, so they should be avoided as much as possible.

Double-Check around Locking

Actually a lock is needed only before the only instance is created in order to make sure that only one thread get the chance to create an instance. After the instance is created, no lock is necessary. We can improve performance with an additional `if` check before the lock as shown in Listing 2-15.

Listing 2-15. *C# Code for Singleton (Version 3)*

```
public class Singleton3 {
    private Singleton3() {
    }

    private static object syncObj = new object();

    private static Singleton3 instance = null;
    public static Singleton3 Instance {
        get {
            if (instance == null) {
                lock (syncObj) {
                    if (instance == null)
                        instance = new Singleton3();
                }
            }

            return instance;
        }
    }
}
```

In the class `Singleton3`, it locks only when `instance` is `null`. When the instance has been created, it is returned directly without any locking operations. Therefore, the time efficiency of `Singleton3` is better than `Singleton2`.

`Singleton3` employs two `if` statements to improve time efficiency. It is a workable solution, but its logic looks a bit complex, and it is error-prone for many candidates during interviews. Let us explore simpler, and also better, solutions.

Utilization of Static Constructors

It is guaranteed that a static constructor in a C# class is called only once at most. If it only creates an instance in the static constructor, there is one instance at most. A concise solution for the singleton pattern with a static constructor is shown in Listing 2-16.

Listing 2-16. *C# Code for Singleton (Version 4)*

```
public class Singleton4 {
    private Singleton4() {
    }

    private static Singleton4 instance = new Singleton4();
    public static Singleton4 Instance {
        get {
            return instance;
        }
    }
}
```

In the class `Singleton4` above, an instance is created when the static field `instance` gets initialized. Static fields in C# are initialized when the static constructor is called. Since the static constructor is called only once by the .NET runtime, it is guaranteed that only one instance is created even in a multithreading application.

The time to execute the static constructor is out of the programmers' control. When the .NET runtime reaches any code of a class the first time, it invokes the static constructor automatically. Therefore, the time to initialize `instance` is not the first time to invoke `Singleton4.Instance`. If a static method is added into `Singleton4`, it is not necessary to create an instance to invoke such a static method. However, the .NET runtime invokes the static constructors automatically to create an instance when it reaches any code for `Singleton4`. Therefore, it is possible to create an instance too early, and it impairs the space efficiency.

Creating an Instance When Necessary

In the last implementation of the singleton pattern in Listing 2-17, `Singleton5`, it creates the only instance on demand.

Listing 2-17. *C# Code for Singleton (Version 5)*

```
public class Singleton5 {
    Singleton5() {
    }

    public static Singleton5 Instance {
        get {
            return Nested.instance;
        }
```

```
    }

    class Nested {
        static Nested() {
        }

        internal static readonly Singleton5 instance = new Singleton5();
    }
}
```

There is a nested private class `Nested` in the code for `Singleton5`. When the .NET runtime reaches the code of the class `Nested`, its static constructor is invoked automatically, which creates an instance of type `Singleton5`. The class `Nested` is used only in the property `Singleton5.Instance`. Since the nested class is defined as private, it is inaccessible outside of the class `Singleton5`.

When the `get` method of `Singleton5.Instance` is invoked the first time, it triggers execution of the static constructor of the class `Nested` to create an instance of `Singleton5`. The instance is created only when it is necessary, so it avoids the waste associated with creating the instance too early.

Solution Comparison

Five solutions are introduced in this section. The first solution is workable only in a single-threading application. The second one works in a multiple-threading application, but it is inefficient because of unnecessary locking operations. The first two solutions are not acceptable from an interviewer's perspective. In the third solution, it employs two `if` statements and one lock to make sure it works in a multiple-threading application efficiently. The fourth one utilizes the static constructor to guarantee only an instance is created. The last solution improves space efficiency with a nested class to create the instance only when it is necessary. The last two solutions are recommended for interviews.

Source Code:

`003_Singleton.cs`

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

Summary

Programming languages are basic tools for software engineers. Many interviewers like to check candidates' proficiency level on one or more programming languages.

This chapter discusses interview questions of the four most popular languages: C, C++, C#, and Java. Usually interview questions of programming languages can be divided into three categories: understanding of keywords and concepts, analyzing execution of a given piece of code, and writing code. Sample interview questions are discussed in each category. More coding interview questions solved in these four languages are available in the next chapters.