# A Comparison of Multithreading in Python, Java, and C++

Grant Haataja

CSCI 365 - Organization of Programming Languages - Spring 2020

May 5, 2020

# Introduction

Multithreading is an advanced computer science concept that is quickly becoming more widely used for a variety of tasks. This makes it increasingly important to gauge which programming languages support multithreading in the best way for the selected task.

Multithreading works by allowing a program to utilize multiple threads to share process resources but execute independent processes. This allows a program to work on multiple processes concurrently, as if it were multitasking. [1]

In this paper, mulithreading capabilities will be analyzed and compared in Python, Java, and C++ alongside a general multithreading example, to determine the strengths and weaknesses for multithreading in each language.

Although the use of multithreading techniques can greatly improve efficiency of programs, it often requires greater forethought and more care when programming, as there are numerous problems that can arise including deadlock, race conditions, and unpredictability. These issues will be discussed, particularly if the languages analyzed have specific tools to mitigate these issues.

# Background

Python was released in 1991, as a successor of the programming language ABC. [2] Its applications are numerous, but for this analysis we will be focusing on the threading library in Python, which provides capability for multithreading.

Java was created in 1995, and is a large driving force behind the internet, and many

mobile applications. [3] Java is also used frequently in multithreading applications, so an analysis of threading would not be complete without including Java.

The development of C++ is not as easily defined. It was built off C, with the main difference being the addition of classes and inclusion of extensive libraries, including some that support the use of threading. This analysis will compare the threading libraries in each of these three languages, to analyze the reliability and ease of use over a basic threading program.

## Analysis

Below is a snippet of code showing how to set up threads in Python3. [4]

```
#Basic Python threading example
import threading
#function to print cube of given number
def print_cube(num):
  print("Cube: {}".format(num * num * num))
#function to print square of given number
def print_square(num):
  print("Square: {}".format(num * num))

if __name__ == "__main__":
  #create threads
  t1 = threading.Thread(target=print_square, args=(10,))
  t2 = threading.Thread(target=print_cube, args=(10,))

```

```
#start threads
t1.start()
t2.start()
#wait until threads are finished executing
t1.join()
t2.join()
#both threads completely executed
print("Process Complete.")
```

This code will first print "100", then "1000", then "Process Complete." The same example will be used to show how threading works in Java and C++. Note that Java has two main ways to use threading; extending the Thread class and implementing the Runnable interface. [5] The latter will be the focus in this analysis.

```
//Basic Java threading example
class ThreadingExample implements Runnable {
    private String actionType;
    private int number;

    ThreadingExample(String type, int num) {
        if (type.equalsIgnoreCase("Square")) {
            actionType = "Square";
        }
        else if (type.equalsIgnoreCase("Cube")) {
            actionType = "Cube";
        }
        number = num;
    }
```

```
15
16      public void run() {
17        //run the given action
18        if (actionType.equals("Square")) {
19          System.out.println("Square: " + (number * number));
20        }
21        else if (actionType.equals("Cube")) {
22          System.out.println("Cube: " + (number * number * number));
23        }
24      }
25  }
26  //Main Class
27  class Multithread {
28      public static void main(String[] args) {
29      //create threads
30      Thread t1 = new Thread(new ThreadingExample("Square", 10));
31      Thread t2 = new Thread(new ThreadingExample("Cube", 10));
32      try {
33        //start threads
34        t1.start();
35        t2.start();
36        //wait until threads are finished executing
37        t1.join();
38        t2.join();
39      } catch (Exception e) {
40        System.out.println(e);
41      }
```

```
42      //both threads completely executed
43      System.out.println("Process Complete.");
44      }
45  }
```

This will produce identical output to the example in Python, though it requires over twice as many lines of code. In C++, the amount of code is somewhere in between Python and Java, but most of the difference is in the functions themselves, not the code required to start and handle threads.

```
1  //Basic CPP threading example
2  #include <iostream>
3  #include <thread>
4  #include <string>
5
6  //function to print cube of given number
7  void cube(int num) {
8          int numb = num * num * num;
9          std::string str = "Cube: " + std::to_string(numb) + "\n";
10         std::cout << str;
11 }
12 //function to print square of a given number
13 void square(int num) {
14         int numb = num * num;
15         std::string str = "Square: " + std::to_string(numb) + "\n";
16         std::cout << str;
17 }
18
```

```
int main() {
        //create and start threads
        std::thread t1(square, 10);
        std::thread t2(cube, 10);
        //wait until threads are finished executing
        t1.join();
        t2.join();
    //both threads completely executed
        std::cout << "Process Complete." << "\n";

        return 0;
}
```

The main difference here is that the C++ program is not automatically synchronized like the other two programs. There are also two other ways to handle threads in C++, passing a callable object instead of a function, or passing a pre-defined lambda expression. [6]

Speed is an important factor to consider, as threading is often implemented with a motivation to decrease program runtime. Without having any tremendous differences in the implementations of threading libraries, the speed follows the general speed of the three languages, with C++ being the fastest due to it being a compiled language with fewer hand-holding mechanisms, and Java and Python both running rather slow due to Python being interpreted at runtime and Java having massive amounts of processes running behind the scenes for error checking and more.

Deadlock is essentially when more than one thread needs a specific resource at the same time, but one of them is unable to access that resource, and so the program cannot continue

running. It is one of the foremost problems considered when discussing threads, but none of the languages discussed have specific tools to prevent deadlock, so there is no clear advantage between them in that respect.

Race conditions are similar to deadlock; they occur when two threads can access shared data at the same time and change that data. In that instance the results will be different depending on which thread modifies the data first, so that can cause major problems without proper scheduling. This issue is also more related to the concept of threading than which language is being used.

Predictability is one of the greatest challenges with multithreading, and one of the best ways to achieve deterministic results is by thread synchronization. This is simply when threads establish either a set order in which to execute, or have some other method of coordinating with each other to successfully run the program without causing unexpected results or errors. Synchronization is one of the greatest improvements in multithreading, and can also help fix issues with deadlock and race conditions, by preventing multiple threads from accessing important data at the same time.

Python and Java have basic synchronization built in to the `.join()` methods, but if the programs actually require a shared resource or are modifying shared data, it gets more complicated. In Java, the keyword `synchronized` is simply added to the declaration of the specific runnable object used in the implementation. In Python, there is a built-in Lock class within the Threading module with `.acquire()` and `.release()` methods to handle synchronization. [7] The process in C++ is quite similar to Python, with functions for acquiring lock, and waiting until other threads have finished accessing data, but it is less straightforward than in Python. [8]

# Conclusion

Most of the differences between the three languages are programmer preference, but there are a few key points to note. C++ will generally run faster, and has extensive functions for thread use and thread synchronization, making it a prime choice for intense, time-sensitive programs. Python usually seems the easiest to understand what the code is doing and also is very reliable. Java can sometimes be faster than Python, and is a great choice if the threading is related to a project involving multiple classes, but is more complicated to write and thus usually not as worthwhile for stand-alone programs.

# References


[1] Techopedia, (2017, June 20). Multithreading. Retrieved from https://www.techopedia.com/definition/24297/multithreading-computer-architecture

[2] Python Course, (2019). History of Python. Retrieved from https://www.python-course.eu/python3_history_and_philosophy.php

[3] Oracle, (2020). The History of Java Technology. Retrieved from https://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

[4] Nikhil Kumar, GeeksforGeeks, (2020). Multithreading in Python | Set 1. Retrieved from https://www.geeksforgeeks.org/multithreading-python-set-1/

[5] Mehak Narang, GeeksforGeeks, (2020). Multithreading in Java. Retrieved from https://www.geeksforgeeks.org/multithreading-in-java/

[6] Sayan Mahapatra, GeeksforGeeks, (2020). Multithreading in C++. Retrieved from https://www.geeksforgeeks.org/multithreading-in-cpp/

[7] Nikhil Kumar, GeeksforGeeks, (2020). Multithreading in Python | Set 2 (Synchronization). Retrieved from https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/

[8] Cppreference, (2020, April 22). std::condition_variable. Retrieved from https://en.cppreference.com/w/cpp/thread/condition_variable