Lior Shahverdi CSC401

Implementation of Semantics and Constructs in the Java Programming Language

Fifty reserved keywords exist in the Java programming language. Some of these keywords could only be used within a single, distinct context while other keywords could be used in multiple contexts. The keyword final falls under the multiple contexts category. Simply put, final means once some arbitrary entity is declared and initialized it is set in stone. If we define a “unit” in Java as any variable or method or class, then the reserve keyword final could be incorporated into the declaration of any such unit. The Java SE 8 Language Specification states that in terms of variable assignment, a final variable could only be assigned a value once throughout the runtime of the program, otherwise javac (java compiler) will raise a compile-time error. To illustrate this case, suppose we created a class called EnglishClass with a final int variable called numberOfStudents. If we try to initialize this variable upon instantiation with the integer value 10 and later on in the program we attempt to add 1 or subtract 1 from numberOfStudents, the compiler will raise an error saying “*error: cannot assign a value to final variable numberOfStudents*”. Therefore, it would be a poor design choice to use the final keyword in this case because we know the value of numberOfStudents is subject to change throughout the lifetime of the program. What was left unmentioned in this example is that the integer value bound to numberOfStudents is a primitive type in Java. Suppose our same class EnglishClass had a different final variable of type Student called nameOfProfessor that was initialized by calling the default constructor. The difference with this variable is that it does not carry a primitive value, but rather a reference to a Student object. Since this variable is also a final variable, it is the reference, the address of the Student object in the JVM’s memory heap, which will remain the same throughout the program’s lifetime. However, the state of the Student object that our reference refers to is accessible for modification even though our reference to it is final. (If our reference were to point to a different object later in the program, that would also raise a compile-time error.) So if the following three lines of code are placed into a Student class:

int classSize;

Student(int age, String name) { this.age = age; this.name = name; }

final static Student someStudent = new Student(15,”John”);

The name *origin* stores a reference to an object that is an instance of a class Student. The reference stored in Student.someStudent will never change but an operation on this Student object might change its own state by possibly modifying classSize, for example. If we wished to create another class X1 and we wish to make the entire X1 class final, then the limitations that this will create is that our new X1 type could no longer be extended. That is, X1 is not permitted to have child classes that could inherit attributes from it since it is a final class. Likewise, methods in a class that are declared to be final cannot be overridden. To illustrate this concept, suppose we have a class called Worker with a default constructor and a final method named computeSalary() which contains around 4 lines of code and returns a numeric value of type double. If we were to create an additional class called Manager that extends the class Worker, this Manager class is unable to create its own computeSalary() method to override its parent class’s final version of that method. However, if the Worker class was originally an abstract class and computeSalary() in that Worker abstract class was declared but not defined, then computeSalary() would have to be defined in subsequent child classes of this abstract Worker class such as our Manager class.

For the sake of comparison, the C++ programming language handles the final reserve keyword slightly differently yet still quite similar to Java. In C++, there is a fundamental distinction between virtual and non-virtual functions: Virtual functions are known as member functions (member functions are simply functions associated with a certain class) whose actions or behaviors can be replaced by and thus overridden in derived classes. In C++, the closest equivalent of classes are structs and the inclusion of final in a struct’s syntax is slightly different than a class in Java. If we use Oracle’s online Java Documentation example and have a Java Chess application with a class called ChessAlgorithm, then the skeleton for this class would appear like so:

class ChessAlgorithm{

enum ChessPlayer { WHITE, BLACK }

final ChessPlayer get FirstPlayer() { return ChessPlayer.WHITE; }

}

Whereas in C++ this same application might be implemented like so:  
struct initial {

enum ChessPlayer {WHITE, BLACK} ;

virtual ChessPlayer getFirstPlayer() final {return ChessPlayer.WHITE; }

}

Furthermore, C++ has the construct known as a union which essentially allows the same portion of memory to be accessed as different data types. Since unions do not exist in Java given that it is a strongly-typed language, the use of final in conjunction with unions in C++ is a feature unique to the language.

A similar keyword in Java is finally. This keyword is used right before a block of code that comes after a try-catch block. If a programmer writes Java code that may or may not raise an exception, they insert that code into a try block. Since this code may or may not cause the program to crash during runtime, it could be considered unstable since the user is not certain of what will happen. The use of a finally block immediately following a try-catch block confronts this phenomenon. Whether or not an exception occurs, the finally block will immediately be entered once the try-catch is completed executing. Although finally is a keyword used in the context of a process rather than a unit, it is similar to the reserved keyword final in the sense that it will ensure that a certain entity is fixed.

The main feature that Java introduced to the field is its cross-platform compatibility; the same program will run precisely the same on a computer from IBM as a computer from Oracle. The way the developers of Java were able to achieve this was through the abstract computing machine known as the Java Virtual Machine (JVM). The way the JVM implements dynamic memory allocation is through its own data structure known as the Java virtual machine heap, which is split into separate regions known as generations. Depending on the duration of an object’s lifetime in a Java program, the memory associated with the object is either stored in the young generation or tenured generation of the heap. But this memory is only put into the heap once that object of some class has been instantiated. The definition of the class itself is stored elsewhere in a separate region known as permanent generation. Whether or not this permanent generation region should be stored elsewhere has been a philosophical topic of debate since its creation. In the earlier years of programming and computing history, classes were mostly being used statically and so as a result, class unloading (not in terms of objects but in terms of memory management in general) had not been used nearly as much. Objects and their associated class definitions were stored in the same area. The decision to move class definitions to a separate region of memory had been a decision that ultimately optimized compiler performance at that time. In today’s modern applications, class unloading can occur much more frequently so it is not always clear if permanent generation improves performance in this case. As of Java SE 8, permanent generation has been removed from the JVM. Some of the contents of what would have been in permanent generation have been moved to the Java heap while other contents have been moved to native memory.

One of the built-in data types in Java is String, which comes with its own set of available methods, one of which is intern(). Java’s compiler uses a concept known as string interning to store String values in a string pool, which is typically a hashmap. If two string references refer to two equal string values yet intended to be located in two different locations in memory, Java’s compiler will still associate those two references to the same single location and value in the heap to enhance overall storage management; the thought behind this implementation is there is no need to store duplicate copies of the same String value in two locations in memory even if that is the programmer’s intention.

String x = “John”; String y = “Nancy”; boolean theyMatch = false;

if (x.intern() == y.intern()) theyMatch = true;

If two strings are stored as interned strings in the heap and the method intern() is invoked on each of them, the return values for each of those expressions are now integers rather than strings. This reduces the complexity of string comparison to the complexity of integer comparison which, through the perspective of the compiler, takes significantly less amount of time to execute. “In Java 6, Java’s heap had a section within it called PermGen which had a fixed size for storing class definitions and string pool” ( java-performance.info/string-intern-in-java-6-7-8/ ). The string pool stored the interned strings as well as all string literals loaded or called earlier in the program. In Java 7, all strings had been moved to the heap. This meant the string pool was no longer limited to the fixed size of the PermGen region. Since Java 8 has been released, class metadata such as definitions are not adjacent to the Java heap but are currently in native memory.

When designing a programming language, one of the main tasks is how the compiler or interpreter should handle subprograms. In Java subprograms are known as methods whereas other imperative languages may refer to them as functions. Once the runtime machine enters a method, what is accessible within that method? Can any information be accessed outside of that method? This issue is known as scoping. Each method has an associated activation record containing data vital to that method and the collection of activation records that the runtime machine can access at any point of the program’s lifetime is known as the referencing environment. When compilers and interpreters are parsing the source code and come across an identifier, the compiler or interpreter needs to immediately determine the properties associated with this identifier such as variable type, memory location and literal value. This association of mapping low level data to high level structures is known as binding and the unique time that this association happens is known as binding time. In some languages like JavaScript for example, functions are allowed to be nested within other functions which clearly complicates when and how binding time should be handled. The three ways to handle scoping are known as shallow binding, deep binding and ad hoc binding. To illustrate this example, let us consider a simple JavaScript program:

function s1(){

var x;

function s2() { alert(x); }

function s3() {

var x;

x = 3;

s4(s2);

}

function s4(sx) {

var x;

x = 4;

sx();

}

x = 1;

s3();

}

In this program, function s1 has a declaration for a variable called x, 3 functions nested within it, an assignment statement for x and a function call to s3. When s3 is called, the program will branch to s3, assign the value 3 to the variable x, call function s4 and pass function s2 as a parameter to call within s4. When s2 gets called within s4 it prints a value for x but since there are multiple places in this trace where x gets assigned a value, the value for x at this point in the trace is dependent on what type of binding is implemented. If shallow binding is implemented then s2 will use the value from the assignment statement in s4 (4) because it was where x had been declared most recently. This case is unique because s4 is a function that has another function passed to it as a parameter. If deep binding is implemented the compiler or interpreter would look to the first time the variable x had been declared instead of the last and use that nesting level. In this program, the first time x is declared is within s1 and because it is assigned the value 1 at that same nesting level, x would store 1. This means that no matter how many times the value of a variable declared at nesting level 0 (outermost, global level) gets modified in deeper nesting levels, if deep binding is being implemented those changes will have never occurred. Modifications made to a variable are then forced to happen at the same nesting level as the declaration of the variable. With ad hoc binding, x would store the value assigned at the referencing environment of the s4 method call in s3. This means that if the method s4 will modify x when it is entered, it will not matter. While JavaScript allows the nesting of functions within functions, Java does not. However, Java’s use of access modifiers such as private, public, protected assist in making methods more flexible.

Some languages (including Java) allow passing subprograms as parameters to other subprograms. In Java, one method can be passed to other methods as a parameter only if that method’s return type is non-void (object or primitive).  
String x = “Programming”;

System.out.print( x.substring(0,3) ) ;

In the example above, x.substring(0,3) is being passed as an argument and will return a shorter string to System.out.print. As of Java 8, lambda expressions have been introduced to the language. According to Oracle.com, “lambda expressions are a clear way of representing methods using expressions”. Lambda expressions are anonymous methods that could be used on anonymous classes. The syntax for a lambda expression is as follows: (argument list) 🡪 body

The special part about the argument list component in this syntax is that argument types must either be inferred or declared. Before Java 8, passing an argument whose type was inferred had never been allowed. This allows programmers to write generic methods for objects whose type is not established at compile-time, which is quite useful. Suppose a company had 40 employees with 4 different salary rankings. The programmer’s task is to create a class for each salary rank and a method within each class to compute the salary for that type of employee called computeSalary(). If the company’s budget were ever subject to change, the number of different salary rankings will have to ultimately change as well. Rather than have the programmer update the code according to the company’s current configuration, he or she can create an anonymous method to compute an employee’s salary to use on any employee. This way, if the code were ever to change the programmer would only have a single area of code to refactor instead of multiple areas of code.

Works Cited

1. <https://docs.oracle.com/javase/specs/>
2. <http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html>
3. <http://en.cppreference.com/w/cpp/language/final>
4. <https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation>
5. <https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html>
6. Sebesta, Robert W. *Concepts of Programming Languages*. Reading, MA: Addison-Wesley, 1999. Print.
7. <http://java8.in/java-8-lambda-expression/>