A COMPARITIVE ANALYSIS OF THE C++, JAVA, AND PYTHON LANGUAGES

Elvis Foster
Keene State College
98 PUBLICATIONS **58** CITATIONS

A COMPARITIVE ANALYSIS OF THE C++, JAVA, AND PYTHON LANGUAGES

Keene State College

Project from course CS430 Principles of Programming Languages
First Draft April 9, 2014; Revised December 23, 2014

Abstract

Over the past two decades, we have observed an occurrence of dominance of the programming arena by C-based programming languages. We have also observed a heightened interest in Python over the past few years. This paper reports the result of a comparative study on three C-based languages, namely C++, Java, and Python. The criteria used for analysis are the standard programming language features and principles as covered in a typical course on Principles of Programming Languages. The paper advances through ten sections: Section 1 discusses the rationale for the study, an overview of each language, and an overview of the study. Section 2 looks at programming domain and paradigm for each language. Section 3 looks at each language based on standard evaluation criteria of readability, simplicity, orthogonality, portability, programming environment, and usage cost. Section 4 examines each language based on the translation process. Section 5 focuses on data types, variables, and support for abstraction. In section 6, the emphasis is placed on expressions and assignment statements. Section 7 looks at each language based support for standard control structures and subprograms. Section 8 looks at support for inheritance, polymorphism, and file processing. Section 9 focuses on exception handling. Finally, section 10 provides a summary and some concluding remarks.

Stephen J. Humer is a recent graduate of Keene State College, where he completed a Bachelor of Science (BS) degree in Computer Science, specializing in Software Engineering and Computer Networking. Currently, he is an IT Analyst at Liberty Mutual Insurance. Elvis C. Foster is Associate Professor of Computer Science at Keene State College, New Hampshire. He holds a Bachelor of Science (BSc) in Computer Science and Electronics, as well as a Doctor of Philosophy (PhD) in Computer Science from University of the West Indies, Mona Jamaica.

1. Preliminary Items

This section contains:

- Rationale for the Comparative Analysis
- Overview of Java and Significance of Language
- Overview of C++ and Significance of Language
- Overview of Python and Significance of Language
- Overview of Comparative Analysis

1.1 Rationale for the Comparative Analysis of the Three Programming Languages

The rationale for a comparative analysis between three programming languages, namely Java, C++, and Python, is to establish improved background for choosing appropriate languages, improved understanding of the significance of implementation, and improved learning of existing programming languages. As a result, someone with a career in software engineering work will be able to pick the right language for the right project based on an understanding of the underlying programming principles.

One way to develop an ability to choose and learn programming languages is to simply learn several different languages. Learning the first language is hard, with the next one often not being so hard and so on. One problem is that you may only be working with languages that are only slightly different from each other. Thus, they will only include a small part of the spectrum of concepts underlying modern programming languages. A second problem includes the time investment to learn new languages and how to apply them under certain programming constraints appropriately.

By comparing three prominent programming languages based on underlying concepts, we will be able to build a foundation about programming language paradigms and issues that provide a strong preparation for selecting and learning new languages effectively, regardless of how many languages are already known.

The languages we will be dissecting, Java, C++, and Python, were picked due partly to their popularity. Based on the TIOBE Index for April 2014, which is an indicator of the popularity of programming languages, Java ranks second, C++ ranks fourth and Python ranks eighth in popularity [TIOBE, 2014]. We may conclude that all three languages are prominent and as such will give us a strong opportunity to demonstrate a comparative analysis among well-known languages.

1.2 Overview of Java and Significance of Language

The Java programming language was originally developed by Sun Microsystems, which was initiated by James Gosling in June 1991 for use in one of his many set-top box projects. Sun released the first public implementation in 1995 as a core component of Sun Microsystems' Java platform (Java 1.0). It promised to be Write Once, Run Anywhere (WORA); providing no-cost run-times on popular platforms.

The original goal of Java was to be used in embedded consumer electronic appliances. In 1994, the team realized Java, then known as Oak, was perfect for the Internet. In 1995, when the language was officially renamed to Java, it was redesigned for developing Internet applications.

As of March 2014, the latest release of the Java Standard Edition is 8, with Oracle discussing hopeful plans for Java SE 9 to be released in 2016. However, the recommended version of Java for users, as of this writing, is Version 7 Update 51, which was released in January 2014. The widespread popularity of Java has led to multiple configurations to suite various types of platforms. For example, Java 2 Enterprise Edition (J2EE) was developed for Enterprise Applications and Java 2 Micro Edition (J2ME) was developed for Mobile Applications, which are both separate configurations from the Java 2 Standard Edition (J2SE).

Java is:

- **Object Oriented**: In Java, everything is an Object. Java can be easily extended as a result of the Object model.
- **Platform independent**: Unlike languages such as C++, when Java is compiled, it is not compiled into platform specific machine code, but rather into platform independent byte code. This byte code can be distributed to any system over the web and is then interpreted by the Java Virtual Machine (JVM) on whichever platform it is being run.
- **Simple**: Java is designed to be easy to learn.
- **Secure**: With Java's secure feature it allows for the development of virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

- **Architectural-neutral**: The Java compiler generates an architecture-neutral object file format which makes the compiled code executable on many processors, with the presence of a Java runtime system.
- **Portable**: Being architectural-neutral and having no implementation dependent aspects makes Java portable. The Java compiler is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust**: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded**: With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted**: Java byte code is translated to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an increment and light weight process.
- **High Performance**: With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed**: Java is designed for the distributed environment of the internet.
- **Dynamic**: Java is considered to be more dynamic than C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

One of the main reasons Java is so popular is due to its platform independence, allowing Java programs to be run on many different types of computers. Further, being a simple language makes it easy to learn for programmers. However, being inherently object-oriented and having a library of classes that provide commonly used utility functions, the Java API, allows the language to cover a wide spectrum of applications.

1.3 Overview of C++ and Significance of Language

The C++ programming language was originally developed by AT&T Bell Labs in Murray Hill, New Jersey, starting in 1979 by Bjarne Stroutstrup, as an enhancement of the C language. C++ is a superset of C, in that virtually any legal C program is a legal C++ program. It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.

C++ is currently used by hundreds of thousands of programmers in essentially every application domain. Some domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. The language is also widely used for teaching and research because it is clean enough for successful teaching of basic concepts. C++ has greatly influenced many other popular programming languages; most notably, Java.

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming. Unlike other languages, such as Java, complicated run-time libraries and virtual machines have not traditionally been required for C++. Compatibility with C libraries and traditional development tools is emphasized.

The significance of the language is largely due to it being extremely fast, supporting a variety of different coding styles, and giving the programmer low level access to the computer through being a middle-level language. When dealing with large projects, the object-oriented principles and C++ support is excellent for this.

1.4 Overview of Python and Significance of Language

The Python programming language was originally developed by Guido van Rossum in the late 1980s and early 1990s at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.

Python is currently one of the most popular dynamic programming languages, along with Perl, Tcl, PHP, and Ruby. Although it is often viewed as a scripting language, it is really a general purpose programming language along the lines of Lisp or Smalltalk. Today, Python is used by the likes of scientists writing applications for the world's fastest supercomputers to children learning it as a first programming language.

Python mainly differentiates from C-like languages, such as Java and C++ above, by dispensing braces altogether, along with trailing semicolons, and instead opts to use whitespace. The other major area where Python differs is in its use of dynamic typing. In C, variables must always be explicitly declared and given a specific type such as **int** or **double**. In Python, variables are simply names that refer to objects.

Python's feature highlights include:

- **Easy-to-learn**: Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.
- **Easy-to-read**: Python code is much more clearly defined and visible to the eyes.
- **Easy-to-maintain**: Python's success is that its source code is fairly easy-to-maintain.
- **A broad standard library**: One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on Unix, Windows and Macintosh.
- **Interactive Mode**: Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
- **Portable**: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable**: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customer their tools to be more efficient.
- **Databases**: Python provides interfaces to all major commercial databases.
- **GUI Programming**: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of Unix.
- **Scalable**: Python provides a better structure and support for large programs than shell scripting.

Python is important for software development because it gets a lot of different things right, and in a combination that no other language has seemingly done so far. It recognizes that you'll spend a lot more time reading code than writing it, and focuses on guiding developers to write readable code. Python also acknowledges that speed of development is important, so it has access to powerful constructs that avoid tedious repetition of code. By combining a powerful library with a user-friendly language, Python has been able to excel in a field with established giants like C++ and Java.

1.5 Overview of Comparative Analysis

The principal goal of this analysis is to introduce and compare the fundamental features and advanced features of three prominent programming languages — Java, C++, and Python — to establish favorability and future expectations of each language, with respect to one another.

The fundamental features we will cover are based on the following criteria:

- Programming Domain
- Programming Paradigm
- Readability
- Simplicity
- Orthogonally
- Portability
- Programming Environment
- Usage Cost

The advanced features will cover are based on the following criteria:

- Translation Process
- Data Types, Variables and Support for Abstraction
- Expressions and Assignment Statements
- Control Structures
- Subprograms
- Support for Inheritance
- Support for Polymorphism
- Support for File Processing
- Exception Handling

The conclusion about the suitability of the language and expectations of the future will be submitted from the perspective of a practitioner in the field of software development.

2. Programming Domain and Paradigm

In this section, we cover:

- Rationale for Understanding Programming Domains and Paradigms
- Programming Domain
- Programming Paradigm

2.1 Rationale for Understanding Programming Domains and Paradigms

By identifying what languages were designed for certain domains and paradigms, an appropriately useful language may be chosen for any given programming problem.

2.1.1 Programming Domains

Since computer science affects all aspects of business and life, there are different programming languages for different purposes. Among the programming domains identified are scientific applications, business applications, AI applications, systems programming, scripting languages and hybrid languages [Foster, 2014a, section 1.4].

2.1.2 Programming Paradigms

The programming paradigm of a language deals with the design characteristics of that language. A programming paradigm establishes the capabilities and styles of various programming languages. Among the programming paradigms of interest are procedural, object-oriented, rule-based and hybrid languages [Foster, 2014a, section 1.5].

2.2 Programming Domain

Java and C++ are understood hybrid languages due to their ability to support different purposes within programming. This is a common occurrence for popular languages due to their expansion and growth to deal with numerous programming challenges. For example, C++ and Java both contain simple data structures with the facility to represent very small and very large numbers, for use in scientific applications. Further, C++ and Java both support a wider range of data structures, for use in business applications.

One difference between Java and C++ is the use of C++ for large projects, such as operating system support. For example, Linux was written in C++ and Windows was partly written in C++. Other C/C++ supported operating systems include Palm OS, Symbian OS and QNX Neutrino [IBM, 2014]. Java does have some use in operating system design, notably JavaOS, however no project has been undertaken to the level of Windows, Mac OS or Unix-like systems.

The use of Python is largely as a scripting language, with such languages having non-procedural features and being widely used for Web programming. However, Python is also used in a wide range of non-scripting contexts. Python code can be packaged into standalone executable programs and Python interpreters are available for many operating systems. It is appropriate to call Python a hybrid language as a result, due to its ability to be used in business application settings as well as Web scripting.

Conclusively, all three programming languages are hybrid languages in the sense that they have uses in multiple programming contexts, otherwise known as domains. It is appropriate to remind someone in software development the distinguishing use of C++ for systems programming and the use of Python as a scripting language.

2.3 Programming Paradigm

Java is a multi-paradigm programming language that includes support for object-oriented, structured, imperative, and reflective programming among others. By object-oriented, computation is effected by sending messages to objects; objects have state and behavior. This is demonstrated by the adding of uppercase names in figure 2.3.1. By structured, programs have clean, nested control structures. This is demonstrated by the for-loop and if-statement structure in figure 2.3.2. By imperative, control flow is an explicit sequence of commands. This is demonstrated by the step-by-step computation in figure 2.3.3. By reflective, programs manipulate their own structures.

Figure 2.3.1: Object-Oriented Programming

```
result = []
for p in people {
    if p.name.length > 5 {
        result.add(p.name.toUpper);
    }
}
return result.sort;
```

Adopted from [Ray, 2014]

Figure 2.3.2: Structured Programming

```
result = [];
for i = 0; i < length(people); i++ {
    p = people[i];
    if length(p.name)) > 5 {
        addToList(result, toUpper(p.name));
    }
}
return sort(result);
```

Adopted from [Ray, 2014]

Figure 2.3.3: Imperative Programming

```
    result = []
    i = 0
start:
    numPeople = length(people)
    if i >= numPeople goto end
    p = people[i]
    nameLength = length(p.name)
    if nameLength <= 5 goto next
    upperName = toUpper(p.name)
    addToList(result, upperName)
next:
    i = i + 1
    goto start
end:
    return sort(result)
```

Adopted from [Ray, 2014]

C++ is a multi-paradigm programming language that includes support for procedural, functional, object-oriented and generic programming. By procedural, it contains imperative programming with procedure calls. Likewise, C++ and Java, both based on C, have similar paradigms.

Python is a multi-paradigm programming language that includes full support for object-oriented programming and structured programming. There are a number of language features which support functional programming and aspect-oriented programming as well, although only with limited support. By aspect-oriented, programs have cross cutting concerns applied transparently. This also makes Python a hybrid language based on its design characteristics.

Conclusively, all three languages are designed for multiple paradigms, which further allow them to easily handle programming problems within multiple domains. The slight differences in which domain and paradigm have more focus will lead a programmer to still prefer one language over another.

3. Criteria for Evaluating Programming Languages

The basic evaluation criteria to be employed in the analysis are:

- Readability
- Simplicity
- Orthogonality
- Portability
- Programming Environment
- Usage Cost

3.1 Readability

Readability includes clarity and consistency. Clarity relates to how readable the code is, such as whether there are cryptic keywords and constructs. Consistency relates to whether the rules are consistent or if there are many exceptions [Foster, 2014a, section 1.6].

Java has around 50 keywords, as shown in table 3.1.1, that are considered to be easy to remember for programmers due to the nature of their words and what they do. For a massively popular language, the readability of Java is highly impressive.

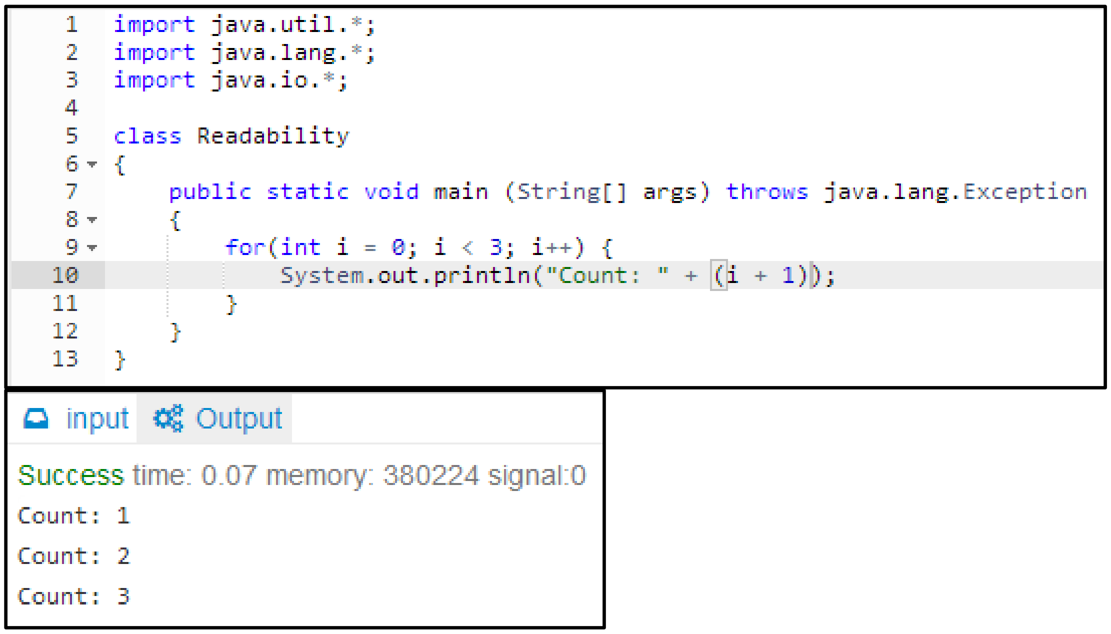
Table 3.1.1: Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Adopted from [Java, 2014]

Java has many rules that are consistent, such as coding conventions, dealing with exceptions, importing libraries, variable scope, and more. There are some exceptions, such as using the + symbol as an addition operator, for iteration, or for concatenation, as depicted in figure 3.1.1. However, Java is still considered a highly readable language.

Figure 3.1.1: Use of Java + Symbol



```java
import java.util.*;
import java.lang.*;
import java.io.*;

class Readability
{
    public static void main (String[] args) throws java.lang.Exception
    {
        for(int i = 0; i < 3; i++) {
            System.out.println("Count: " + (i + 1));
        }
    }
}
```

```
Success time: 0.07 memory: 380224 signal:0
Count: 1
Count: 2
Count: 3
```

C++ is known as a cryptic language, meaning its consistency is worse off than languages such as Java. C++ has 84 keywords, as shown in table 3.1.2, and contains many that are difficult to understand without a thorough understanding of the C++ language. Compared to Java and Python, C++ has the worst rating for readability.

Table 3.1.2: C++ Keywords

alignas	enum	return	alignof	explicit
short	and	export	signed	and_eq
extern	sizeof	asm	false	static
auto	float	static_assert	bitand	for
static_cast	bitor	friend	struct	bool
goto	switch	break	if	template
case	inline	this	catch	int
thread_local	char	long	throw	char16_t
mutable	true	char32_t	namespace	try
class	new	typedef	compl	noexcept
typeid	const	not	typename	constexpr
not_eq	union	const_cast	nullptr	unsigned
continue	operator	using	decltype	or
virtual	default	or_eq	void	delete
private	volatile	do	protected	wchar_t
double	public	while	dynamic_cast	register
xor	else	reinterpret_cast	xor_eq	

Adopted from [C++, 2013]

C++ is also known to have few rules, but several exceptions to these rules. For example, the asterisk symbol (*) could be used as a multiplication symbol, a pointer declaration, or as an indirection. The use of the * symbol as an indirection in figure 3.1.2 demonstrates why this reduces readability, as this is a separate distinction from a pointer declaration.

Figure 3.1.2: Use of C++ * symbol

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
   int n = 5;
   int *pn = &n;
   int **ppn = &pn;

   cout  << "Value of n:\n"
         << "direct value: " << n << endl
         << "indirect value: " << *pn << endl
         << "doubly indirect value: " << **ppn << endl
         << "address of n: " << pn << endl
         << "address of n via indirection: " << *ppn << endl;
}
```

Adopted from [C++, 2014]

Python, as compared to Java and C++, is specifically built to be more readable than both languages. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code. It is designed to have an uncluttered visual layout, frequently using English keywords where other languages use punctuation. The list of 35 keywords, as shown in table 3.1.3, demonstrates how simple a language can be in wording.

Table 3.1.3: Python Keywords

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	class
exec	in	raise	continue	finally
is	return	def	for	lambda
try	False	None	nonlocal	True

Adopted from [Python, 2014]

Furthermore, Python has a smaller number of syntactic exceptions and special cases compared to Java and C++. This makes Python the most readable language out of the three from an objective standpoint, with Java coming in second and C++ in last.

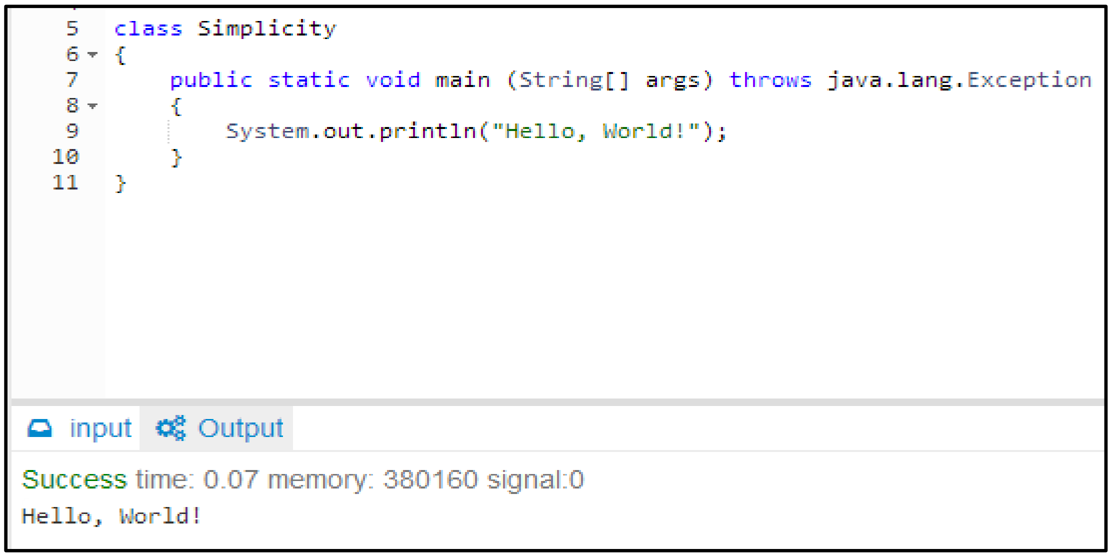
3.2 Simplicity

Simplicity affects the understandability of the language, and how easy it is to learn. A language that has a large number of basic components is more difficult to learn than one with fewer components. Multiplicity of features may provide flexibility, but can lead to increased difficulty in learning the language. Operator overloading, though powerful, is also potentially confusing [Foster, 2014a, section 1.6].

A look at the "Hello World" application for each language, which is a well-known example program that displays the message "Hello, World!" to the screen, is a good start in demonstrating how simple each language is.

In Java, as shown in figure 3.2.1, there is a three-part structure to print something to the screen. In the figure, **System.out** is a package that is shipped with the Java language, while **println** is a method responsible for outputting information to the console. The dot operator (.) is what allows a programmer to access properties (data and/or methods) of an object or class. Further, the parenthesis contains the data, with a string surrounded by quotes, and the line ending with a semicolon. For a novice programmer, this is a quite a lot to retain and understand for a simple print statement.

Figure 3.2.1: Java "Hello, World!" Program

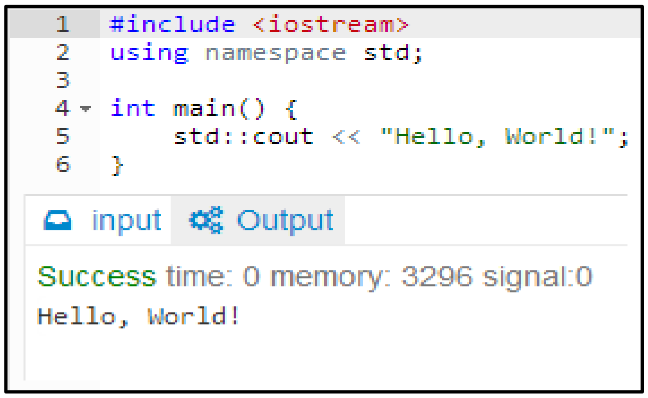


```
class Simplicity
{
    public static void main (String[] args) throws java.lang.Exception
    {
        System.out.println("Hello, World!");
    }
}
```

```
Success time: 0.07 memory: 380160 signal:0
Hello, World!
```

In C++, as shown in figure 3.2.2, there are different components to print to the console, and are likely to require similar understanding for a novice programmer. The portion "**std::cout**" identifies the standard character output device, which is usually the computer screen. The insertion operator (<<), indicates what follows is inserted into **std::cout**. The sentence within quotes, embedded in the parenthesis, is the content inserted into the standard output. The statement must also end with a semicolon.

Figure 3.2.2: C++ "Hello, World!" Program

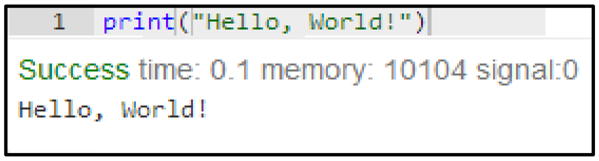


```
#include <iostream>
using namespace std;

int main() {
    std::cout << "Hello, World!";
}
```

```
Success time: 0 memory: 3296 signal:0
Hello, World!
```

In Python, as shown in figure 3.2.3, the print command is much simpler, making the life of the programmer much easier. However, Python presents a problem in that there are two popular versions of the language, and sometimes changes present simplicity issues. A look at figure 3.2.4 shows the same "Hello World!" application in Python 2.x, while figure 3.2.3 showed the application in Python 3.x. Overall, changes like this between Python 2.x and Python 3.x are few and far between and do not affect simplicity.

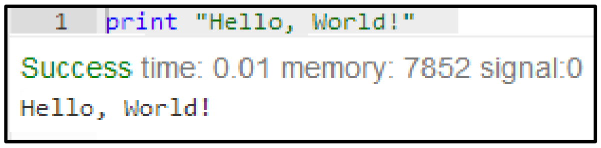
Figure 3.2.3: Python 3.x "Hello, World!" Program



```
print("Hello, World!")
```

```
Success time: 0.1 memory: 10104 signal:0
Hello, World!
```

Figure 3.2.4: Python 2.x "Hello, World!" Program

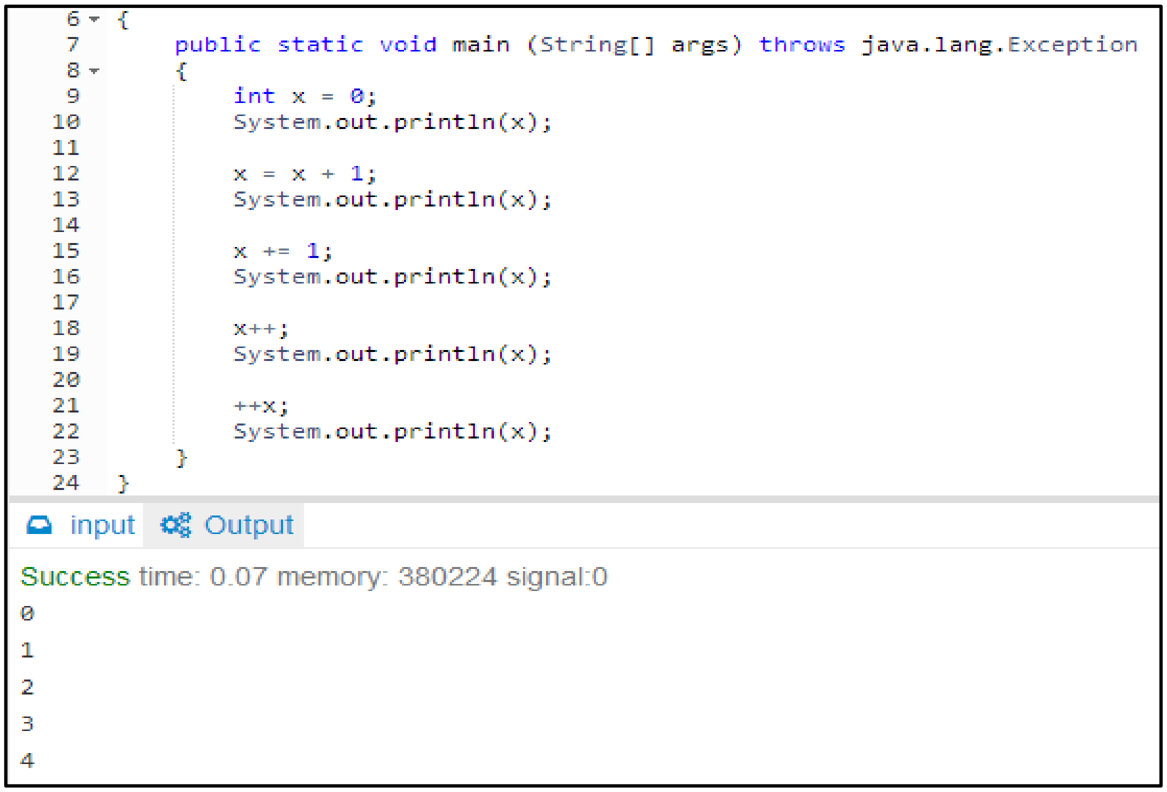


```
print "Hello, World!"
```

```
Success time: 0.01 memory: 7852 signal:0
Hello, World!
```

On the other side of simplicity, multiplicity of features, there is another distinct difference between Python, and the two C-based languages in the study. For example, in C++ and Java, you can increment a variable in four ways: x = x + 1; x += 1; x++; ++x. This is demonstrated in figure 3.2.5 for Java, and reduces simplicity.

Figure 3.2.5: Java Incrementing



```
{
    public static void main (String[] args) throws java.lang.Exception
    {
        int x = 0;
        System.out.println(x);

        x = x + 1;
        System.out.println(x);

        x += 1;
        System.out.println(x);

        x++;
        System.out.println(x);

        ++x;
        System.out.println(x);
    }
}
```

```
Success time: 0.07 memory: 380224 signal:0
0
1
2
3
4
```

While these variations provide flexibility to the experienced programmer, they introduce learning complexities for the novice programmer. A program like Python, which has a philosophy supporting simplicity, does not allow as many multiplicity features for this reason.

3.3 Orthogonality

Orthogonality relates to primitive features, particularly data types, being independent of the context in which they can be used. If a language is orthogonal, as an example, each operator can be applied to each data type. This feature facilitates easier learning of the language [Foster, 2014a, section 1.6]. The more orthogonal a language is, the more language simplicity is improved, as there are fewer exceptions. An excellent example of orthogonality in a language is Structured Query Language (SQL); however, for the purpose of this paper, we will defer further discussion of SQL.

In Java, the public and static keywords do not interfere with each other, so they are both orthogonal. Further, there are multiple ways to solve a problem in Java, which lends itself to being predominantly non-orthogonal. For example, a programmer may use a **for-loop**, a **while-loop**, or a **do-while-loop** to solve the same problem.

In C-based languages, there are certain examples of non-orthogonality:

- Records can be returned from functions, but arrays cannot (except via pointers).
- A member of a structure can have any data-type except void or a structure of the same type.
- An array element can be any data-type except void or a function.
- Parameters are passed by value, unless they are arrays, in which they are passed by reference.
- The expression $a + b$ usually means that numeric elements are added; if the elements are strings or the first element is a pointer, the behavior is different.

The language C++ exhibits a significant deviation from this norm in respect of the feature of operator overloading. C++ allows the programmer to overload most of its operators (with exceptions being **new**, **delete**, →, the comma, and the indirection operator). This feature makes C++ a particularly powerful languages that experienced programmers are enamored by; moreover, when operator overloading of this sort is applied, it can be argued that C++ is more orthogonal than typically meets the eye.

In Python, the language is known to be syntactically orthogonal, but semantically not. There are many ways to do things in Python, which can serve as unnecessary complexity or added options, depending on one's perspective.

The programming community appears to entertain various opinions on whether or not languages such as Java, C++, and Python are to be considered orthogonal. Overall, there is no definitive answer to this question. As a general observation, each of the three languages (Java, C++, and Python) has been expanded to handle several types of programming problems, as portrayed in this paper, and is likely to have several non-orthogonal features. That said, at the time of this investigation, C++ and Python do appear to exhibit more orthogonality than Java.

3.4 Portability

Portability refers to the ability of the language to work in different environments. The pre-requirement for portability is the generalized abstraction between the application logic and system interfaces. When software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction.

Java is touted as a massively cross platform language due to its Java Virtual Machine (JVM) and ability to produce portable applications and applets. Java provides three distinct types of portability: source code portability, CPU architecture portability, and OS/GUI portability.

Source code portability is the simplest and most familiar form of portability, and a given Java program should produce identical results regardless of the underlying CPU, operating system, or Java compiler. Languages such as C++ have used this idea, but C++ also provides numerous opportunities to create non-portable code as well. Unless programs written in C++ are designed to be portable from the beginning, the ability to move to different machines is more theoretical than practical. The semantics in C++ are looser, while the semantics in Java are more rigorous and leaves less up to the implementer.

CPU architecture portability is where Java produces object code, called J-code, for a CPU that does not yet exist, where most compilers produce object code that runs on one family of CPU, or at least for one CPU type at a time. For each real CPU on which Java programs are intended to run, a Java interpreter, or virtual machine, executes the J-code. This non-existent CPU allows the same object code to run on any CPU for which a Java interpreter exists.

OS/GUI portability is where Java provides a set of library functions that talk to an imaginary OS and imaginary GUI. Every Java implementation provides libraries implementing this virtual OS/GUI. This is unlike languages like C++ that have to deal with different operating system and GUI API calls. Thus, it is easy to conclude that Java has powerful portability capability [Java, 2014a].

C++ is still a very portable language due to its availability on all major operating systems from the numerous compilers that have been developed due to its popularity. However, not all target compilers use the same standard; for instance, there is C++98, C++03, C++TR1 or C++11. Depending on what standard the code is written in, with newer standards having less compliance with the range of compilers, the portability may change. When GUI comes to question, there are cross-platform options (such as QT), but it may be wise to keep the UI platform-specific depending on the situation, due to the GUI API calls mentioned before [C++, 2014a].

Python also has excellent portability, with stock Python offering a portable set of bindings to support portable GUIs across Unix, MacOS, and Windows. There are numerous third-party Python libraries that can be packaged together to enhance the portability of a programming language. For example, Portable Python [Python, 2014a] allows a user to run Python from a USB storage device to have a portable programming environment.

3.5 Programming Environment

The programming environment of a language is important for a programmer to comfortably use the language, and there are many options. Specifically, a programming environment is checked against usefulness as well as user-friendly features such as debugging for the programmer. Whether the language is interpretive, which is more user-friendly, or compiler-based, which is more efficient, is also a point of interest [Foster, 2014, section 1.6].

For Java programming environments, there are two stand-out integrated development environments (IDEs), NetBeans and Eclipse, which are industry standards. Eclipse allows a programmer to start a Java program in Debug mode, and has a special Debug perspective which gives the programmer a preconfigured set of views. In this perspective the programmer can control the execution process of the program and can investigate the state of the variables by setting breakpoints. Further, a program can trace errors by printing debugging messages to standard output when in Debug mode. This uses the **Plugin.isDebugging** and **Platform.getDebugOption** methods. NetBeans [Java, 2014b] packs a visual editor which makes GUI development a straight-forward process, and saves a programmer from unnecessary coding (Java, 2014c]). The NetBeans IDE provides all the features available in Eclipse, while throwing in some additional features.

Finally, Java compiles source code into an intermediate language, expressed generically as "byte code", using a just-in-time (JIT) compiler. This is due to the portability of Java and is one of the reasons it is able to run on any hardware. The Java Virtual Machine (JVM) interprets the compiled .class (byte code) file and converts it into machine specific instructions. JIT compilation has made the Java VM competitive in terms of performance when compared to natively compiled code.

For C++ programming environments, NetBeans and Eclipse are both compatible IDEs due to their extensibility and plugin support, although the downsides of how bloated they are still plays a factor. Fortunately, the useful debugging and visual editing features are available using C++ when extended. Visual Studio is another option for C++ programmers, although it is costly, while NetBeans and Eclipse are both free. Conclusively, the popularity of Java and C++ have produced programming environments that have plenty of support, arguably too much support.

Unlike Java, the C++ language does not use a JIT compiler and interpreter, but a specific compiler depending on the system being used and the IDE of choice. As mentioned in section 3.4 on portability of systems, there are multiple standards for target compilers. For example, Visual Studio uses the Visual Studio C++ compiler, while some recommend learning C++ using the GNU GCC compiler. The choice of compilation and functionality of an IDE comes down to user preference, as there are many options.

In similar fashion, Python has many programming environments, including NetBeans, Eclipse, and Visual Studio support through the use of plugins. Another upcoming IDE, PyCharm, provides quick code navigation, code completion, refactoring, unit testing and a debugger. There is an edition that fully supports Web development with Django, Flask, Mako and Web2Py and allows developing remotely. The complexity of such an IDE is beyond the scope of a new developer, but may be appropriate and useful for an intermediate or experienced programmer. Overall, the differences in IDEs between the three languages is small and is likely similar for any other popular language.

For compilation, Python is compiled to bytecode and then interpreted in a virtual machine, similar to Java, as reiterated in the translation process section. If the Python compiler is able to write out the bytecode into a .pyc file, it will usually do so. However, there is no explicit compilation step in Python as there is with Java or C++. Python also offers an interactive prompt where you can type Python statements and have them executed immediately. This means that the workflow in Python is much more similar to that of an interpreted language than that of a compiled language. To many developers, this distinction of workflow is more important than whether there is an intermediate bytecode step, and lends itself as a better environment to work in.

3.6 Usage Cost

Usage cost refers to the cost of languages based on several constituents [Foster, 2014a, section 1.6]:

- **Training Cost**: This is significantly increased if the language is difficult to learn.
- **Development Cost**: If it is difficult to use the language, it is undesirable.
- **Compilation Cost**: If the compiler is not efficient, this creates a system drag.
- **Program Execution Cost**: If the language requires many run-time checks, this could inhibit efficient execution.
- **Marketing Cost**: Languages that are free or relatively inexpensive will attract more takers, especially if the product is good.
- **Unreliability Cost**: Failure of a system due to language unreliability can be quite costly.
- **Maintenance Cost**: If the language is difficult to learn and use, this will affect the maintainability of systems developed with the language.

Much of the usage cost can be developed from a sense of the previous criteria for evaluating programming languages. As discussed in section 3.1, readability, and section 3.2, simplicity, Python is considered the most understandable and easy-to-learn language out of the three languages discussed. This in turn reduces the training cost, development cost and maintenance cost of the language. It was stated that Java is the more readable and simple language compared to C++, thus its training, development and maintenance cost would be less than that of C++. These factors contribute to enterprises and businesses that choose languages to use to develop software solutions.

Compilation cost may affect the languages, but due to the nature of the well-written compilers available for Java, C++ and Python, there is a trivial amount of overhead compared to more important issues, such as runtime performance overhead. It is important for a C++ programmer to be aware of the compilers that work with the standards of the programming environment and platform they are using, although this will also make negligible difference.

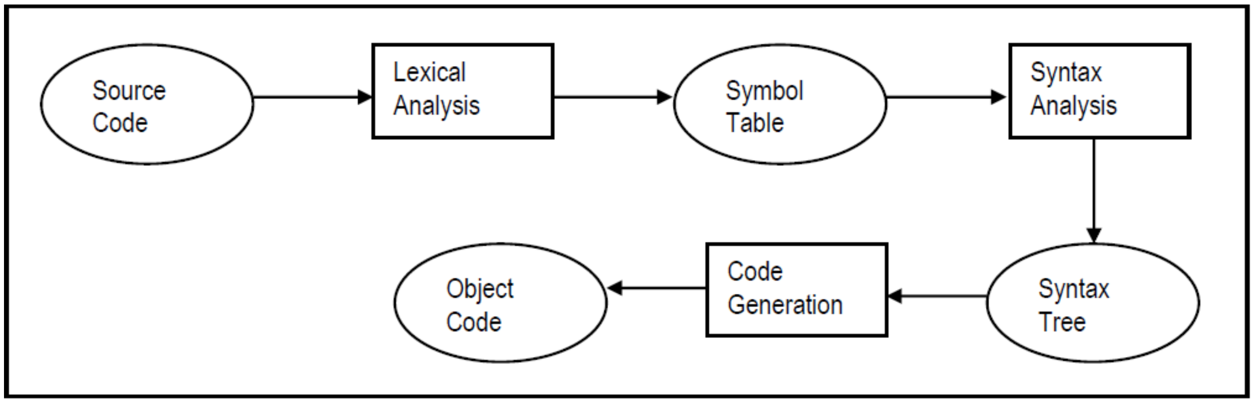
Programming execution cost may be affected by how strongly-typed a language is. The Python language is strongly typed as the interpreter keeps track of all variable types. This is done by not applying types to variables until after something has been assigned to that variable. Coding practices like this reduce execution cost and errors, since the data types do not need to be checked.

As for marketing cost, Java, C++, and Python are all free languages that have large communities that produce free programming environments, namely the IDEs or compilers, which developers may use. Overall, the marketing cost of these languages has minor differences and the free nature has further led to their popularity growth.

4. Translation Process

The translation process, or compilation process, is the series of stages that the program passes through in order to be converted from source code to object code. The translation process (as discussed in [Foster, 2014a, section 1.3] and [Foster, 2014b]) is outlined in figure 4.1; it involves three stages, namely, lexical analysis, syntax analysis, and code generation. All programming languages implement this translation process, but not necessarily in identical ways; understanding how a language implements this process therefore provides useful insight into the internal workings of the language.

Figure 4.1: Translation Process



Adopted from [Foster, 2014b]

The figure depicts the conversion of instructions written in a high-level language program to machine readable instructions for implementation. In lexical analysis, the source code is converted to a sequence of characters and terminal symbols from the character set of the language. All symbols identified must be defined by the grammar of the language. These symbols are loaded into the symbol table, which contains critical information relating to identifiers, data items, subprograms and other program components. An example of symbol table contents is depicted in figure 4.2.

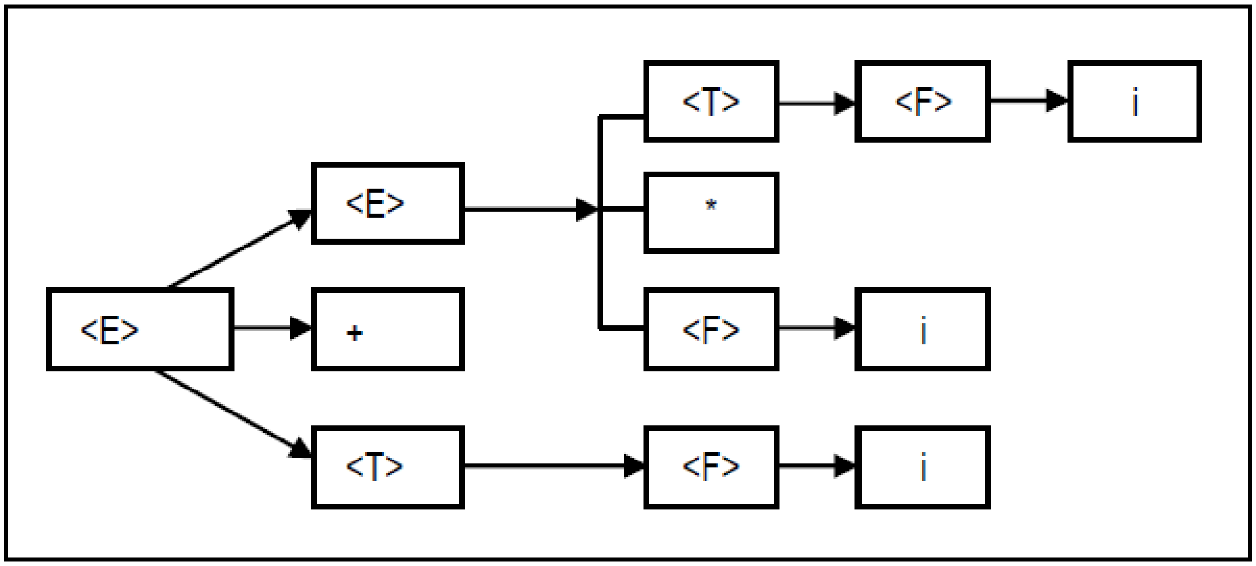
Figure 4.2: Example Symbol Table Contents

Name	Object Type	Description	Address	No. Of Bytes
Add	Prog Name	Program Name	604A0	
a	Variable	Integer variable	604C1	4
b	Variable	Integer variable	604C5	4
Sum	Variable	Integer variable	604C9	4
. . .				

Adopted from [Foster, 2014b]

In syntax analysis, the syntax tree — a hierarchical representation of the source code — is produced based on the syntax of the language. The analysis determines whether the sequence of symbols, given a certain grammar, belongs to the language. A finite state machine (FSM) is useful to test the validity of an input string, but only for regular grammars. A deterministic finite state machine (DFSM) may be an alternative to a syntax tree. A derivation tree is depicted in figure 4.3.

Figure 4.3: Example Derivation Tree Contents



Adopted from [Foster, 2014b]

The output of the lexical analysis is the input to the syntax analysis, and the output of the syntax analysis is the input of the code generation.

The process may occur via a single pass, or multiple passes. Multi-pass compilers typically use two or more passes to convert the source code to optimized object code. In single-pass compilation, code analysis and code generation are done in the same phase. In two-pass compilation, code analysis is typically done in the first phase and code generation in the second phase. In three-pass compilation, two approaches are prevalent. In the first approach, pass one is used for source code analysis, pass two for initial code generation, and pass three for code optimization. In the second approach, pass one is used for source code analysis, pass two for the generation of an intermediate code, and pass three for the generation of the final optimized code [Foster, 2014b].

The Java programming language makes use of three-pass compilation process by generating an intermediate code before generating the final optimized code. The compiler (more precisely the interpreter since Java is interpretive) would allocate the memory location during the semantic analysis (considered part of the syntax analysis phase of figure 4.1), leading to all declarations being processed before their use. The benefit of being a multi-pass compilation is machine independence, since the passes can be reused for different hardware and machines, and more expressive languages [Java, 2014d]. This ties in well with understanding Java's focus on portability and hardware independence.

Translation efficiency in Java depends to some extent on the IDE being used. Eclipse requires that all the components of a Java program be combined for compilation, while NetBeans allows the programmer to focus on specific components that have been changed since the last compilation. Both IDEs are interpretive, thus tracking down and reporting syntax errors early (on a line by line basis) and long before final compilation.

In C++, the full compilation process is a four-step process. The four steps are identified below. Note this is separate from whether the compiler uses multi-pass compilation.

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

One useful note of the compilation process for C++ is that you can stop the compilation of each source code file separately. The advantage of this is that you do not need to recompile everything if you only change a single file. On the topic of compiler passing, it depends on the compiler. The **.gcc** compiler does a single pass, whereas other compilers like those used by Visual Studio use two passes. The two passes would occur once for the preprocessor and then once for the compiler.

C++ IDEs tend to be compiler-based rather than interpretive. This means that syntax errors are typically not reported early, but delayed until the programmer actually compiles the code.

Python runs the advantage of interpreting code when a program first executes, then having the code compiled into bytecode. When next time the program is run, Python runs the bytecode, not the program file. Python only recreates the bytecode if you change the program file. Thus, Python combines the advantages of both interpretation and compilation. The Python translation process may also vary depending on the implementation of the language. The bytecode can be interpreted (CPython) or JIT compiled (PyPy). Like Java, Python IDEs tend to be interpretive.

5. Data Types, Variables and Support for Abstraction

This section looks at the three languages in terms of:

- Data Types and Variables
- Support for Abstraction

5.1. Data Types and Variables

Data types and variables are basic components typically featured in the syntax of a programming language. A data type describes a set of data values and a set of predefined operations that are applicable to the data items belonging to the set [Foster, 2014c, section 4.1]. A primitive data type is a data type that is not defined in terms of other data types, and is the building block for data representation in a programming language. Most programming languages support the following primitive data types: *integer*, *real number*, *Boolean*, *character*, and *string*. A variable is a name or identifier used to specify a data item. During the translation process, variables are translated to memory locations at execution-time [Foster, 2014c, sections 4.2 and 4.3]. In some languages, external names are restricted to a specific length or to specific character sets.

Java implements variations of the integer type (namely, **byte**, **short**, **int** and **long**) in its list of primitive data types: **byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, and **char** [Java, 2014b].

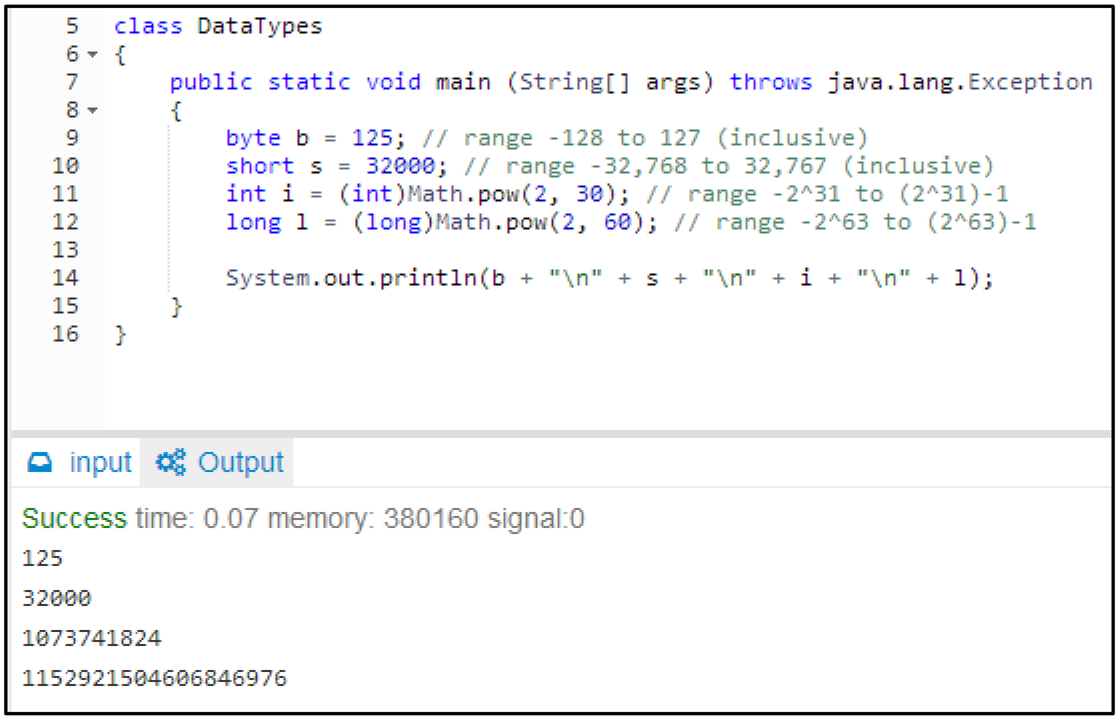
C++ implements a similar but slightly more expanded list of primitives: **int**, **unsigned int**, **short**, **unsigned short**, **long**, **unsigned long**, **float**, **double**, **long double**, **bool**, **char**, and **unsigned char** [C++, 2014b].

Python supports four different numerical types (**int**, **long**, **float**, and **complex**) in its list of primitive data types: **int**, **float**, **long**, **complex**, **bool**, **str**, **byte**, **list**, **tuple**, **set**, **frozen set**, and **dict** [Wikibooks, 2014]. Three things are worth noting from this list: Firstly, "str" represents the string data type; a string is a finite set of one or more characters (there is no char type). Secondly, the **byte** type in Python, represents a sequence of integers in the range of {0 . . 255}, and is only available in Python 3.x. Thirdly, the data types **byte**, **list**, **tuple**, **set**, **frozen set**, and **dict** are implemented in Python as primitive data types, whereas in most other languages, they are regarded as advanced data types. These data types allow the Python programmer to construct and manipulate various lists of items as needed.

These variations allow Java, C++, and Python to cater to the needs of scientific and business applications without using more scope than necessary. Further, Python does not require that the programmer declares the type on a variable. The declaration happens automatically when you assign a value to the variable. This feature is quite programmer-friendly, but can become the source of untold trouble for inexperienced programmers who may violate assignment consistency throughout their code.

Figure 5.1.1 demonstrates the range of the Java variations and the use of variables to store values. Figure 5.1.2 demonstrates the use of signed and unsigned variations in C++, with signed including negative numbers of the same range as unsigned data types. Figure 5.1.3 demonstrates number examples for each type in Python [Python, 2014b].

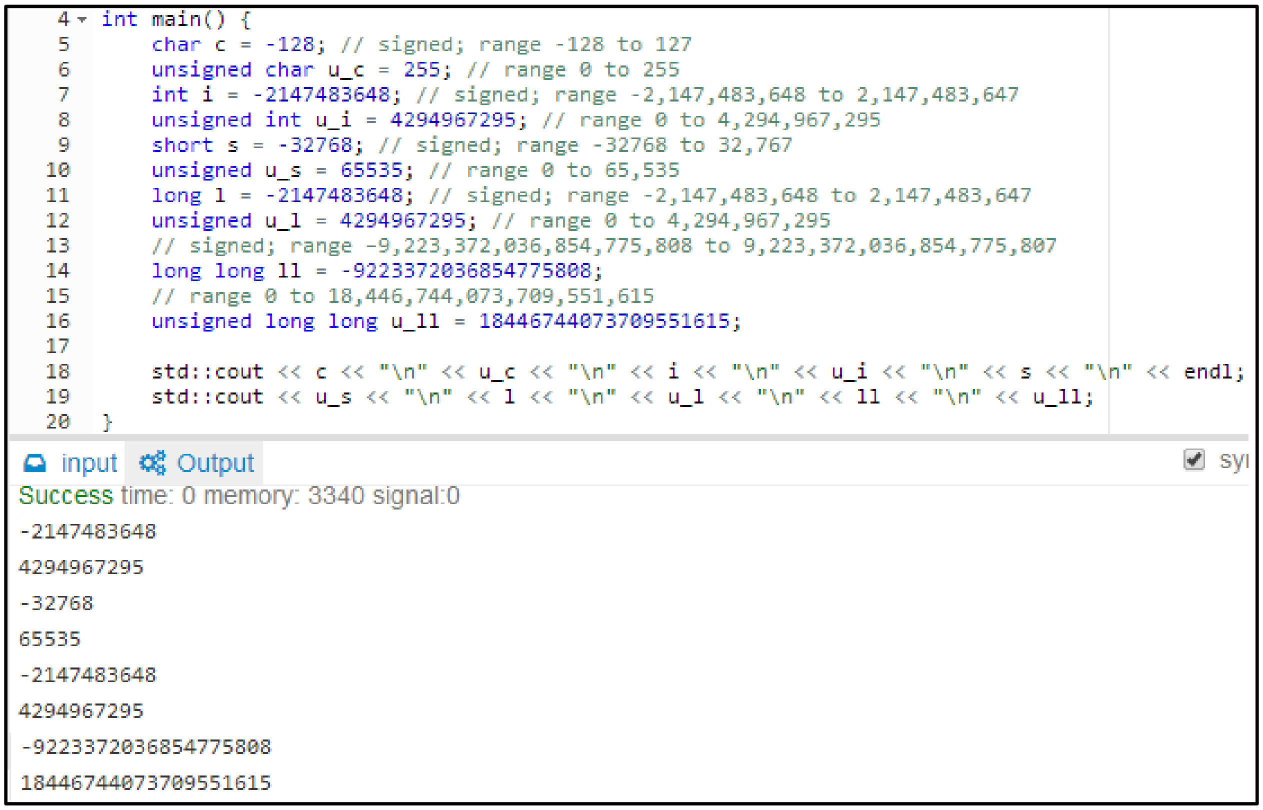
Figure 5.1.1: Java Integer Type Variations and Variable Use



```
class DataTypes
{
    public static void main (String[] args) throws java.lang.Exception
    {
        byte b = 125; // range -128 to 127 (inclusive)
        short s = 32000; // range -32,768 to 32,767 (inclusive)
        int i = (int)Math.pow(2, 30); // range -2^31 to (2^31)-1
        long l = (long)Math.pow(2, 60); // range -2^63 to (2^63)-1

        System.out.println(b + "\n" + s + "\n" + i + "\n" + l);
    }
}
```

```
Success time: 0.07 memory: 380160 signal:0
125
32000
1073741824
1152921504606846976
```

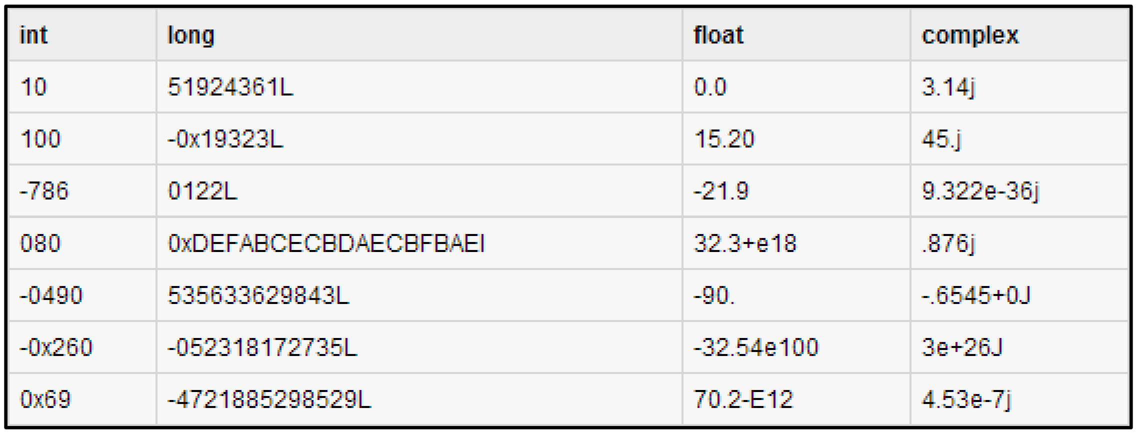
Figure 5.1.2: C++ Integer Type Variations and Variable Use



```
int main() {
    char c = -128; // signed; range -128 to 127
    unsigned char u_c = 255; // range 0 to 255
    int i = -2147483648; // signed; range -2,147,483,648 to 2,147,483,647
    unsigned int u_i = 4294967295; // range 0 to 4,294,967,295
    short s = -32768; // signed; range -32768 to 32,767
    unsigned u_s = 65535; // range 0 to 65,535
    long l = -2147483648; // signed; range -2,147,483,648 to 2,147,483,647
    unsigned u_l = 4294967295; // range 0 to 4,294,967,295
    // signed; range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    long long ll = -9223372036854775808;
    // range 0 to 18,446,744,073,709,551,615
    unsigned long long u_ll = 18446744073709551615;

    std::cout << c << "\n" << u_c << "\n" << i << "\n" << u_i << "\n" << s << "\n" << endl;
    std::cout << u_s << "\n" << l << "\n" << u_l << "\n" << ll << "\n" << u_ll;
}
```

```
Success time: 0 memory: 3340 signal:0
-2147483648
4294967295
-32768
65535
-2147483648
4294967295
-9223372036854775808
18446744073709551615
```

Figure 5.1.3: Python Numerical Types



int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEI	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Java implements real numbers as floats or doubles, with a float being a single-precision 32-bit floating point and a double being a double-precision 64-bit floating point. In similar fashion, C++ caters to both float and double real numbers as well, which is expected from two languages based on C. The Python language deviates from using a double and instead uses the complex data type shown from figure 5.1.3.

As mentioned previously, Python does not bind attributes to variables until runtime. While all three languages exhibit dynamic (late) binding (as opposed to static or early binding), Python implements this principle in the purest sense; this is why up-front variable declaration is not required in the language. Java is known as a strongly typed language, having each variable name associated with a single data type. C++ is not a strongly typed language as Java. Detecting data type violations early in the translation process tends to occur for strongly typed languages. Intuitively, this means that Python does not require detection, Java detects errors early, and C++ does not detect until further in the compilation process. This contributes late detection to an increased level of program execution errors and compilation cost.

In the area of programmer-defined data types, both C++ and Python support this feature; however, Java does so only in a limited sense. C++ allows the programmer to define structures, pointers, unions, static arrays, dynamic lists, vectors, linked lists, stacks, queues, graphs, etc. [Foster, 2014c, section 4.3]. Java allows the programmer to define arrays, sets, vectors, linked lists, stacks, queues, graphs, as well but not always in as flexible a manner as C++ [ibid]. However, it must be noted that the way Java implements arrays is superior to the implementation of static arrays in C++, but not as flexible as C++'s dynamic lists; Java allows the user to delay specifying the dimensions of an array at declaration-time, until it is time to create the array. C++ arrays are static; however, instead of using static arrays, one may opt for pointers, dynamic lists, or vectors. Java's response to the dynamic list in C++ is the array-list . Turning to Python, this language provides support of the main programmer-defined data types that C++ supports (structures, static arrays, dynamic lists, vectors, linked lists, stacks, queues, graphs, etc.). Pointers are not supported directly, but readily achieved via dynamic lists; moreover, Python gives the programmer the flexibility to create any ad-hoc data types ass deemed necessary.

Some languages allow data type conversions. Implicit conversions involve promotion from a lower range to a higher range (e.g. integer to real number). Implicit conversions work in C++ and Java through the compiler. Explicit conversions involve demotion from a higher range to a lower range, or conversion across apparently dissimilar ranges (e.g. object to string). Explicit conversions work in C++ and Java by casting [Foster, 2014d, section 5.4]. Python handles implicit and explicit conversions through the use of built-in functions. The functions return a new object representing the new value. The standard data-conversion function for each Python data type caries the same name as the intended data type, along with any related parameter(s). For instance, **str**(X) converts the Python object X to a string. There are a few additional data-conversion functions that do not directly map to a Python data type, but are included for convenience. For instance, **hex**(X) converts an integer to a hexadecimal string, and **oct**(X) converts an integer to an octal string.

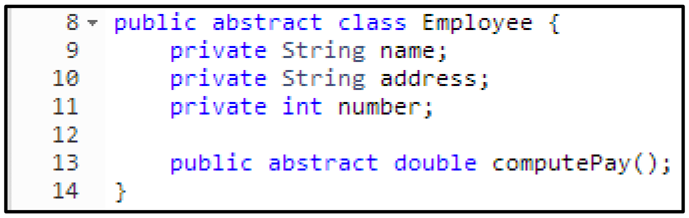
5.2. Support for Abstraction

Abstraction refers to the ability to create classes, structures, and/or any of the complex data types typically covered in a course on data structures and algorithms. It is not feasible to fully explore this feature of each language in this paper; suffice it to say that each language has its idiosyncratic way of supporting these abstract data types (ADTs). As an introduction to such an exploration (which is encouraged), we will briefly examine how each language in the study facilitates implementation of a class.

As you are no doubt aware, a class is an ADT that incorporates related properties in the form of data items (also called attributes or elements) and methods. An abstract class is a class that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class. If a class is abstract and cannot be instantiated, the class does not have much use unless it is a super-class. This is typically how abstract classes come to be in use. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

In Java, the **class** keyword in used to create a class; the basic syntax appears in [Foster, 2014g, section 8.2], [Foster, 2014k], and [Oracle, 2013]. To make the class abstract, the **abstract** keyword is used in the class declaration before the **class** keyword. Inside a parent class, methods may also be declared as abstract. The **abstract** keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body. An abstract method would have no definition, and its signature is followed by a semicolon, not curly braces. An example of this is shown in figure 5.2.1.

Figure 5.2.1: Java Abstract Class and Method



```java
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public abstract double computePay();
}
```

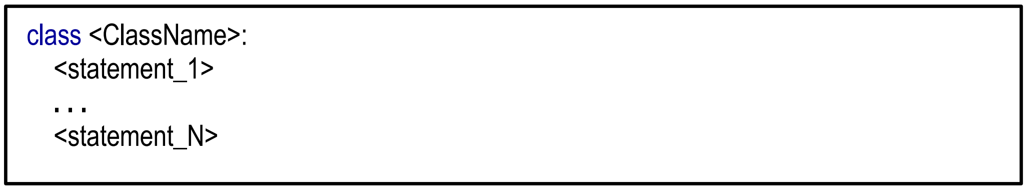
Declaring a method as abstract has two results:

- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare it abstract.

In C++, any program where you implement a class with public and private members is an example of data abstraction. C++ class definition involves use of the **class** keyword, but the construction is a bit more cryptic than in Java; the full syntax is clarified in [Foster, 2012a] as well as [Foster, 2014g, section 8.2]. Making a class abstract in C++ is achieved indirectly by making its components virtual, and this is done simply by using the **virtual** keyword in the declaration of such components.

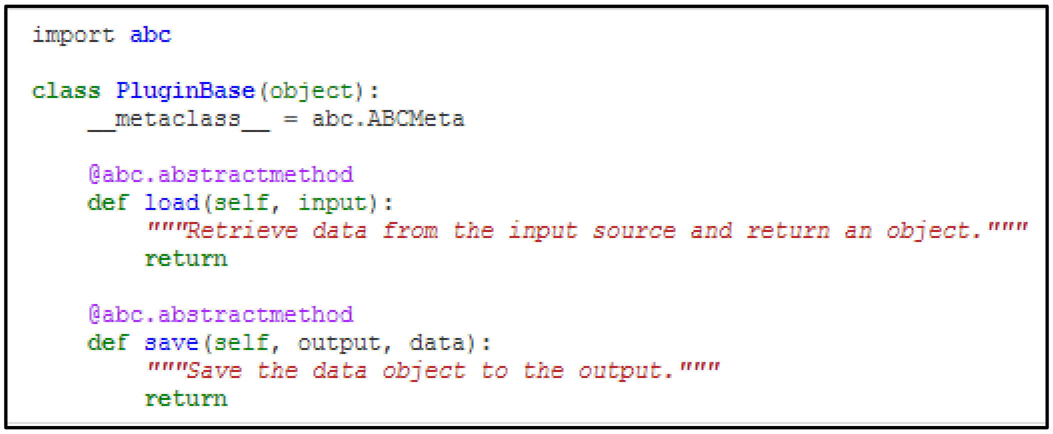
Python draws from C++ and Modula-3 in its implementation of classes. The basic syntax for class definition, which also involves the use of the **class** keyword, is shown in figure 5.2.2. In Python 2.6 and later, abstract base classes allow the programmer to specify methods that must be implemented in subclasses. Further, since Python, as well as C++, supports multiple inheritances, the language does not use interfaces, so and you would want to use base classes or abstract base classes. Figure 5.2.3 demonstrates how to implement abstract methods.

Figure 5.2.2: Basic Syntax for Python Class Definition



```
class <ClassName>:
   <statement_1>
   . . .
   <statement_N>
```

Figure 5.2.3: Python Abstract Methods



```
import abc

class PluginBase(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source and return an object."""
        return

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
        return
```

6. Expressions and Assignment Statements

This section contains:

- Expressions
- Assignment Statements

6.1 Expressions

Like data types and variables, expressions are basic components typically featured in the syntax of a programming language. Expressions are the building blocks for arithmetic and Boolean uses, as well as assignment statements. Expressions are influenced by operators, operator precedence, type mismatches, data coercion, and short circuit evaluations [Foster, 2014d].

Essentially, an expression is a, "construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluate to a single value" [Java, 2014f]. The C++ standard defines expressions as, "a sequence of operators and operands that specifies a computation." Examples include 42, 2 + 2, "Hello, World!" and func (“argument”), which all produce values. Figure 6.1.1 demonstrates an expression by utilizing the Backus-Naur Form (BNF) notation. This describes how an expression can be broken down into operators, variables and numbers to result in a single value.

Figure 6.1.1: BNF Notation of an Expression

```
<expression ::= <expression> + <term> | <expression> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= <primary> ^ <factor> | <primary>
<primary> ::= <primary> | <element>
<element> ::= (<expression>) | <variable> | <number>
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Arithmetic expressions are used for evaluating values that will be used in assignment statements or other Boolean checks. C-based languages have more operators compared to Pascal or Ada [Foster, 2014d]. The simplest is the assignment operator discussed below, with others including arithmetic operators, as seen here, compound assignment operators, also discussed below, relational operators, comparison operators, bitwise operators, comma operators and logical operators. C++ and Java also include an alternative ternary operator for conditions. Most operators are outside the scope of this discussion.

Languages such as C++ allow the programmer to overload most of the operators by creating operator functions so that they relate to complex objects. Indirectly, Java approaches the flexibility of C++ by allowing manipulation of all objects as instances of the Object class [Foster, 2014d].

Conditional expressions are used to evaluate a statement as true (a non-zero value) or false (0), such as inside an if-statement. The data type of the value returned by an expression depends on the elements used in the expression. Figure 6.1.2 shows examples of expressions, illustrated in bold.

Figure 6.1.2: Expression Examples

```
int cadence = 0; // cadence represents the value 0
anArray[0] = 100; // anArray[0] represents the value 100
// The + operator concatenates the inside of the println to form one string value.
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // arithmetic expression that represents the value 3
if (value == value2) // conditional expression that represents either 0 or non-zero
System.out.println("value1 == value2"); // no manipulation done is still an expression
```

Expressions are the same in Java, C++ and Python, in the sense that they are all parts of a statement that result in a value, although the syntax may change slightly depending on the language.

6.2 Assignment Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution, while an expression forms a portion of execution. In figure 6.1.2, the bold sections of the code were expressions, while each line of code is a statement. Thus, statements include things like variables and printing commands.

In C-based languages (such as Java and C++), an assignment statement has the following form:

```
<variable> = <expression>;
```

This statement changes the value of the variable on the left side of the equal sign, which is the assignment operator, to the value of the expression on the right-hand side. The variable is often just specified by a variable name, but there are also expressions that specify variables. Java treats an assignment as both an expression and as a statement. What officially makes this a statement is the use of a semicolon. As an expression, its value is the value assigned to the variable. This is done to allow multiple assignments in a single statement, such as:

```
a = b = 5;
```

By treating b = 5 as an expression with the value 5, Java makes sense of this statement by assigning 5 to a, as well as 5 to b. This is essentially a shortcut way of programming that may improve, or reduce, readability and simplicity depending on the viewpoint of the programmer. Further shortcuts can be found in using compound assignment operators. A compound assignment operator is an operator that performs a calculation and an assignment at the same time. All eleven Java and C++ binary arithmetic operators have equivalent compound assignment operators, as depicted in table 6.2.1.

Table 6.2.1: Compound Assignment Operators in Java and C++

Operator	Description
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Remainder and assignment
>>=	Signed right bit shift and assignment
<<=	Signed left bit shift and assignment
>>>=	Unsigned right bit shift and assignment
&=	Bitwise AND and assignment
^=	Bitwise exclusive OR and assignment
|=	Bitwise inclusive OR and assignment

Clarified in Python documentation, assignment statements are used to, "(re)bind names to values and to modify attributes or items of mutable objects." The BNF notation of a Python assignment statement is depicted in figure 6.2.1.

Figure 6.2.1: BNF Notation – Python Assignment Statement

```
assignment_stmt ::= (<target_list> =) + (<expression_list> | <yield_expression>)
target_list ::= <target>({, <target>}) [,]
target ::= <identifier> | (<target_list>) | "["<target_list>"]" | <attributeref> | <subscription> | <slicing>
expression_list ::= <expression> ({, <expression>}) [,]
yield_expression ::= yield [<expression_list>]
attributeref ::= <primary>.<identifier>
primary ::= <atom> | <attributeref> | <subscription> | <call>
atom ::= <identifier> | <literal> | <enclosure>
subscription ::= <primary> [<expression_list>]
slicing         ::=  <simple_slicing> | <extended_slicing>
simple_slicing   ::=  <primary> "[" <short_slice> "]"
extended_slicing ::=  <primary> "[" <slice_list> "]"
slice_list       ::=  <slice_item> ({"," <slice_item>}) [","]
slice_item       ::=  <expression> | <proper_slice> | <ellipsis>
proper_slice     ::=  <short_slice> | <long_slice>
short_slice      ::=  [<lower_bound>] : [<upper_bound>]
long_slice       ::=  <short_slice> : [<stride>]
lower_bound      ::=  <expression>
upper_bound      ::=  <expression>
stride           ::=  expression
ellipsis         ::=  …
call ::= <primary> ([<argument_list>] | <expression> genexpr_for)
// Note: expression and argument_list are also defined at [Python, 2014e]
```

Adopted from [Python, 2014e]

The Python language also uses only seven compound assignment operators, as opposed to the eleven mentioned in table 6.2.1 for Java and C++. The first five are the same as those in Java and C++. All seven operators for Python are depicted in table 6.2.2 below.

Table 6.2.2: Compound Assignment Operators in Python

Operator	Description
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Remainder and assignment
**=	exponent and assignment
//=	Floor division and assignment

Likewise, all three languages use the simple assignment operator as a single equal sign.

7. Control Structures & Subprograms

This section looks at the main programming control structures, and proceeds under the following captions:

- Importance of Structures
- Selection Structures
- Iteration Structures
- Recursion
- Subprograms

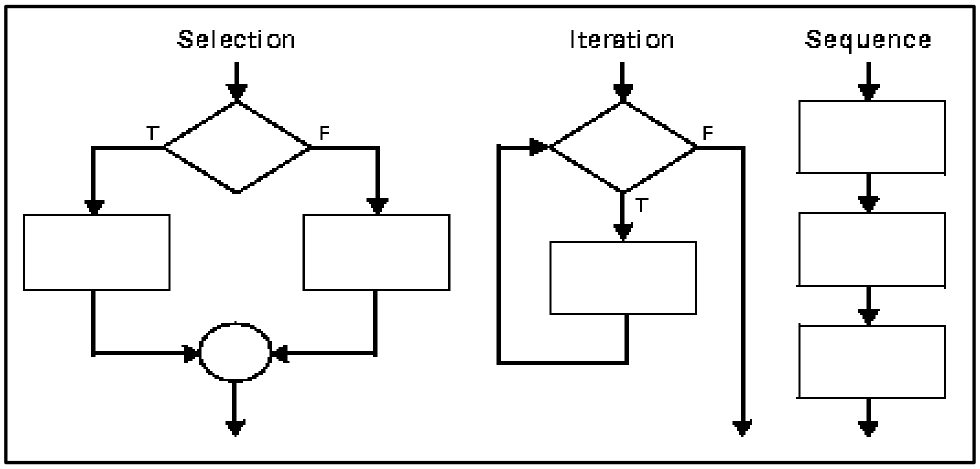
7.1 Importance of Structures

Control structures are the building blocks for constructing program logic. The fundamental control structures are as follows:

- Sequential structures
- Selection structures
- Iteration structures
- Recursion

A control structure is a block of programming that analyzes variables and chooses a direction in which to go based on given parameters. The term flow control details the direction the program takes. Hence, control structures represent the basic decision-making process in computing. Figure 7.1.1 shows the flow of the first three structures.

Figure 7.1.1: Selection, Iteration and Sequential Structures Flow



Adopted from [Oracle, 1999]

A real world example of the importance of a control structure is demonstrated in Figure 7.1.2. Similarly, programming situations must determine the proper course of action when running and manipulating data.

Figure 7.1.2: Control Structure Demonstration

Precondition: The vehicle is in motion.
Control Structure
    Is the traffic light green? If so, then the vehicle may stay in motion.
    Is the traffic light red? If so, then the vehicle must stop.
End of Control Structure
Postcondition: The vehicle comes to a stop.

The default condition, a sequential structure, executes instructions one after another. This may occur if only simple operations, such as carrying out a series of math equations, are necessary. Depending on the anticipated conditions, different structures may be used. These control structures may also be combined.

7.2 Selection Structures

Selection structures facilitate decisions based on certain pre-conditions. The If-Structure and Case-Structure are supported by most programming languages, but the syntax may vary. In C-based languages, the syntax remains roughly the same [Foster, 2014e, section 6.2]. The If-Structure may be a single-alternative selection, where a condition is evaluated and executed only if it is true, or a dual-alternative selection, that executes certain code if it is true and certain code if it is false. The Case-Structure may be viewed as a multiple-alternative selection, where multiple conditions are evaluated.

7.2.1 If-Structure

The If-Structure is the type of structure that would be implemented to handle the demonstration in figure 7.1.1. If something is true, execute, otherwise skip it or execute something else. A generic representation of an If-Structure is depicted in figure 7.2.1.

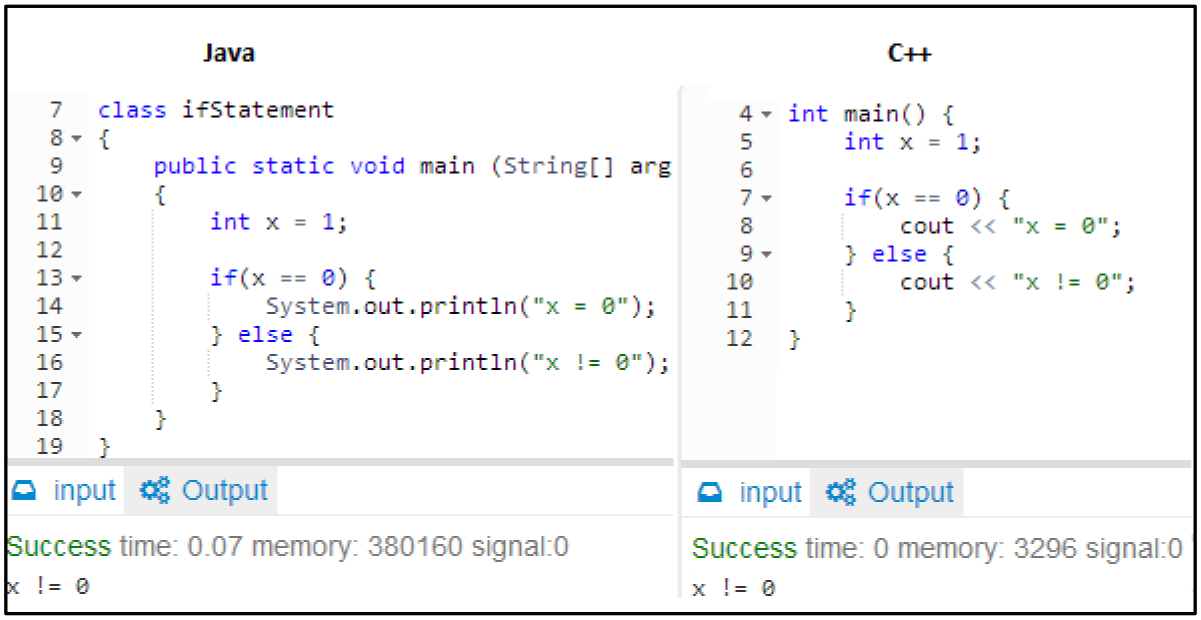
Figure 7.2.1: Generic Representation of If-Structure

```
If (condition)
    Statement(s);
[Else
    Statements;]
End-If;
```

Adopted from [Foster 2014e]

This representation proves accurate for C++ and Java, as depicted in figure 7.2.2.

Figure 7.2.2: C++ and Java Representation of If-Structure



Java

```
class ifStatement
{
    public static void main (String[] arg
    {
        int x = 1;

        if(x == 0) {
            System.out.println("x = 0");
        } else {
            System.out.println("x != 0");
        }
    }
}
```

```
Success time: 0.07 memory: 380160 signal:0
x != 0
```

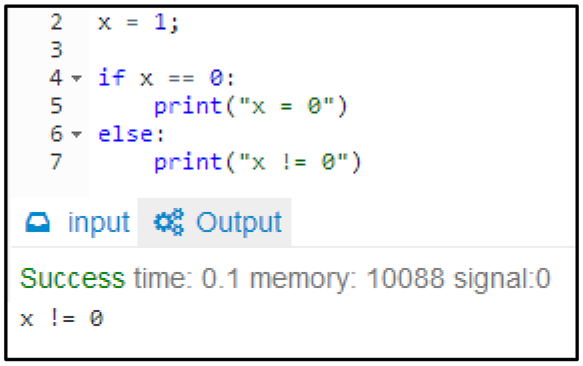
C++

```
int main() {
    int x = 1;

    if(x == 0) {
        cout << "x = 0";
    } else {
        cout << "x != 0";
    }
}
```

```
Success time: 0 memory: 3296 signal:0
x != 0
```

The Python syntax is slightly different, as python does not use parenthesis in conditional expressions and utilizes colons instead of brackets. Further, although all three examples showcase the use of whitespace indentation, only Python requires it. Figure 7.2.3 depicts the syntax for a Python If-Structure.

Figure 7.2.3: Python Representation of If-Structure



```
x = 1;

if x == 0:
    print("x = 0")
else:
    print("x != 0")
```

```
Success time: 0.1 memory: 10088 signal:0
x != 0
```

Multiple If-Structures may be used to check multiple conditions, and in some cases this is appropriate. When there are multiple conditions to check that warrant different statements to occur, a case structure may be used to improve readability and simplicity.

7.2.2 Case-Structure

A Case-Structure allows a programmer to perform certain actions when one of many conditions is met. The structure provides multiple alternatives. It inspects a value and chooses the path of execution that is assigned to that value. An example of a Case-Structure is the switch statement. Figure 7.2.4 represents a generic Case-Structure.

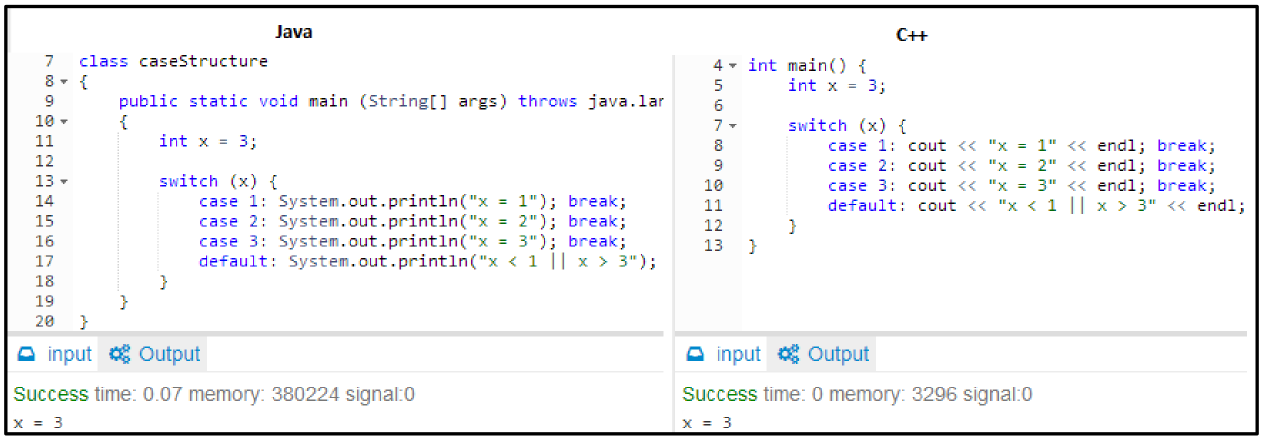
Figure 7.2.4: Generic Representation of Case-Structure

```
Case Variable | Expression is
Value-1: Statement(s);
Value-2: Statement(s);
…
Value-N: Statement(s);
Otherwise: Statement(s);
End-Case
```

Adopted from [Foster 2014e]

The representation of a case-structure in C++ and Java is depicted in figure 7.2.5. One difference between Java and C++ is that Java 7 allows string values to be checked as expressions of a switch statement, while C++ does not.

Figure 7.2.5: Java and C++ Representation of Case-Structure



```
class caseStructure
{
    public static void main (String[] args) throws java.lar
    {
        int x = 3;

        switch (x) {
            case 1: System.out.println("x = 1"); break;
            case 2: System.out.println("x = 2"); break;
            case 3: System.out.println("x = 3"); break;
            default: System.out.println("x < 1 || x > 3");
        }
    }
}
```

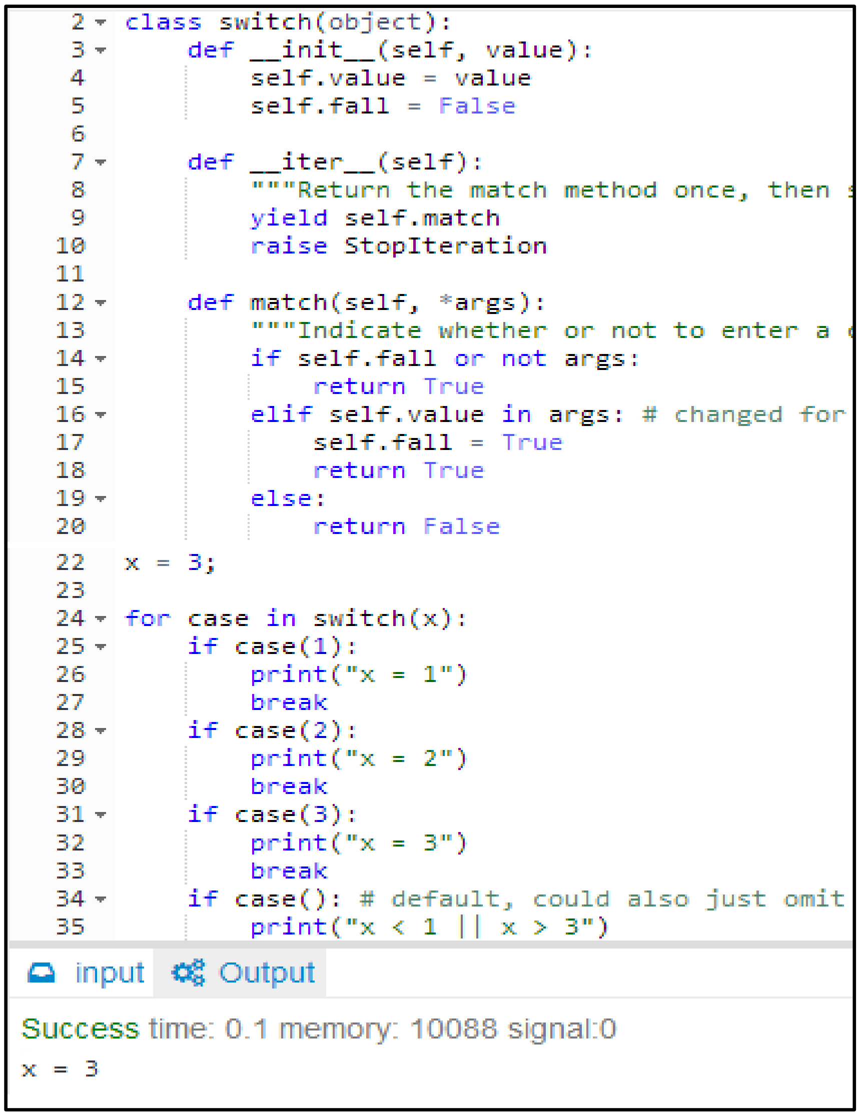
```
Success time: 0.07 memory: 380224 signal:0
x = 3
```

```
int main() {
    int x = 3;

    switch (x) {
        case 1: cout << "x = 1" << endl; break;
        case 2: cout << "x = 2" << endl; break;
        case 3: cout << "x = 3" << endl; break;
        default: cout << "x < 1 || x > 3" << endl;
    }
}
```

```
Success time: 0 memory: 3296 signal:0
x = 3
```

In Python, a Case-Structure requires a different approach, as there is no switch statement available to programmers. This has created discussions among the Python community, although alternatives have proven useful. One alternative is to create a Python class to mimic a switch statement. Once the class has been implemented, the syntax is relatively similar to C-based languages. This example is shown in figure 7.2.6.

Figure 7.2.6: Python Representation of Case-Structure



```
class switch(object):
    def __init__(self, value):
        self.value = value
        self.fall = False

    def __iter__(self):
        """Return the match method once, then s
        yield self.match
        raise StopIteration

    def match(self, *args):
        """Indicate whether or not to enter a c
        if self.fall or not args:
            return True
        elif self.value in args: # changed for
            self.fall = True
            return True
        else:
            return False
x = 3;

for case in switch(x):
    if case(1):
        print("x = 1")
        break
    if case(2):
        print("x = 2")
        break
    if case(3):
        print("x = 3")
        break
    if case(): # default, could also just omit
        print("x < 1 || x > 3")
```

```
Success time: 0.1 memory: 10088 signal:0
x = 3
```

7.3 Iteration Structures

The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. All imperative and OO languages have iteration structures. The common structures are the While-Structure, the Repeat-Until-Structure, and the For-Structure. Similar to selection structures, all C-based languages have similar syntax [Foster, 2014e, section 6.3]. Choosing which structure occurs based on whether there is a known amount of iterations, an unknown amount or knowledge that code must occur once.

7.3.1 While-Structure

The While-Structure continually executes a block of statements while a particular condition is true. The while statement evaluates an expression, which must return a Boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression valuates to false. This is useful when an unknown amount, from zero to numerous, of iterations are expected. A generic representation of a While-Structure is depicted in figure 7.3.1.

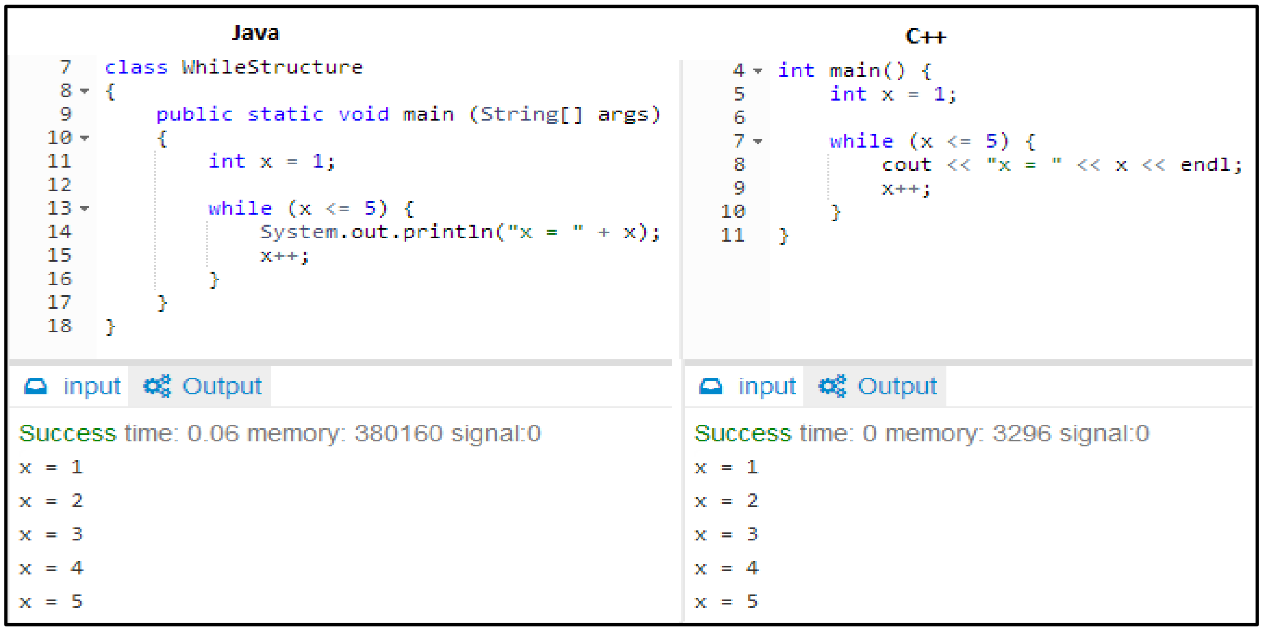
Figure 7.3.1: Generic Representation of While-Structure

```
While (condition) Do the following:
    Statement(s);
    …
End-While;
```

Adopted from [Foster 2014e]

Figure 7.3.2 demonstrates a While-Structure printing values 1 to 5 in Java and C++ while figure 7.3.3 demonstrates the same While-Structure in Python. As expected, the C-based languages have consistent syntax, while Python has slightly different syntax, but overall contains the same underlying structure.

Figure 7.3.2: Java and C++ Representation of While-Structure



Java

```java
class WhileStructure
{
    public static void main (String[] args)
    {
        int x = 1;

        while (x <= 5) {
            System.out.println("x = " + x);
            x++;
        }
    }
}
```

input Output

```
Success time: 0.06 memory: 380160 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

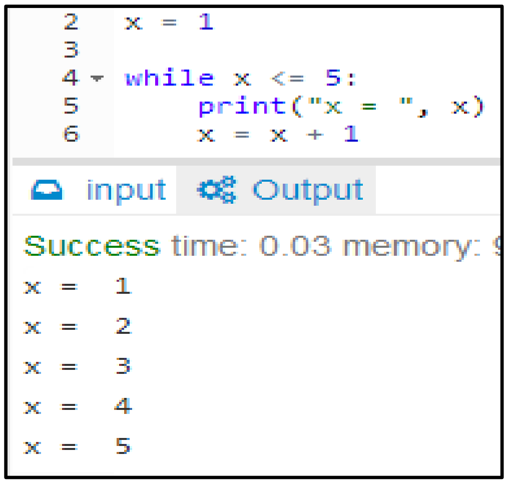
C++

```cpp
int main() {
    int x = 1;

    while (x <= 5) {
        cout << "x = " << x << endl;
        x++;
    }
}
```

input Output

```
Success time: 0 memory: 3296 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

Figure 7.3.3: Python Representation of While-Structure


```
x = 1

while x <= 5:
    print("x = ", x)
    x = x + 1
```

```
Success time: 0.03 memory: 9
x =  1
x =  2
x =  3
x =  4
x =  5
```

7.3.2 Repeat-Until-Structure

The difference between a While-Structure and Repeat-Until-Structure is that a Repeat-Until-Structure evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the **do-block** are always executed at least once. A generic representation of a Repeat-Until-Structure is depicted in figure 7.3.4.

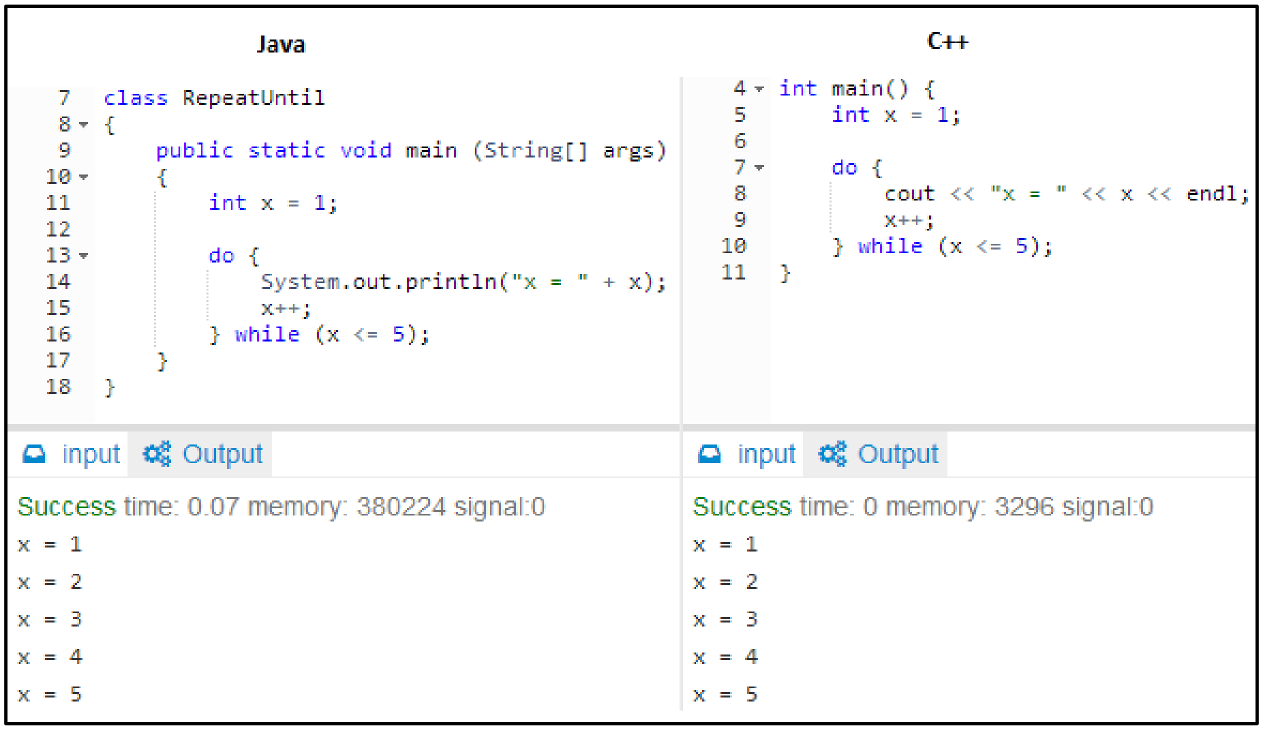
Figure 7.3.4: Generic Representation of Repeat-Until-Structure

```
Do the following:
   Statement(s);
   …
Until (condition);
```

Adopted from [Foster 2014e]

Figure 7.3.5 demonstrates a Repeat-Until-Structure printing values 1 to 5 in Java and C++ while figure 7.3.6 demonstrates the same Repeat-Until-Structure in Python. Python does not have a built-in Repeat-Until-Structure, but there are ways to mimic one by using a While loop. This includes utilizing the break statement to exit the loop once a condition is met.

Figure 7.3.5: Java and C++ Representation of Repeat-Until-Structure



Java

```
class RepeatUntil
{
    public static void main (String[] args)
    {
        int x = 1;

        do {
            System.out.println("x = " + x);
            x++;
        } while (x <= 5);
    }
}
```

input Output

```
Success time: 0.07 memory: 380224 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

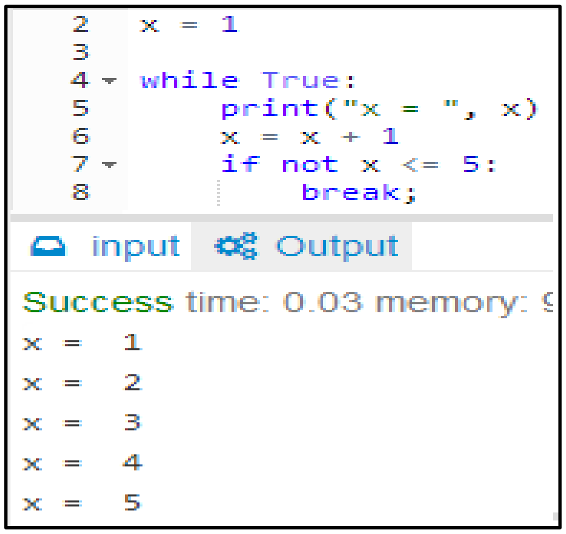
C++

```
int main() {
    int x = 1;

    do {
        cout << "x = " << x << endl;
        x++;
    } while (x <= 5);
}
```

input Output

```
Success time: 0 memory: 3296 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

Figure 7.3.6: Python Representation of Repeat-Until-Structure



```
x = 1

while True:
    print("x = ", x)
    x = x + 1
    if not x <= 5:
        break;
```

input Output

```
Success time: 0.03 memory: 9
x =  1
x =  2
x =  3
x =  4
x =  5
```

7.3.3 For-Structure

The For-Structure provides a compact way to iterate over a range of values. Programmers often refer to it as the **for-loop** because of the way in which it repeatedly loops until a particular condition is satisfied. The For-Structure is useful when the amount of iterations needed is known. A generic representation of a For-Structure is depicted in figure 7.3.7.

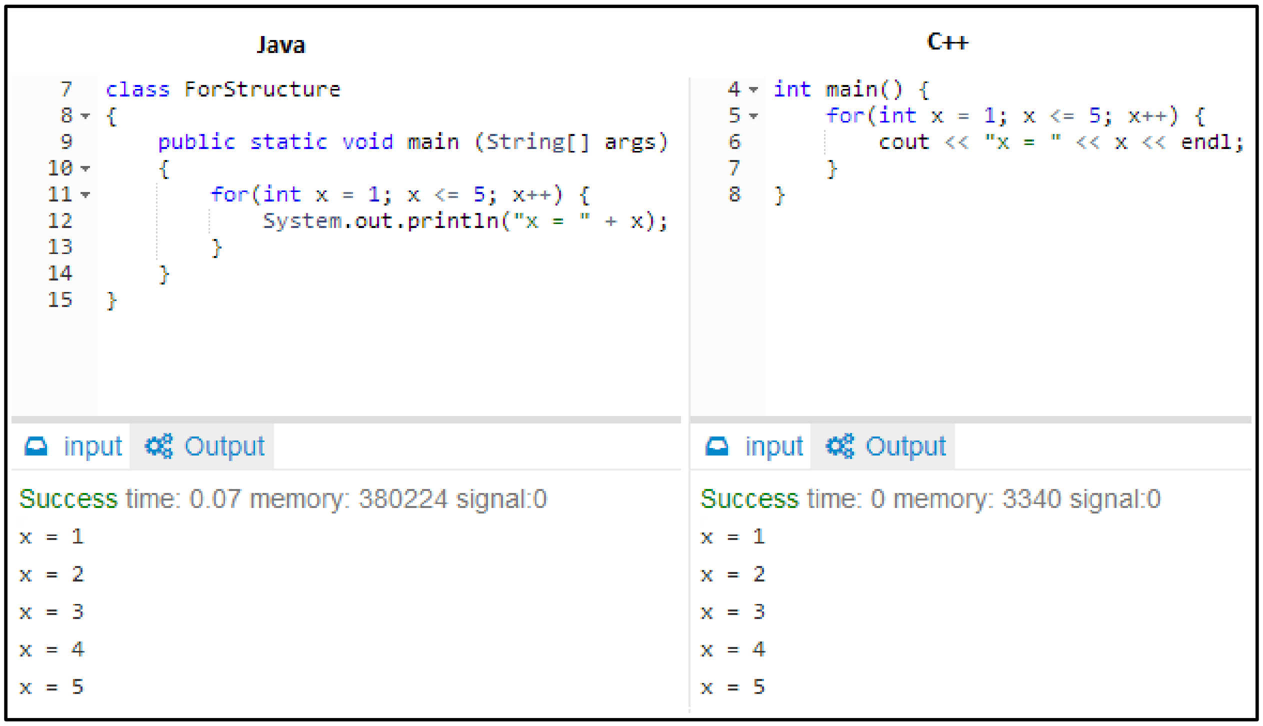
Figure 7.3.7: Generic Representation of For-Structure

```
For Variable := Value1 To Value2 with increments of Value3, Do the following:
    Statement(s);
    …
End-For;
```

Adopted from [Foster 2014k]

Figure 7.3.8 demonstrates a For-Structure printing values 1 to 5 in Java and C++ while figure 7.3.9 demonstrates the same For-Structure in Python. In Java, an enhanced for loop was introduced, mainly used for arrays. In Python, the For-Structure simplifies the loop conditions by using the range() method to iterate through a range of values.

Figure 7.3.8: Java and C++ Representation of For-Structure



Java

```java
class ForStructure
{
    public static void main (String[] args)
    {
        for(int x = 1; x <= 5; x++) {
            System.out.println("x = " + x);
        }
    }
}
```

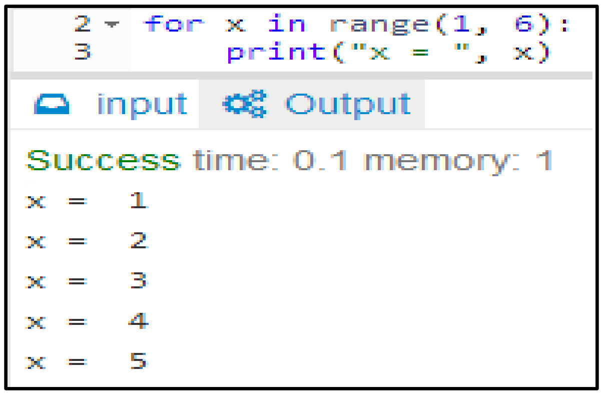
```
Success time: 0.07 memory: 380224 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

C++

```cpp
int main() {
    for(int x = 1; x <= 5; x++) {
        cout << "x = " << x << endl;
    }
}
```

```
Success time: 0 memory: 3340 signal:0
x = 1
x = 2
x = 3
x = 4
x = 5
```

Figure 7.3.9: Python Representation of For-Structure



```
for x in range(1, 6):
    print("x = ", x)
```

```
Success time: 0.1 memory: 1
x =  1
x =  2
x =  3
x =  4
x =  5
```

7.4 Recursion

Recursion is the ability of a subprogram to invoke itself. All contemporary programming languages support recursion [Foster, 2014f, section 7.4]. The main reasons to use recursion over iteration are that it is more intuitive in many cases when it mimics certain problems, and some data structures like trees are easier to explore using recursion.

A programmer must be able to detect whether an algorithm for certain problems can be developed easier through iteration or recursion before selecting the appropriate strategy. It is currently correct to say that everywhere recursion is used an iterative strategy could be used. The Fibonacci sequence is one example of a problem that may utilize a recursive solution effectively.

All three languages, Java, C++ and Python, have the ability to use recursion. Figures 7.4.1 and 7.4.2 depict all three languages solving a factorial problem by creating a method called factorial. It must be noted that all three solutions could have been solved iteratively. Notice the similarity in code, with the Python code being slightly terser. For all practical reasons, we may conclude that the languages are similar in their treatment of recursion.

Figure 7.4.1: Java and C++ Factorial Recursive Function

```
// Java Recursive Factorial Method
public static double Factorial(int n)
{
  double Result;
  if ((n == 1) || (n == 0)) Result = 1;
  else Result = n * Factorial(n - 1);
  return Result;
} // End of Factorial Method
```

```
// C++ Recursive version of N!
double Factorial (int n)
{
double Result;
if ((n==1) || (n == 0)) Result =1;
else Result = n * Factorial (n-1);
return Result;
}
```

Adopted from [Foster 2014f]

Figure 7.4.2: Python Factorial Recursive Function

```
def Factorial (n):
if n == 1 or n == 0:
  Result = 1.0
Result = n * Factorial (n – 1)
return Result
```

For completeness, figure 7.4.3 shows a generic factorial iterative function to compare.

Figure 7.4.3: Generic Factorial Iterative Function

```
Iterative (Variable n)
START
   Variable Fact := n;
   For Variable x := n - 1 To 1 with decrements of 1, Do the following:
      Fact := Fact * x;
   End-For
   Return Fact;
STOP
```

Adopted from [Foster 2014f]

7.5 Subprograms

Subprograms are the building blocks for structuring a program into readable components that are easy to follow. Further clarified, a subprogram is a component of a program that carries out a specific activity or set of related activities [Foster, 2014f, section 7.1]. The terminology for subprograms ranges from procedure, function, method to routine.

Subprograms exhibit the following properties:

- The subprogram has a single entry point
- On execution of the call, the calling program component is suspended, and control is transferred to the called subprogram. When the subprogram completes, control is returned to the calling statement.
- The subprogram may be defined with parameters, in which case it must be called with corresponding arguments.

A subprogram contains a heading and a body, with the heading containing the name of the subprogram, the return type, parameters, and in some cases, accessibility keywords.

Different languages employ different structures for subprograms, which must be factored into design issues and ease-of-use. Among the commonly known structures are the following [Foster, 2014f, section 7.2]:

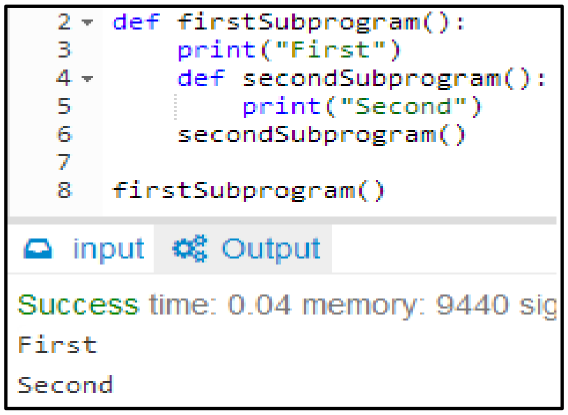
- **Separate Subprograms**: Subprograms are written in separate files.
- **Integrated Nested Subprograms**: Subprograms are part of the program file, and can be nested.
- **Integrated Independent Subprograms**: Subprograms are part of the program file, and cannot be nested.
- **Non-separate Subprograms**: Subprogram definitions are not separated from the main program.

In the Java and C++ languages, subprograms are integrated independent subprograms. The subprograms are part of the program file, and cannot be nested. However, Python implements subprograms as integrated nested subprograms. The subprograms are part of the program file, and can be nested. This difference is slight, but may allow for more flexibility for a programmer. It also may add confusion for a beginner, which reduces readability and simplicity of the language.

Referring back to figures 7.4.1 and 7.4.2, the subprograms are the factorial functions. In Java, the subprograms can be defined after usage, while in C++ and Python, the factorial subprogram had to be defined ahead of time. If this does not occur, the programmer will receive an error. It is essential that these design issues are understood before someone in the software development field attempts to implement subprograms.

A look at figure 7.5.1 depicts the usage of nested subprograms in Python.

Figure 7.5.1: Integrated Nested Subprograms in Python



```
def firstSubprogram():
    print("First")
    def secondSubprogram():
        print("Second")
    secondSubprogram()

firstSubprogram()
```

```
Success time: 0.04 memory: 9440 sig
First
Second
```

Parameter handling is another issue to discuss when a subprogram is called [Foster, 2014f, section 7.3]. The arguments supplied on the call are copied to their corresponding parameters in a positional manner. Care must be taken on the call to ensure that the supplied arguments are of the same data type as their corresponding parameters.

All major programming languages support pass-by-value, where the argument(s) supplied on the call are copied to the subprogram's parameters for internal use in the subprogram. Java and C++ also support pass-by-reference, where an access path to the address of each reference argument is provided on the call of the subprogram. As the subprogram executes, changes to its parameter(s) result in changes to the originally supplied argument(s) via the access path(s). Unlike Java and C++, Python utilizes the pass-by-assignment strategy, which is similar to pass-by-reference.

Further, most languages support overloading subprograms, including Java, C++ and Python. The compiler typically distinguishes among the choices based on parameters. A concise way to overload is to create a generic subprogram, which provides the algorithm without specifying the type of data used in the algorithm. Java, C++ and Python all support generics, but Python supports them by default without any special syntax.

8. Support for Inheritance, Polymorphism, and File Processing

In this section, we examine the following:

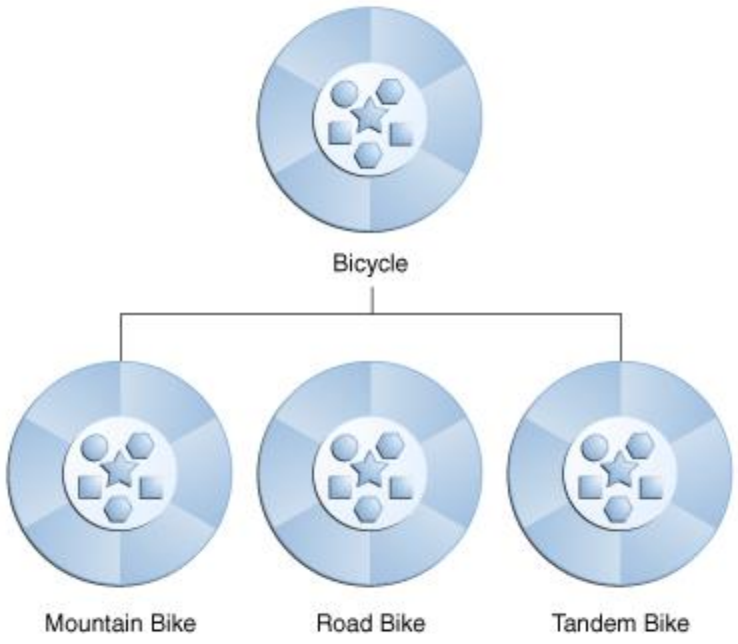
- Support for Inheritance
- Support for Polymorphism
- Support for File Processing

8.1 Support for Inheritance

Inheritance is the capacity of an object to adopt the features of its parent class on which it has been defined. It also relates to a class adopting the features of another class [Foster, 2014h, section 9.2]. Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles. Yet each also defines additional features that make them different. Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. Figure 8.1.1 depicts one superclass with several subclasses.

Inheritance makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and to create fast implementation time. In Java, single inheritance is supported but not multiple inheritances. However, through interfaces and abstract classes, the effect of multiple inheritances can be achieved, while eliminating the drawbacks. Java can block a method from being inheritable via allowing use of the keyword final. In C++, single inheritances and multiple inheritances are allowed. Keywords supported for both languages include public, private and protected.

Figure 8.1.1: A Hierarchy of Bicycle Classes



Adopted from [Java, 2014g]

In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses. The syntax for creating Java and C++ subclasses is compared in [Foster, 2014g, section 8.2]. A simplified illustration specific to the current scenario for a Java environment is depicted in figure 8.1.2. At the beginning of the class declaration, the **extends** keyword is used, followed by the name of the class to inherit from. This gives the subclass all the same fields and methods as the superclass, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read.

Figure 8.1.2: Basic Syntax for Creating a Java Subclass

```
class Bicycle
{
  // fields and methods
 . . .
}
 . . .

class MountainBike extends Bicycle
{
    // new fields and methods
    ...
}
```

In C++, the terminology used is that of a *base class* (superclass in Java) and *derived class* (subclass in Java). One significant difference is that a class can be derived from more than one class, which means it can inherit data and functions from multiple base classes. Java only allows data to be inherited from one base class. A simplified C++ implementation of the aforementioned scenario is depicted in figure 8.1.3. A programmer must be aware of the syntax changes from Java to C++. Also, C++ provides more flexibility in that multiple classes can be defined in the same program file but this is strictly prohibited in Java.

Figure 8.1.3: Simple Syntax for Creating a C++ Subclass

```
// Base class
class Bicycle
{
    public:
        void attributeOne()
        ...
};

// Derived class
class MountainBike: public Bicycle
{
    public:
        int derivedAttribute();
        ...
};
```

A derived class can access all the non-private members of its base class. The different access types according to who can access them are presented in figure 8.1.4.

Figure 8.1.4: C++ Access Types

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Adopted from [C++, 2014c]

Python also allows a programmer to similarly define two classes, with one being a subclass of another. A simple Python implementation for the aforementioned scenario is depicted in figure 8.1.5. Note the **Bicycle.__init__** call in the MountainBike class, which calls the superclass **__init__** method. All three languages support this calling, which makes it possible to use and define all values.

Figure 8.1.5: Basic Syntax for Creating a Python Subclass

```
class Bicycle(object):
    def __init__(self, type):
        self.type = type

    def getType(self):
        return self.type

class MountainBike(Bicycle):
    def __init__(self, type, color):
        Bicycle.__init__(self, “Mountain Bike”, “Red”)
```

Like C++ and unlike Java, Python also supports multiple inheritances. This means that a Python class can inherit from more than one super-class. No illustration is provided here, but this can be easily verified by consulting [Python 2014b].

8.2 Support for Polymorphism

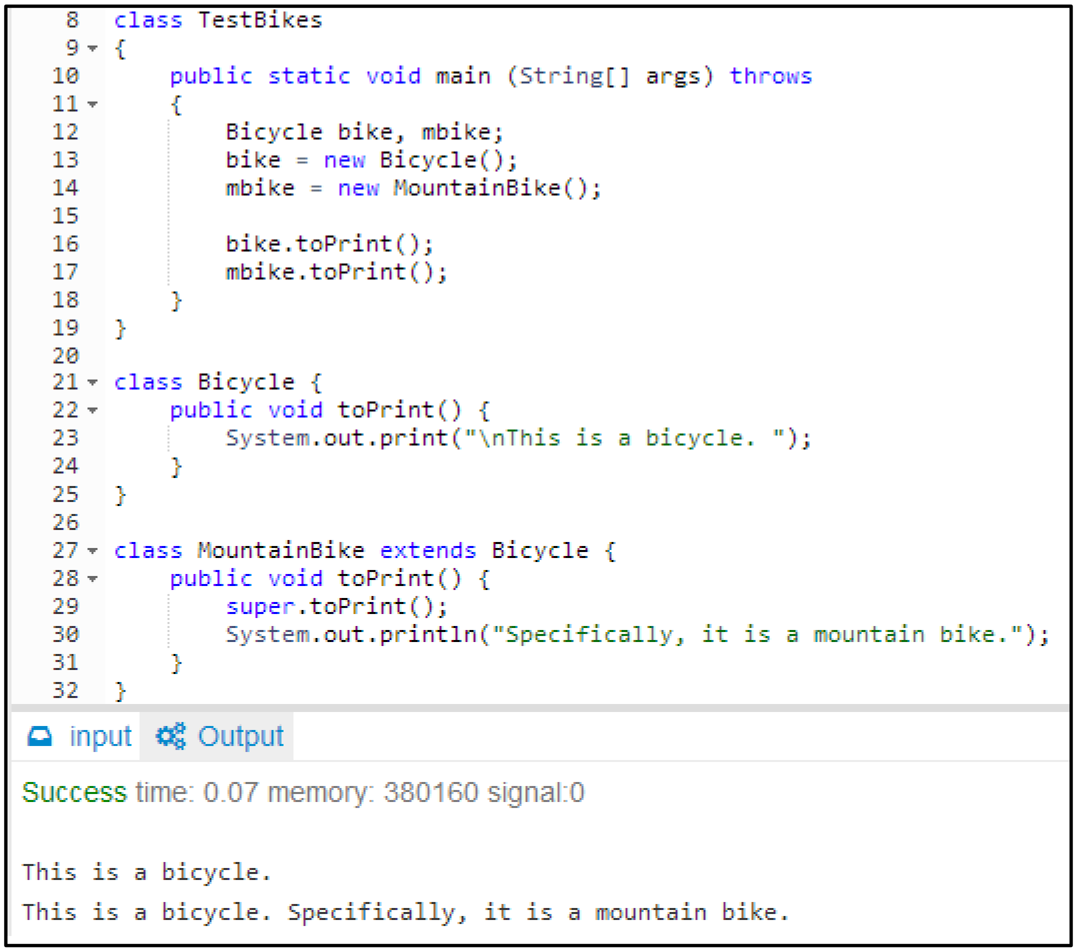
Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism occurs when a parent class reference is used to refer to a child class object. Java supports method overriding, method overloading, and limited operator overloading. Method overriding also extends to abstract classes and interfaces. Java also supports generic classes, this feature representing a higher level of polymorphism. In C++, function overloading, function overriding, operator overloading, and functions with default arguments are supported. C++ also supports template (i.e., generic) functions and classes, both features leading to advanced levels of polymorphism [Foster, 2014h]. A good way to test if an object is polymorphic is to see if it can pass more than one IS-A test. An IS-A test is described in figure 8.2.1.

Figure 8.2.1: Polymorphic IS-A Test

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

A Deer IS-A Animal
A Deer IS-A Vegetarian
A Deer IS-A Deer
A Deer IS-A Object
```

In Java, we can demonstrate polymorphism by modifying the Bicycle concept from figure 8.1.2. For example, a **toPrint()** method can created in the superclass and overridden in a subclass, and the JVM will print the appropriate **toPrint()** based on the object that is referred to in each variable. Figure 8.2.2 demonstrates how this may occur.

Figure 8.2.2: Java Polymorphism Example



```java
class TestBikes
{
    public static void main (String[] args) throws
    {
        Bicycle bike, mbike;
        bike = new Bicycle();
        mbike = new MountainBike();

        bike.toPrint();
        mbike.toPrint();
    }
}

class Bicycle {
    public void toPrint() {
        System.out.print("\nThis is a bicycle. ");
    }
}

class MountainBike extends Bicycle {
    public void toPrint() {
        super.toPrint();
        System.out.println("Specifically, it is a mountain bike.");
    }
}
```

```
Success time: 0.07 memory: 380160 signal:0

This is a bicycle.
This is a bicycle. Specifically, it is a mountain bike.
```

One of the key features of class inheritance in C++ is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature. Like Java, C++ is able to override methods and call the correct one. As this is similar to Java, no figure will be given. Refer to [C++, 2014d] for an example.

Further, Python is similar to Java and C++ in the sense that it too has support for polymorphism. To iterate once more, polymorphic behavior allows you to specify common methods in an "abstract' level, and implement them in particular instances. This will save the programmer time by allowing them to enhance the design of super- and subclasses when working with inheritance. Refer to [Python, 2014c] for an example.

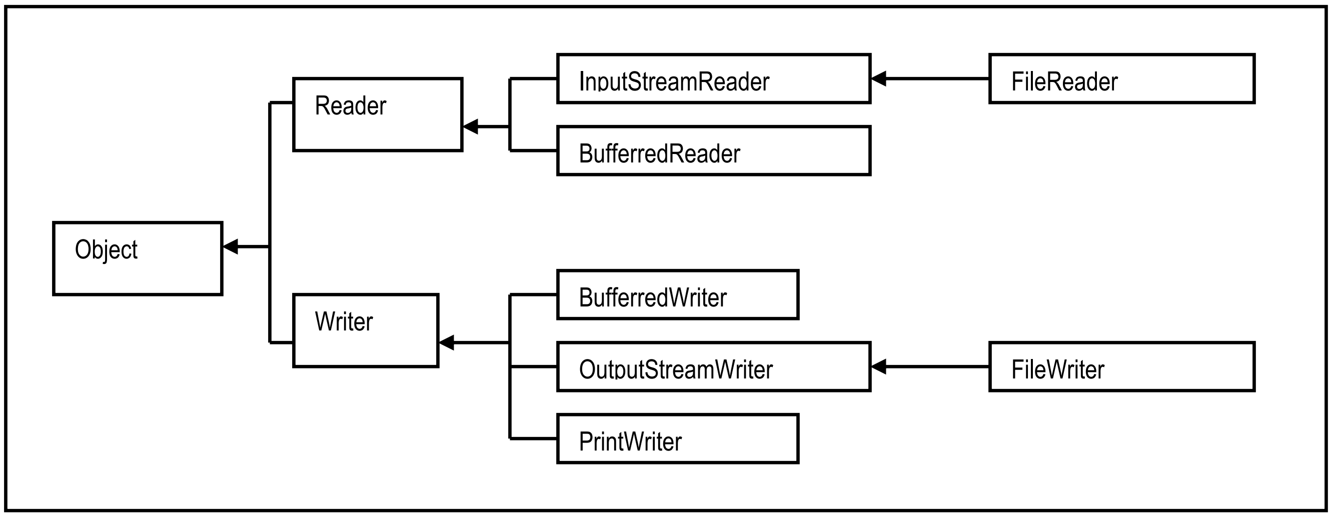
A programmer should be concerned with whether multiple inheritance should be supported, and what the negative side-effects are and how to avoid them?

8.3 Support for File Processing

Support for file processing is essential to reading and accepting data that will be used in a program. Full discussion of file processing, and the nuanced approaches in each language is beyond the scope of this paper. A basic introduction of file implementation in each language will suffice.

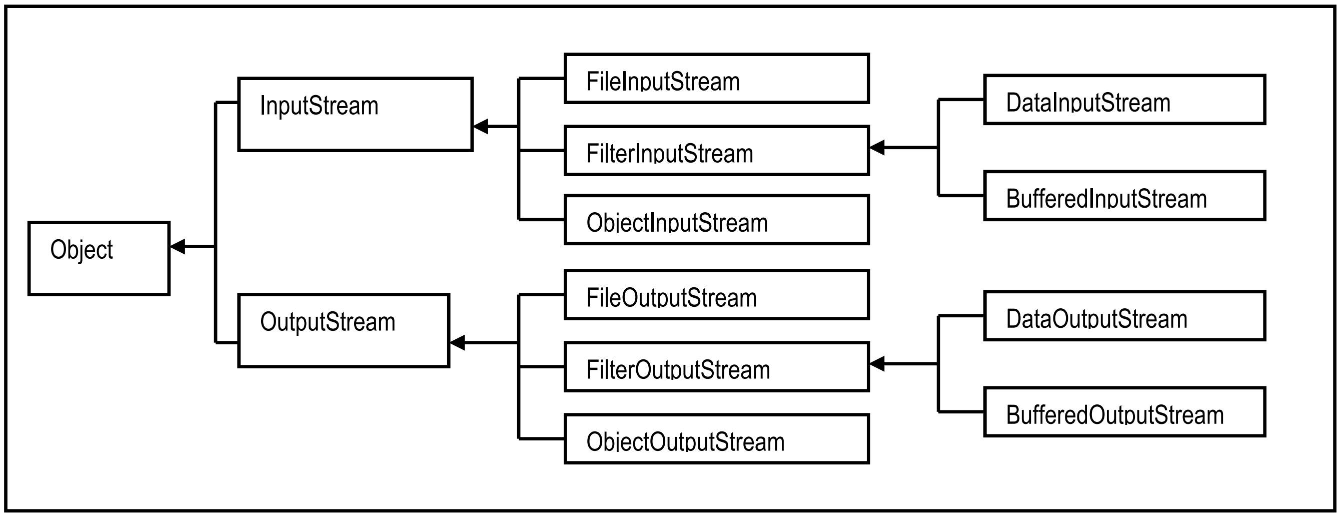
In Java, file processing is performed using various classes. Figures 8.3.1 and 8.3.2 show the Java class hierarchies for text files and binary files respectively. These are summarized in [Foster, 2014k] and more fully discussed in [Oracle, 2013]. One primary class used to handle files is called **File**, which is part of the **java.io package**. To use it, it must be imported into a project. Writing from a file, creating a file, checking if a file exists, opening a file and reading from a file are all likely to be expected needs. Figure 8.3.3 demonstrates some of these implementations.

Figure 2.40: Java Class Hierarchy for Text I/O



Adopted from [Foster, 2014k]

Figure 2.41: Java Class Hierarchy for Binary I/O



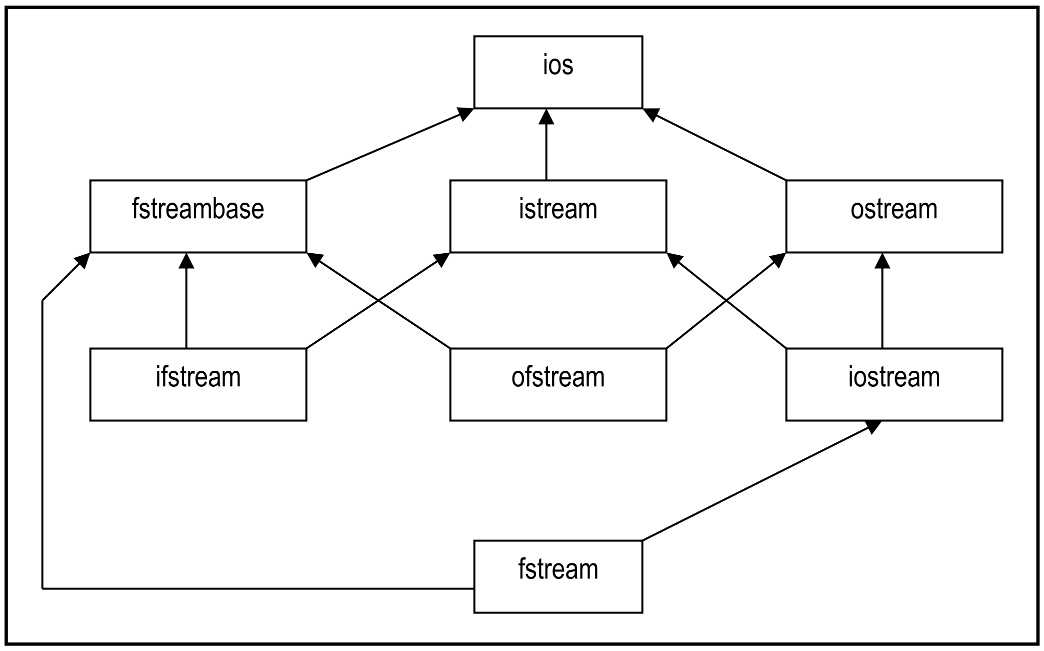
Adopted from [Foster, 2014k]

Figure 8.3.3: Java Support for File Processing

```
// The File class must be imported into any project dealing with file processing
import java.io.File;

// -- Create File --
// To create a new file, you must use the PrintWriter class.
import java.io.PrintWriter
// Indicate that you are planning to use a file
File fileExample = new File(“Example.xpl”);
// Create that file and prepare to write some values to it
PrintWriter pwInput = new PrintWriter(fileExample);

// -- Write File –
// Write a string, double-precision number or Boolean to the file
pwInput.println(“Line One”);
pwInput.println(true);
// Once done writing, close the PrintWriter object to de-allocate its memory
pwInput.close();

// -- Read File –
Scanner opnScanner = new Scanner (fileExample);
// Read each line in the file
while (opnScanner.hasNext()) {
   // Read each line and display its value
   System.out.println(“Line: “ + opnScanner.nextLine());
   System.out.println(“True/False: “ + opnScanner.nextLine());
}
// De-allocate the memory that was used by the scanner
opnScanner.close();
```

Similar to Java, C++ has a filing system consisting of several classes; this system is summarized in figure 8.3.4 and discussed in [Foster, 2012b]. In C++, file processing is performed using the **fstream** class. The **fstream** class is a complete C++ class with constructors, a destructor and overloaded operators. To perform file processing, you can declare an instance of an **fstream** object. If you do not yet know the name of the file you want to process, you can use the default constructor. The **fstream** class provides a method to write to a file and another to read. As Java has demonstrated reading and writing, a figure will not be supplied for C++. Please refer to [C++, 2014e] and [Foster, 2012b] for additional discussions and examples. There you will learn about file processing methods such as **open()**, **close()**, **write()**, **read()**, **flush()**, **seekg()**, **seekp()**, **tellg()**, **tellp()**, **sizeof()**,**rename()** and **remove()** methods.

Figure 8.3.4: C++ I/O System



Adopted from [Foster, 2012b]

In Python, the simplest way to produce output is using the **print** statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output. For keyboard input, Python provides the built-in functions **raw_input** and **input**. The **raw_input** function reads one line and returns it as a string. The **input** function is equivalent, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

When it comes to files in Python, the language provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object, similar to Java. Before you can read or write to a file, you must open it using Python's built-in **open()** function. The syntax is shown in figure 8.3.5.

Figure 8.3.5: Syntax for using Python open() Function

```
file object = open(file_name [, access_mode][, buffering])
```

Once a file is opened and you have one file object, you can find out if it is closed, what mode it is in and the name of it. The **close()** method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done. This confirms that Python is similar to Java and C++ when it comes to dealing with resource allocation when supporting file processing.

To read and write in python, the readable and simple syntax structure continues to be used. The **write()** method writes any string to an open file. The **read()** method reads a string from an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character to the end of the string. Figure 8.3.6 shows the **write()** and **read()** syntax.

Figure 8.3.6: Using Python write() and read() Methods

```
fileObject.write(string)
fileObject.read(string)
```

Further, Python has a **tell()** method that tells you the current position within the file and a **seek()** method that changes the current file position. Files may also be renamed and deleted using the **rename()** and **remove()** method. ([Python, 2014d]) These methods support Python simplicity, and have led to file processing being an exceptionally easy component of the language when compared to Java and C++.

In summary, file processing in Java and Python simply adopts the basic file processing features of the older language C++, and tweaks them to the nuances of the respective language.

9. Exception Handling

Exceptions may refer to unexpected situations, such as:

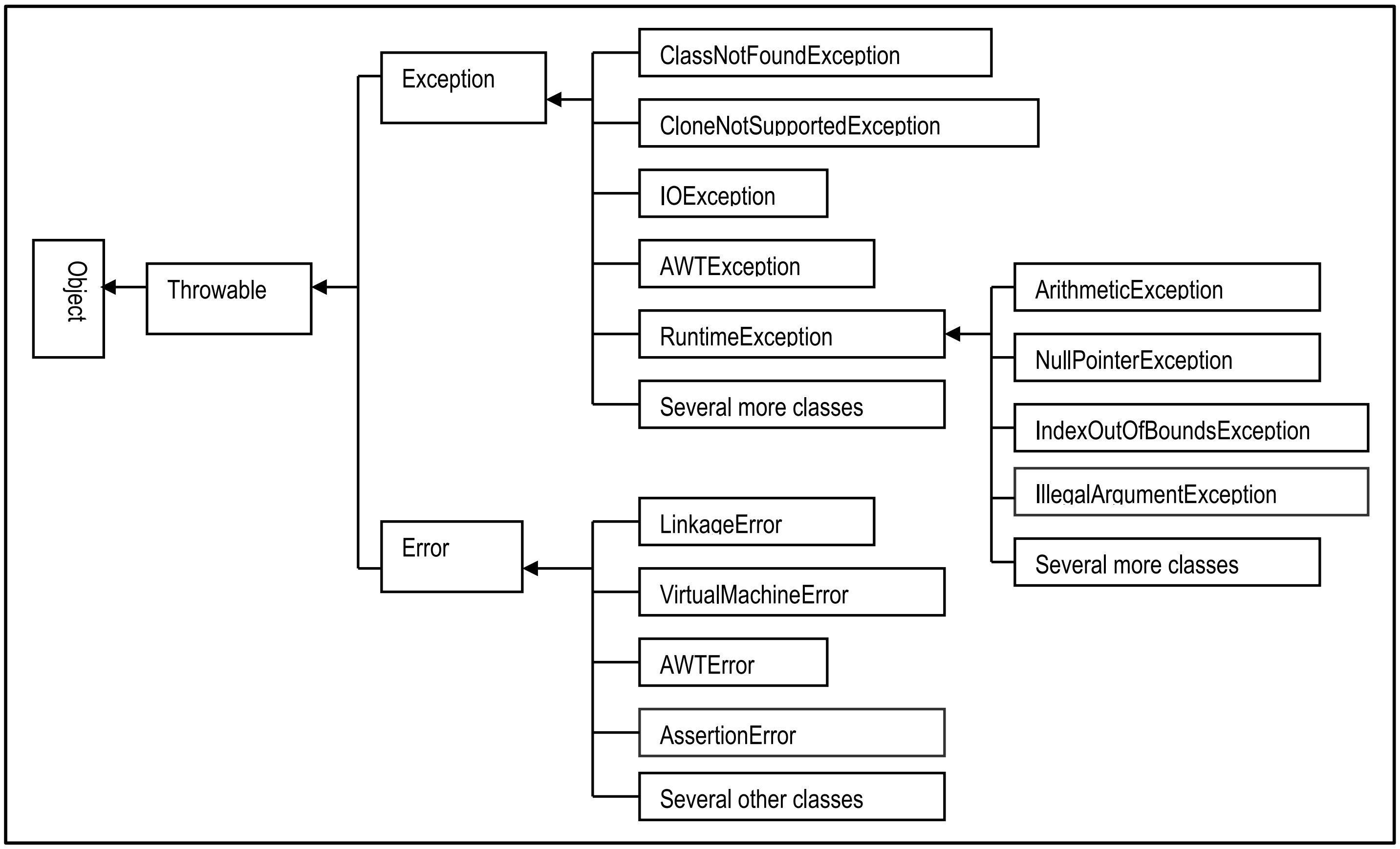
- The user enters a character where an integer is expected;
- The program uses an array subscript that is outside of the range of values;
- An attempt is made at dividing by zero;
- An attempt is made to write to a file that does not exist.

There are three broad categories of programming:

- Syntax errors
- Logic errors
- Runtime errors

In Java, exceptions are provided in a very elaborate way, consisting of a hierarchy of exception classes, as summarized in figure 9.1(adopted from [Foster, 2014k] and elaborated on in [Oracle, 2013]). Each Java method that throws an exception must have an adjustment in its heading to indicate what type of exception is thrown. The **try-catch** blocks are used in a manner that actually mirrors to C++, to recover from any exception that might have been raised. The anatomy of a Java method that throws an exception is shown in figure 9.2 while an illustration of a Java **try-catch** block is shown in figure 9.3.

Figure 9.1: The Java Exception Class Hierarchy



Adopted from [Foster, 2014k]

Figure 9.2: Revised Anatomy of a Java Method

```
Method ::=
<Modifier><ReturnType><MethodName>(<Parameters>) throws <Exception1>,…<ExceptionN>
{
  … // Body of Method
}

Modifier ::=
[final] public | private | protected [static] [abstract]
```

Note: Exception1… ExceptionN represent valid exception class(es) as indicated in figure 9.1

Adopted from [Foster, 2014i]

Figure 9.3: Simple Illustration of a Try-Catch Block

```
public void InputData () throws Exception
{
  String InputString;
  try
  {
    int InputInteger = Integer.parseInt(JOptionPane.showInputDialog (null, “Integer Please!”, “Integer Prompt”));
  }
  catch (Exception Ex)
  {
    System.out.println(Ex.getMessage( )); // Prints the message of the exception
    System.out.println(“Invalid integer entered”); // Prints additional message
  }
 // …
}
```

Adopted from [Foster, 2014i]

In C++, exceptions are not handled in a hierarchy manner as done in Java. Instead, all exceptions are treated similarly. You can throw an exception from any function and catch an exception, report it or re-throw it for another function. C++ also does not require any modification to its method definition as Java does, but may use the throw keyword deliberately, as depicted in figure 9.4. Note that C++ does utilize a **try-catch**, like Java; however, the whole exception handling mechanism is much simpler than in Java. For more discussion of C++ exception handling, please refer to [Foster, 2012c] and [Oracle, 2013].

Figure 9.4: Illustrating Exception Handling in C++

```
const int DIV_BY_ZERO = 10;  // A global constant
// …
double divide (double First, double Second); // Function prototype
// …

int main(int argc, char* argv[])
{
  // …
  try
 {
          double Result;
          double FirstNumber;
          // …
          Result = divide(FirstNumber, 0);
   // …
  }

  catch (int ErrNum)
 {
          if  (ErrNum == DIV_BY_ZERO) cout << “Cannot divide by zero!\n”;
          // …
  }

  // …
}  // End of main

double divide (double First, double Second)
{
          if (Second == 0) throw DIV_BY-ZERO;
          return First/Second;
}
// …
```

Adopted from [Foster, 2014i]

In Python, there are two important features to handle any unexpected error. Assertions, which will not be discussed here, are often placed by programmers at the start of a function to check for valid input, and after a function call to check for valid output. The second is standard exception handling. Similar to Java and C++, Python uses a two-part syntax. Instead of **try-catch** block, however, Python calls it a **try-except** block. Java and C++ also have an optional third-part, the **finally-clause**. In Python, this is an **else-clause**. A simple example of the syntax is shown in figure 9.5.

Figure 9.5: Python Try-Except-Else Block

```
try:
    Operations here;
    ...
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
else:
    If there is no exception then execute this block.
```

As is the case with Java and C++, a single try statement can have multiple except statements, which is useful when the try block contains statements that may throw different types of exceptions. You can also provide a generic except clause, which handles any exception. Again, this is supported by all three languages. Further, user-defined exceptions are allowed to customize messages to the user.

10. Summary and Concluding Remarks

This final section provides some concluding remarks in light of the foregoing discussions. These insights are organized in three categories:

- Language Summary
- Favorability Conclusion
- Future Expectations

10.1 Language Summary

After a thorough discussion of the fundamental and advanced features of three prominent programming languages, Java, C++ and Python, a developer should have a reasonable understanding of what differences to look for when considering any of these languages.

Starting from choosing a language that fits the target domain, usually based on the appropriate paradigm, to programming criteria such as readability and simplicity, a programmer has a direct way to compare languages. In our discussion, Python has proven to be a more readable and simpler language, by design. All three languages also fit multiple domains and paradigms, but have slight differences that may be more or less useful for the programmer.

On matters relating to the translation process, data types and variables, support for abstraction, and expressions, the three languages exhibit some similarities as well as some idiosyncratic differences. The syntax and semantics may play a factor in usability, while the different data types may suit different programming needs. All three languages supporting abstraction, prove their usefulness for complex problems-solving. The various expressions and assignment statements differ only slightly, but certain uses may require the extended options in C-based languages as compared to Python. C++ is particularly impressive in its handling of operator overloading and templates. Python and C++ are particularly impressive in their flexibility in facilitating programmer-defined abstractions.

The control structures and subprograms affect the way code is organized to solve programming problems. In C-based languages, control structures are similar, hence a strong similarity between Java and C++. In Python, certain control structures are handled differently, but still in a user-friendly and easily-understandable. As for subprograms, C++ and Java support integrated independent subprograms while Python supports integrated nested subprograms. Subtle differences have also been noted in the area of parameter-handling among the three languages.

Support for inheritance, polymorphism and file processing are found in all prominent languages, including the three discussed here. The use of inheritance and polymorphism go hand-in-hand and adds flexibility and simplicity to code implementation. Both C++ and Python support multiple-inheritance while Java supports single-inheritance. In the area of file processing, Python and C++ exhibit a distinct advantage in the simplicity of file processing, compared to Java.

Finally, our discussion of exception handling shows that Java employs a hierarchical approach to exception while C++ and Python both employ a more flattened general approach. All three languages use some form of a **try-catch-finally** block, although Python uses the naming convention of a **try-except-else** block. Fortunately, the underlying implementations of all three languages are similar and allow a programmer to handle unexpected situations efficiently in all three languages.

10.2 Favorability Conclusion

The Java language is a highly successful language that was largely influenced by C and C++. This has led the language to strip down some of what is considered unnecessary in C++ and focus on the important constructs. Many consider Java easier to learn, compared to C and C++, which has led to its increased popularity. Once a programmer arrives at the understanding that Java is free, provides platform independence, and already exists in most domains, it is easy to consider the language in a favorable way.

By allowing abstraction, encapsulation, polymorphism and inheritance, programmers may utilize a language such as Java to its full ability. Java also has a rich API which extends its functionality for I/O, networking, utilities, xml parsing, database connection and much more. Further, the open source libraries, contributed by Apache, Google, and other organizations, make Java development easy, fast, and cost-effective. Another significant separation of Java from C++ is the use of Java to create Web applets, which was actually its original purpose in the industry.

Compared to Java, C++ seems to lack slightly in the area of ease-of-learning, although veteran programmers will live and die by the C++ language. Relative to Java, the language tends to provide more flexibility and efficiency to experienced programmers. Much of the favorability of Java or C++ comes down to immediate need. If the organization that a software developer is working for uses Java or C++, it is reasonable to assume that either language will be highly favorable. Moreover, the business and scientific domains are both filled with many technologies that utilize Java and C++.

One of the main differences between the two languages, as discussed earlier, involves the use of C++ for major projects, such as systems programming. Developing an operating system or working with a large-scale project may be better suited to the more complex nature of C++. C++ also allows a programmer to learn basic concepts such as memory management, pointers and experience coding without rich built-in libraries. This leads to a favorable foundational understanding of other programming languages.

The development environments for both Java and C++, namely Eclipse and NetBeans, play a huge role in making both languages highly favorable. IDEs like these not only help in code completion, but also provide powerful debugging capabilities, which is essential for real world development.

The Python language, as discussed throughout this discourse, excels when it comes to the readability and simplicity of code, but still holds its own when implementing other features. It is estimated that a typical Python program will require significantly fewer lines of code than a corresponding program in an earlier language such as Java or C++.

The various implementations of Python also lead to the favorability of the language. By including the ability to use a Web framework, such as Django, or an enhanced IDE, such as PyCharm, the flexibility of the language and what a programmer can do with it is endless.

Essentially, Python covers the range of programmers from beginners who rely on easy-to-learn programming languages with a gradual learning curve, to advanced programmers who need complex frameworks and complex functionality similar to a C-based language. The rapid ability of Python code interpretation and nature as a scripting language may allow it to contribute to domains that C-based languages are otherwise ill-prepared for. A comparison among similar languages, such as Ruby, would be needed to discuss where Python fits in the general realm of scripting languages. Overall, the language appears to be quite enticing for programmers.

Ultimately, when it comes to favorability, it depends on the context of the problem and the needs of the programmer. All three languages have high favorability due to the fact that they may be used in many environments, are easy to learn, free, and implement most, if not all the constructs any programmer may need to solve typical programming problems.

10.3 Future Expectations

The expectations of the three languages discussed, Java, C++ and Python, require a contrasting approach.

The first viewpoint is that of the past, and what already exists in the market today. Replacing existing software systems is expensive and time consuming exercise [Foster, 2014l]. Training new developers to learn new languages to implement new software is also a risky strategy. In light of these realities, the dominance of C-based languages, namely C++ and Java, is likely to continue to hold a large portion of market share.

The second viewpoint is that of the future, with a massive movement toward Web technologies instead of standalone programs. One of the big booms of scripting languages such as Python is the ability to incorporate object-oriented principles, much to the same functionality as C-based languages, while placing increased emphasis on the Web, fast development, and code readability. It is therefore reasonable to expect an increased usage of such languages (Python included) in the foreseeable future.

For Java, many in the enterprise computing world will continue to see this language as relevant moving forward. Oracle has a strong hold on the market in the area of databases and business applications; the company's acquisition of Sun Microsystems and Java almost guarantees a bright future for the language. However, Oracle must still press on with the evolution of the Java language, if it is to remain relevant in the future. One significant benefit is the ease at which the language can be learned.

C++ will continue to be important in terms of systems programming and for highly optimized native code requirements such as gaming. However, these are specialized niches. If the language is to regain some of its dominance of the past, it will be necessary for more powerful CASE and RAD tools that support the language to emerge.

For Java and C++, considering the Android platform must also be considered when it comes to the popularity of the language. The Android framework is built using C/C++ libraries and applications are largely coded in Java. Currently, Android has over fifty-percent of the mobile OS market, and does not show signs of falling behind in a maturing market. With the move toward Web development, the use of these languages will not disappear, and may largely continue to improve the popularity of languages.

For Python, its real strengths have been server side technology, software development by non-programmers, and as an embedded scripting engine for trusted plugins. Apple has expressed their support for Python by building tools that rely on it and Microsoft ships their Python tools for Visual Studio bundle. Google has also chosen Python as the only dynamic language supported on their App Engine platform. By having the support of major players in the technology industry, Python looks to be running away from its competitors, such as Ruby, and is therefore expected to prosper moving forward.

In conclusion, Java is likely to appeal to the general market, C++ is likely to appeal to advanced programmers and anyone working on software projects with a certain level of complexity or size, and Python will likely appeal to the general market more interested in web development and a dynamic language that focuses on readability.

References

[C++, 2013] cppreference. C++ Keywords. 2013. http://en.cppreference.com/w/cpp/keyword (accessed April 22, 2014)

[C++, 2014] Microsoft Developer Network. Indirection Operator: *. 2014. http://msdn.microsoft.com/en-us/library/fw63e3c3.aspx (accessed April 22, 2014)

[C++, 2014a] stackoverflow. How portable is C++?. 2011. http://stackoverflow.com/questions/7862336/how-portable-is-c (accessed April 18, 2014)

[C++, 2014b] Microsoft Developer Network. Data Type Ranges. 2014. http://msdn.microsoft.com/en-us/library/s3f49ktz(v=vs.80).aspx (accessed May 1, 2014)

[C++, 2014c] tutorialspoint. C++ Inheritance. 2014. http://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm (accessed May 2, 2014)

[C++, 2014d] cplusplus. Polymorphism. 2014. http://www.cplusplus.com/doc/tutorial/polymorphism/ (accessed May 2, 2014)

[C++, 2014e] C++ File Streaming. File Processing in C++. 2014. http://www.functionx.com/cpp/articles/filestreaming.htm (accessed May 2, 2014)

[Foster, 2012a] Foster, Elvis C. *Lecture Notes in C++ Programming*. Keene NH: Keene State College, 2012. See chapter 6.

[Foster, 2012b] Foster, Elvis C. *Lecture Notes in C++ Programming*. Keene NH: Keene State College, 2012. See chapter 9.

[Foster, 2012c] Foster, Elvis C. *Lecture Notes in C++ Programming*. Keene NH: Keene State College, 2012. See chapter 11.

[Foster, 2014a] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 1.

[Foster, 2014b] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 3.

[Foster, 2014c] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 4.

[Foster, 2014d] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 5.

[Foster, 2014e] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 6.

[Foster, 2014f] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 7.

[Foster, 2014g] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 8.

[Foster, 2014h] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 9.

[Foster, 2014i] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 11.

[Foster, 2014j] Foster, Elvis C. *Lecture Notes on Programming Languages*. Keene NH: Keene State College, 2014. See lecture 12.

[Foster, 2014k] Foster, Elvis C. *Lecture Notes in Data Structures & Algorithms*. Keene NH: Keene State College, 2014. See chapter 2.

[Foster, 2014l] Foster, Elvis C. *Software Engineering: A Methodical Approach*. New York: Apress Publishing, 2014. See chapter 18.

[IBM, 2014] IBM. C/C++ Supported Operating Systems. http://publib.boulder.ibm.com/infocenter/db2e/v9r1/index.jsp?topic=%2Fcom.ibm.db2e.doc%2Fdbeapr0202.html (accessed April 26, 2014).

[Java, 2014] Oracle. Java Language Keywords. 2014. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html (accessed April 22, 2014)

[Java, 2014a] JavaWorld. Java's three types of portability. 1997. http://www.javaworld.com/article/2076944/java-s-three-types-of-portability.html (accessed April 18, 2014)

[Java, 2014b] eclipse. FAQ How do I use the platform debug tracing facility?. 2014. https://wiki.eclipse.org/FAQ_How_do_I_use_the_platform_debug_tracing_facility%3F (accessed April 29, 2014)

[Java, 2014c] NetBeans. Designing a Swing GUI in NetBeans IDE. 2014. https://netbeans.org/kb/docs/java/quickstart-gui.html (accessed April 29, 2014)

[Java, 2014d] Bornat, Richard. *Understanding and Writing Compilers: A do-it-yourself guide*. London: Middlesex University, 2014.

[Java, 2014e] The Java Tutorials. Primitive Data Types. 2014. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (accessed May 1, 2014)

[Java, 2014f] The Java Tutorials. Expressions, Statements, and Blocks. 2014. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html (accessed May 1, 2014)

[Java, 2014g] The Java Tutorials. What is Inheritance?. 2014. http://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html (accessed May 2, 2014).

[Python, 2014] python for beginners. Keywords in Python. 2014. http://www.pythonforbeginners.com/basics/keywords-in-python (accessed April 22, 2014)

[Python, 2014a] Portable Python. What is Portable Python?. 2014. http://portablepython.com/ (accessed April 18, 2014).

[Python, 2014b] tutorialspoint. Python Variable Types. 2014. http://www.tutorialspoint.com/python/python_variable_types.htm (accessed May 1, 2014).

[Python, 2014c] stackoverflow. Practical example of Polymorphism. 2014. http://www.cplusplus.com/doc/tutorial/polymorphism/ (accessed May 2, 2014).

[Python, 2014d] tutorialspoint. Python Files I/O. 2014. http://www.tutorialspoint.com/python/python_files_io.htm (accessed May 2, 2014).

[Python, 2014e] Python Software Foundation. Assignment Statements. 2014. https://docs.python.org/2/reference/simple_stmts.html (accessed December 23, 2014).

[Oracle, 1999] PL/SQL User's Guide and Reference. Control Structures. 1999. http://www.csee.umbc.edu/portal/help/oracle8/server.815/a67842/03_struc.htm (accessed May 2, 2014)

[Oracle 2013] Oracle Corporation. 2013. Java API Specification. 2013. http://www.oracle.com/technetwork/java/api-141528.html (accessed December 19, 2013).

[TIOBE, 2014] TIOBE Software. TIOBE Index for April 2014. 2014. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (accessed May 1, 2014)

[Wikibooks, 2014] Wikibooks. Python Programming / Data Types. 2014. http://en.wikibooks.org/wiki/Python_Programming/Data_Types (accessed December 23, 2014].