CSci 365: Organization of Programming Languages

Report 2

Max Forst & Kenneth Jahnke

10/31/2024

**Ada 2022**

The last major update to the Ada programming language was in 2012. Over a decade, many developments were introduced within the realm of computing. The ISO Working Group 9 (the managing body for Ada) sought an update to the language to meet modern demands. The group had two overarching goals when requesting improvements to the language: (A) "Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and security are prime concerns"; and (B) "Improvements that will remedy shortcomings of Ada." The most potent computing-realm consideration was the advancement of multi-core/multi-thread processors and how Ada could integrate with them. How Ada 2022 integrated with them:

1. Parallelism. Ever-increasing core and thread counts were not a technology that Ada 2012 could capitalize on to the greatest extent. Ada 2022 introduced added the ability to explicitly declare that blocks of code should be executed in parallel, thus making applications more high-performance as they can deliberately take advantage of multi-core/multi-thread architecture.
2. `Nonblocking` aspect. Ada introduced the `Nonblocking` aspect to designate if specific tasks or subprograms are allowed to cause blocking operations, such as `sleep`, input/output data delays, or waiting for a signal from another task. Given that Ada is implemented in aerospace systems where timing of tasks and functions is critical, it is important to specify where deadlocks within code are and are not allowed to be. Hence, the `Nonblocking` aspect. This plays well with the parallelism considerations from manifestation (1) because it decreases the likelihood that more critical functions of a system will not “lock up” due to waiting on subprograms or separate tasks, thus increasing the safety of the system.
3. Explicit definition of global variable access. Ada now allows user to declare which subprograms can and cannot access global variables. The `global-in` principle allows a subprogram to read a global. The `global-out` principle allows a subprogram to modify a global. Within large and complex systems running processes in parallel, it is imperative to explicitly allow/disallow subprogram access to variables from higher scopes since unintended modification of commonly used variables can create catastrophic effects.

There were other improvements upon the language, such as expansion of the standard library, ability for users to define their own numeric literals, and code integration with C/C++, but these are minor compared to the major Ada 2022’s major consideration of utilizing multi-core/multi-thread processing architectures.

Website: <https://ada-lang.io/>

Reference Material: <http://ada-auth.org/standards/index.html>

Sources:

<http://ada-auth.org/standards/22over/html/Ov22-TTL.html>

<https://www.gyata.ai/nodejs/blocking-vs-non-blocking-operations>

<https://ntrs.nasa.gov/api/citations/19890006967/downloads/19890006967.pdf>

**Fortran 2023**

The reasoning for updating Fortran is like that of Ada: Modernize a legacy programming language. Some modernizations:

1. Enhancement of concurrency and parallelism features. The reason for implementing this is obvious: modern systems are powerful given multi-core/multi-thread architecture. To be specific, the `do concurrent` redaction specifier was implemented so developers could run loop iterations in parallel, thus quickening execution time of the overall loop.
2. Greater line lengths and character counts. The limit on the length of a line was increased to 10,000 characters. The limit on a statement length was increased to one million characters. The reason? This aids development by allowing developers to write code without concern for arbitrary formatting constraints. What's more, few programs are confined to a terminal where line length and character counts would matter.
3. New conditional expression features. Example: `value = ( e>6.9 ? e : 6.9)`. This is essentially just an if-then-else statement, but in a more condensed form. By reducing the number of lines required to implement logic, the code base may become more human-readable and easier to maintain.

These are some of the major feature introductions to Fortran 2023. There were a few other updates, such as improved system clock handling, `split` and `tokenize` were added as string parsing routines, and some features to enhance how the language implements trigonometric functions.

Website: <https://fortran-lang.org/>

Sources:

<https://wg5-fortran.org/N2201-N2250/N2212.pdf>

<https://degenerateconic.com/the-new-features-of-fortran-202x.html>

**Java**

There have been many versions of Java released since 2015. Rather than focusing on one major release, we will give some details of the more impactful releases, the long-term support (LTS) releases, as well as the current version, Java 23. As for some background on each language, Java 11 was introduced in 2018 by Oracle, along with support from OpenJDK, an open-source community dedicated to enhancing Java. The Java version 17 was released 2021, and version 21 was released in 2023. Java 23 is the latest version to be released, only coming out in September of 2024. Though not a LTS version, we thought we would cover it here as well.

The main design goals all of these versions of Java aimed to implement are as follows. They needed to be robust and have great performance in order to have staying power with other languages. They also needed to be portable and platform independent to attain wide usability. These goals have certainly been met, as you can now find Java running smoothly almost anywhere, from mobile devices, to internet services, to data analytical companies.

Java 11.

* Introduced new methods for string manipulation, such as .isBlank(), .strip(), and .lines(). Also, new methods for files were also added, such as .readString() and .writeString(), making it easier to deal with files.
* Support for the ‘var’ keyword in lambda expressions was also added, which helped with code readability and allowed the compiler to infer what type a variable would be, without having to specifically define it.
* A new HTTP client was made a standard feature, replacing the old HttpURLConnection API. This switch allowed for better overall performance for HTTP requests.

Java 17.

* Enhanced pattern matching for `switch`. Prior to this version, `switch` in conjunction with the `instanceof` operator could work on specific types (numeric types, enumeration types, and strings) and search for exact equivalence values. This limited the search for alternative/similar values. The syntax was improved (i.e., the operability of `switch` was expanded) to not only condense the amount of code written to perform such a search but to allow for pattern matching across a wider range of types and to explore similar values. This feature was implemented in Java 17, but the feature continued to be refined. It reached its current full operability in Java 21.
* Sealed classes. With the implementation of sealed classes, developers were granted options on how other classes can extend and implement them. This allows control over inheritance, dictation of implementation responsibility, and restricted use of super-classes.

Java 21.

* Pattern matching for `switch` finalized. First introduced in Java 17, the pattern matching abilities of the `switch` statement were finalized. Part of the revisions were in part for better integration with record patters functions.
* Record patterns. The pattern matching features were expanded to recognize instances of `record` classes. This allowed for more sophisticated data queries.
* Virtual threads implementation. Virtual threads consume fewer system resources than platform (OS-managed) threads. Since they are not tied to the physical limitations of the OS (i.e., wait time is limited), processes running along virtual threads are much faster. When the Java Virtual Machine (JVM) recognizes that a blocking operation is encountered along a particular thread, it can create or call upon the resources of an existing thread to keep the program moving. Virtual threads are not faster than platform threads. The JVM merely diverts processes to open (non-blocked) virtual threads. In other words, more of the breadth of the JVM's resources are utilized, though the speed of individual resources is fixed. That the JVM can create millions of threads and divert processes to such a large pool of open threads, this infrastructure creates highly efficient environments for concurrent subprograms and tasks.

Java 23.

* Enhanced primitive type patterns for the ‘instanceof’ and ‘switch’ statements, meaning that now primitive types could be used directly in these statements, leading to more concise and cleaner looking code.
* Logic for importing modules was simplified, meaning that declaring dependencies became more flexible.
* Introduced Generational ZGC, which allowed for garbage collection to handle short-lived and long-lived objects differently, minimizing time between memory deallocation and improving overall performance.

Comparing Java to other languages, it’s only natural that we compare it to its closest relative, C++. James Gosling, Java’s developer, actually created Java to fix an issue with C++. He wanted to use a language for devices like set-top boxes or televisions, but C++ was to memory intensive. This lead to the development of Java. C++ has better overall performance than Java, but Java offers much better memory management. Java is much more portable, meaning that it will run the same on every machine, while C++ is highly platform dependent, meaning that code compiled on machine might not run the same on another. Java is also entirely OOP, unlike C++, which has OOP capabilities, but is largely procedural.

Overall, Java has made significant contributions as a programming language. It runs smoothly on almost any device with little to no recompilation when switching between devices. That is a major advantage when compared to other programming languages. However, there are concerns with Java’s performance, which might make other languages more desirable. Despite this though, Java is still a great language to program with.

Website: <https://www.java.com/en/>

Sources:

<https://openjdk.org/jeps/409>

<https://openjdk.org/jeps/406>

<https://openjdk.org/jeps/441>

<https://openjdk.org/jeps/440>

<https://openjdk.org/jeps/444>

<https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-BEC799E0-00E9-4386-B220-8839EA6B4F5C>

<https://www.geeksforgeeks.org/java-11-features-and-comparison/>

<https://www.baeldung.com/java-11-new-features>

<https://www.geeksforgeeks.org/jdk-23-new-features-of-java-23/>

https://blogs.oracle.com/java/post/the-arrival-of-java-23

**JavaScript ES6**

JavaScript ES6 was the second major revision to the JavaScript language, and it is still the major version still in use today. JavaScript ES6 was released to the public in 2015 by the ECMA technical committee, which is an organization that standardizes computer hardware, communications, and programming languages. The ECMA took over the development of the language after its creator, Brendan Eich, who developed it for Netscape, handed over JavaScript in hopes that it could be a widely adopted language if standardized. The ECMA then created a standard for the language called ECMA-262, also known as ECMAScript. Since then, JavaScript as a language in general has gone through several versions and changes, but ES6 is arguably one of the most important changes to the language that has been made.

JavaScript ES6, like its previous versions, had one main goal; make webpages feel “alive.” In fact, JavaScript was originally called LiveScript because of this reason, but the name was changed later for marketing reasons. JavaScript ES6 accomplishes this goal by manipulating two other closely related languages called HTML and CSS. HTML controls the physical structure of a webpage, while CSS controls the appearance or style of a page. So, to manipulate a webpage, JavaScript could physically add or remove HTML elements from the page, changing its structure, or it could change the styling of the CSS, causing color changes, layout changes, font changes, etc. Those are not the only things that JavaScript can do though, it can also react to various user actions, send network requests, remember data, and so much more.

As mentioned earlier, ES6 was a very important update for JavaScript, so it added a lot of new features and improved some existing features. Two of the most notable additions are the ‘let’ and the ‘const’ variable declarations. The ‘let’ variable declaration is useful for variables defined inside block structures, like loops, as they are not visible outside of the block that they are defined in, unlike the previous ‘var’ variable declaration. The ‘const’ variable declaration was added to fix the problem of accidentally overriding previously declared variables. Once a ‘const’ variable was declared, it could not be overridden or changed, allowing for consistent variables in complicated functions. The addition of classes was also added, allowing for more dynamic variables and OOP capabilities. It also introduces promises, which provided a better way to handle asynchronous code with a ‘.then’ structure. It also introduced modular support, which allowed users to import functions from other JavaScript files into their own file. ES6 also added the ‘…’ operator, which allowed functions to take any number of parameters, which would then be stored in an array structure. There are many more new features added in addition to the ones already described, but those were some of the most notable additions. Moving on to some improved features, it can be noted that ES6 added several methods to default objects like Strings, Arrays, Numbers, and the Math object. Strings could now use the .includes(), .startsWith(), and .endsWith() methods for easier searching. Arrays got several new methods for searching them as well, but the most notable method is probably the .from() method, which allowed for the creation of Arrays from several other objects, like Strings, Sets, Maps, NodeLists, and so on. ES6 also added several methods to the Math object that made complex mathematical operations easier, like .cbrt(), which returns the cube root of a number, or .log10(), which returns the base 10 logarithm of a number.

Let’s now compare JavaScript ES6 to other programming languages. One would assume that JavaScript should be compared to Java, but that is not the case, as the two are very different languages. JavaScript adopted its name simply because at the time of its development, Java was very popular, so the developer wanted to make it seem like it was related. In fact, the closest relative that JavaScript ES6 has is PERL. While they are both multi-paradigm, dynamically-typed scripting languages that prioritize flexibility, JavaScript is mainly designed for web development, while Perl focuses on text processing and system scripting. Despite their similarities in approach, their key applications set them apart fundamentally.

JavaScript has made a significant contribution to the web development world. Without it, most websites would not exist as they are today. JavaScript ES6 is a very powerful language because it can run in almost any environment with a JavaScript engine. This includes both the front end of websites as well as the backend with Node.js. A major disadvantage that it might have over other languages is that it is primarily run in browsers, which limit its ability to do things like access user files and manipulate them for security reasons. While these limitations do make sense, I do think it would be beneficial for JavaScript ES6 to have these capabilities in certain situations. Another advantage that it has over other languages is that it is interoperated, not compiled, so it can run very quickly in browser settings. JavaScript ES6 does not actually have an official website were one can go to download the lates version, instead the closest thing is the ECMA-262 specification which all modern browsers follow to fully adapt JavaScript ES6 for webpages. Documentation can be found on various websites as well, such as Mozilla, and W3Schools.

Website: <https://262.ecma-international.org/15.0/index.html>

Sources: <https://www.w3schools.com/js/js_es6.asp>

<https://www.w3schools.com/js/js_history.asp>

<https://javascript.info/intro>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

**Swift 5.0**

Swift 5.0 was a major upgrade to the Swift language, a language mainly used for developing applications for Apple devices. This version was released in 2019 and was a significant milestone in the evolution of the Swift programming language compared to its initial release. The most notable thing about 5.0 was that it allowed for long-term development on Apple’s platforms without requiring constant recompilation for new language versions. Ever since version 1.0, Swift was meant as a replacement for Objective-C, which ran Apple devices before, as Swift was a safer, more efficient, and more expressive programming language. Swift not only took inspiration from Objective-C, but several other languages like Python, Rust, and Ruby as well. Chris Lattner, the creator of the language, later left the language to be further developed by Apple Inc.’s Swift team, who also received contributions from an open-source community on GitHub. These contributions helped launch Swift into a new era with version 5.0, which was a major leap forward in both usability and longevity for developers.

As mentioned, Swift was developed as a replacement for Objective-C due to its limitations, such as its reliance on dynamic typing and a complex memory management model. Switching to Swift had many benefits, such as safety and performance improvements, but the upgrade to Swift 5.0 had the main goal to improve performance even more and “future proof” the language. It was designed to eliminate bugs through features like optionals, strong typing, and automatic memory management. It also had the goal to make it easier for programmers to understand by implementing features from several other languages, like closures, generics, and functional programming capabilities. Perhaps most importantly though, it was designed to work seamlessly with future versions going forward, as well as still be backward compatible with Objective-C.

Before Swift 5.0, developers’ code would often break and need several minor rewrites when a newer version was published. This is because syntax or features of the code would change and sometimes render the old code unusable. However, with Swift 5.0 one major change made to fit their goal of stability and “future proofing” was the introduction of ABI Stability. This meant that all future releases would incorporate all Swift libraries into macOS, iOS, tvOS, etc. going forward, which meant that apps could be easier to build, as well as have much smaller file sizes, as developers wouldn’t have to manually include those libraries in their apps. This also meant that future updates would no longer break existing code going forward. Performance was also improved in several different ways, including string manipulation, dictionary and set manipulation, and reduced compile times. Strings, in addition to the performance updates, got an overhaul that allowed for raw strings, meaning that things like regular expressions were easier to work with. Swift 5.0 also improved interoperability with other languages, such as Python, JavaScript, and Ruby. In addition, a new data type called Result was added, which allowed for more dynamic handling of complex asynchronous functions depending on if the result was a success or a failure.

Because Swift was built as a replacement for Objective-C, it’s only natural that we should explore their differences and explain why the switch was made. Swift 5.0 focuses on safety with its typing system and optionals, unlike Objective-C, which relies on dynamic typing, which can lead to runtime errors. Swift 5.0’s syntax is also more simple and readable than Objective-C’s syntax because it is closer to human language. Swift also uses an automatic memory management system, which reduces manual memory handling, unlike Objective-C, which uses a manual memory handling system. As mentioned earlier, Swift 5.0 also took inspirations from Rust, so let’s compare them too. Their syntaxes are similar, but some of their bigger differences include how Rust doesn’t support class inheritance, unlike Swift 5.0, how Rust has more low-level control than Swift, and how Swift must rely on outside packages for asynchronous functions, unlike Rust.

While not a big of a contribution as other languages, Swift 5.0 has made a significant impact on many people’s lives. Without it, many of today’s mobile applications might be slower, less secure than today, or might not even exist at all. Despite this though, one major downside of Swift is that it is mainly only tailored for Apple devices and the Apple ecosystem. Even though Swift is available on other platforms as well, it doesn’t have as significant of an impact as other languages on those platforms. So, despite it being extremely fast and efficient for mobile applications, no real use has come from it other than that.

Website: <https://www.swift.org/>

Sources: <https://www.swift.org/blog/swift-5-released/>

<https://www.hackingwithswift.com/articles/126/whats-new-in-swift-5-0>

<https://www.geeksforgeeks.org/difference-between-swift-vs-objective-c/>

<https://dev.to/rhymu/swift-vs-rust-an-overview-of-swift-from-a-rusty-perspective-18c7>