CI646 - Programing Languages, Concurrency and Client/Server Computing

Programming Language comparison report

Joseph Batchelor

Table of Contents

Introduction

This report is designed to provide an in-depth technical comparison of two programming paradigms. Java, an object-oriented language and Haskell, a functional language. Below discusses some similarities, differences, and their approach to solving problems.

Type systems

Haskell is a strong statically typed language, meaning classes and types are declared and determined at compile time. Haskell's type system utilises strong type inference for automatic detection of types for every expression and value, which effectively uncovers bugs or errors at compile time. Furthermore, Haskell has a strongly typed system, which means that variables have a defined type and have strict rules about combining different types in expressions. Haskell has a type-safe system which means its type checking prohibits the combinations of different types. To prevent errors and to provide effective memory safety, this excludes the use of polymorphic techniques such as subtyping and casting. Types do not use the same amount of memory as each other meaning Haskell accesses only the memory location it is authorized to access based upon the type used. Additionally, Haskell's type safe system means that data types will always be unchanged and return to the same type.

Java is also a statically typed language which also supports runtime operations. This means Java has dynamic type checking which at runtime is able to reconfigure types and adjust memory allocation to fit the new type. This allows for casting and subtyping of a type while maintaining memory safety. Java is also strongly typed but is also considered to be weaker to Haskell, due to its ability to allow coercion of types. By default, Java supports automatic widen casting however, narrow casting can also be achieved but requires manual insertion of code to inform the complier, this is to prevent memory problems as the allocation of memory would adjust. Java has an unsafe type of system due to the fact it allows actions to affect the return of types from their original state such as casting and subtyping.

Functional purity

Pure functions are functions where the return value remains the same as the arguments due to the absence of side effects. Haskell is a purely functional language, so implementation of pure functions is straightforward. Due to the fact that by default, Haskell is immutable which means that expressions cannot change after they have been evaluated. This benefits concurrent systems as immutability is useful because an immutable state is essentially read only and cannot not be modified. This prevents conflicts and allows threads to perform safer when handling data and preventing race condition from occurring. However, a disadvantage of having an immutable system is that functions have no side effects, making it difficult to perform I/O operations (Milewski, 2022).

In contrast, Java is mutable by default meaning it is able to reassign variables and objects. Nevertheless, mutation is useful for code reusability and offers flexibility, preventing systems from being overwhelmed with unnecessary objects as it allows adaptation of existing code.

In Java functional purity can be more difficult to implement due to states being susceptible to side effects. Creating a pure function in Java, will require the avoidance of a mutable state. This means methods within a program will have to abide by the following rules:

- Methods can only use mutable objects created within their own body, introducing encapsulation.
- Method calling is only possible if they are pure functions being called.

However, immutability can be achieved by explicitly declaring variables or classes as private or, with a final modifier, which can be seen in appendix A.

Code reuse

One approach to achieving code reusability is using polymorphism, this is a concept that allows objects to decide at compile or runtime the form of the function to implement.

Both paradigms utilise Parametric polymorphism, a technique that allows a language to be more expressive while preserving static type safety. This process uses generic functions and data types to handle values regardless of their type. A basic generic function example in Haskell would be FUNCT:: T → T where T can be of any type required. Java has the capability of using genetic types as parameters to classes, methods, and interfaces. A generic Java class example can be seen in appendix B.

Another similarity between both paradigms is the use of Ad-hoc polymorphism, the process of overloading functions and operators. It is the concept of writing the same function with different parameter types to behave differently. When applied to Java this allows for the collection of methods under the same name, introducing code reusability, as demonstrated in appendix B. Overloading in Haskell requires the use of type classes which are a collection of types (value → type → Type class), which shares a similar resemblance to Java's type system (Object → Classes → Interface). As opposed to Java, Haskell supports operator overloading, a technique that facilitates in the alteration of user defined types. Allowing us to change their behaviour as if they were primitive data types. Appendix B illustrates an example of Ad-hoc polymorphism within Haskell.

Unlike Java, Haskell does not support both subtyping and Ad-hoc polymorphism (coercion). Subtyping is not supported on Haskell due to the strong type of inference. Whereas Java, utilises this technique to allow objects of some type to be considered and used as a subset of that existing object. This means an object is able to provide several functions with different types without additional code.

Coercion polymorphism allows Java to reconfigure types at runtime with its permissive type matching rules. This allows for automatic type conversion known as narrowing and widowing casting, which is the conversation of one data type to another. Appendix B provides an example of Casting. Haskell's type system prohibits casting as if the value was to change from one type to another the allocation of memory would change. When the compiler reads the value at compile time it creates a memory allocation necessary for the first instantiation. However, if it tries to store the new casted version it will not have enough memory allocation to do so and therefore majority of the memory will be overwritten. This affects the pointers within the that area and in most cases causes undefined behaviour or displays a memory exception error.

Modularity

A unique evaluation strategy that doesn’t reside in Java but does in Haskell is lazy evaluation, which is a process to evaluate a program. This means that expressions are not evaluated when bound to variables, but rather their evaluations are delayed until their results are needed by other operations (HaskellWiki, 2022). Lazy evaluation achieves modularity by decoupling functions, which allows each component to perform its tasks independently. Lazy evaluation allows methods to work close to together while still being separated for example: decoupling a function for generating data from a function that decides how much data to generate.

Unlike Haskell, Java achieves modularity by using inheritance, the process of sharing attributes and methods between classes. Java utilizes classes to wrap states and behaviours together and then relies on inheritance to extend them. This has the advantage of making slight modifications for specific purposes more simplistic. The disadvantage is that the total sum of what's going to happen on any given method call can become difficult to comprehend, especially when debugging. The system becomes more complex and messier as layers of inheritance are added on. Appendix C demonstrates the use of inheritance in Java.

A similarity between both paradigms is the use of pure functions to achieve modularity. A pure function is a function that has no side effects, it will always return the same result as the argument values. Pure functions provide modularity by not relying on external variables or states to function, allow them to work independently and allows for encapsulation. As previously stated, Haskell is naturally immutable which allows for the effortless implementation of pure functions. Whereas Java is mutable by default and requires the use of modifiers the make states immutable.

Another parallel between both paradigms is the use of encapsulation, this allows us to control access to different values or states separating the responsibility of what each module can do. The process at which encapsulation is achieved by both paradigm are different.

Java on the other hand utilises encapsulation by declaring variables of a class as private, then provides a public setter and getter method the ability to modify and view the variables. The idea is to wrap variables and methods together into a single unit (Edureka, 2022). Haskell can achieve encapsulation through the use of pure functions; however, these expressions are immutable meaning they are unchangeable. Which gives Java the advantage to modify values by using the setter method. Appendix C provides an example of encapsulation in Java.

A Final resemblance to achieving modularity is the use higher order functions which are functions that take function arguments or returns a function. These functions can be used to abstract behaviour into different reusable segments that can be called by different functions or classes.

Input/output

I/O operations allow the retrieval and or transfer of data to or from an application. These operations require mutation of states in order to affectively work, which produces sides affects. This can be a problem for a pure paradigm like Haskell. Due to its immutable design the approach of I/O in Haskell is achieved differently compared to Java; IO actions in Haskell cannot interact with pure functions. We use type IO to differentiate between referentially transparent code and code that is not (IO actions). When executed type IO is used as a wrapper for data related functions and helps connects these functions to other parts of the system (HaskellWiki, 2022). Appendix D provides an example I/O within Haskell.

Due to both lazy evaluation and purity this makes it difficult for I/O implementation. When I/O evaluations occurs, it can cause arbitrary side effects, thus affecting the order and deferring I/O operations to be performed at different times making it unpredictable.

I/O can be easily achieved in Java, by using a stream concept to perform quick I/O operations. These operations require side effects in order to affectively retrieve or transmits data. Due to Java’s mutable design, I/O operations can be freely performed without the worry of side effect causing conflicts with other elements of the system. This provide Java the advantage of easily implementing I/O operations within an application Appendix D provides an example I/O within Java.

Concurrency

Both paradigms are able to perform multiple tasks simultaneously, however, differ from each other. Introducing threads in Java can be done by implementing the runnable interface, this uses a method called run() which specifics a specific action for a thread to run. Haskell on the other hand, creates threads using forkIO alongside type IO to create new threads. Using the function forkIO :: IO () -> IO ThreadID can allow IO actions to be ran in the background.

Thread safety can be accomplished differently within both paradigms. Java utilizes two thread safety techniques called synchronization and volatile labelling. Synchronization is a common approach that is widely used for thread safety in java. Synchronized code informs the compiler to execute a specific segment of annotated code with one thread at a time. Essentiality any segment of code that has been annotated with synchronization is internally locked to one thread. The premise works by locking and unlocking code before a thread enters into synchronized code, it has to acquire the lock on the object and when code execution ends, it unlocks that segment of code for other threads.

Another technique that java uses is the volatile keyword, this tells threads to read varibles from memory, and not from thread cache. Using the volatile annotation on a variable informs other threads to use more recent values of variables written by previous threads before they write to the volatile variable (JournalDev, 2022). Appendix E provides insight to creating threads in Java.

Haskell's approach to safe multithreading is the use of the Mvars type, which are used for communication amongst threads. MVar acts as a mutable location in memory for data to be stored in, it is used when multiple threads need to read and write access to states. As data being shared is immutable there is no worry for race conditions.

Mvar can be used to wrap shared data between multiple threads similar to java, it uses the approach of locking (we take the MVar) and unlocking (we put the MVar) segments of memory. This means that any operation performed on a state such as modifying it will consist of locking it, changing it, and then unlocking it. Once the operation is finished this allows other threads to access it.

This prevents data inconsistency as it stops threads from intercepting or interrupting another thread from modifying the state. The idea is based around sequential access control. This differs from synchronization as Mvar provides the combination of a lock and mutable variables in Haskell and primarily focuses on locking memory whereas synchronization focuses on locking segments of code/resources (0reilly, 2022).

Client-server systems

A client server system that I think Java would be suitable for is a webserver for frontend development. When developing web applications with the use of Java servlets, this requires a lot of data mutation to satisfy client requests. Unlike Haskell, which is immutable, Java handles mutation very well especially coinciding with threads. Java's advantage is that it handles memory more efficiently with larger applications as it makes space available for new objects by providing dynamic memory allocation (Baeldung, 2022). Modifiability and reusability are two easy techniques that Java can introduce with the concept of OO.

Java's approach to I/O operations can be easily implemented without the complication that side effects would have on Haskell. Additionally, data inconsistency can be easily resolved within Java providing a better advantage.

Haskell on the other hand, excels at complex computations which is why I feel it would be suitable for backend business logic. Its runtime performance allows for high concurrent applications to be reliable and safe due to its immutability and purity, ensuring that data is not subject to race condition.

Complete referential transparency is achieved with pure functions, which provides better security as well as the reduced occurrence of errors. Finally, Lazy evaluation makes it is easier to write large business logic systems because it doesn't load every logic operation at once, instead it generates them as it goes along, which eliminates long term space leaks.

Conclusion

Researching both paradigms varied greatly as certain key areas of the report where easy to find information such as concurrency between the two. Areas like I/O integration were more difficult as information for Haskell mainly focused on implementation with the use of extensions and so information was opinionated and consequently biased. Finding the native approach that Haskell introduces was scarce. Uncovering suitable client server systems for both paradigms was difficult as both are capable of running similar servers. It mainly mattered about the specific operation they performed that varied their suitability.

Bibliography

Wiki.haskell.org. 2022. *Why Haskell matters - HaskellWiki*. [online] Available at: <https://wiki.haskell.org/Why_Haskell_matters#:~:text=Data%20encapsulation%20is%20done%20in,from%20outside%20of%20the%20module> [Accessed 17 March 2022].

Eureka. 2022. *Encapsulation in Java | How to master OOPs with Encapsulation? | Edureka*. [online] Available at: <https://www.edureka.co/blog/encapsulation-in-java/#:~:text=Encapsulation%20in%20Java%20can%20be,and%20view%20the%20variables%20values.> [Accessed 30 March 2022].

Milewski, B., 2022. *Pure Functions, Laziness, I/O, and Monads*. [online] schoolofhaskell. Available at: <https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io#:~:text=Here%20are%20the%20fundamental%20properties,it%20access%20any%20external%20state> [Accessed 25 February 2022].

Le, V., 2022. *Pure Functions*. [online] Yext Engineering Blog. Available at: <https://engblog.yext.com/post/pure-functions> [Accessed 7 April 2022].

Wiki.haskell.org. 2022. *Lazy evaluation - HaskellWiki*. [online] Available at: <https://wiki.haskell.org/Lazy_evaluation#:~:text=From%20HaskellWiki,are%20needed%20by%20other%20computations> [Accessed 2 April 2022].

www.javatpoint.com. 2022. *Java IO - javatpoint*. [online] Available at: <https://www.javatpoint.com/java-io> [Accessed 14 April 2022].

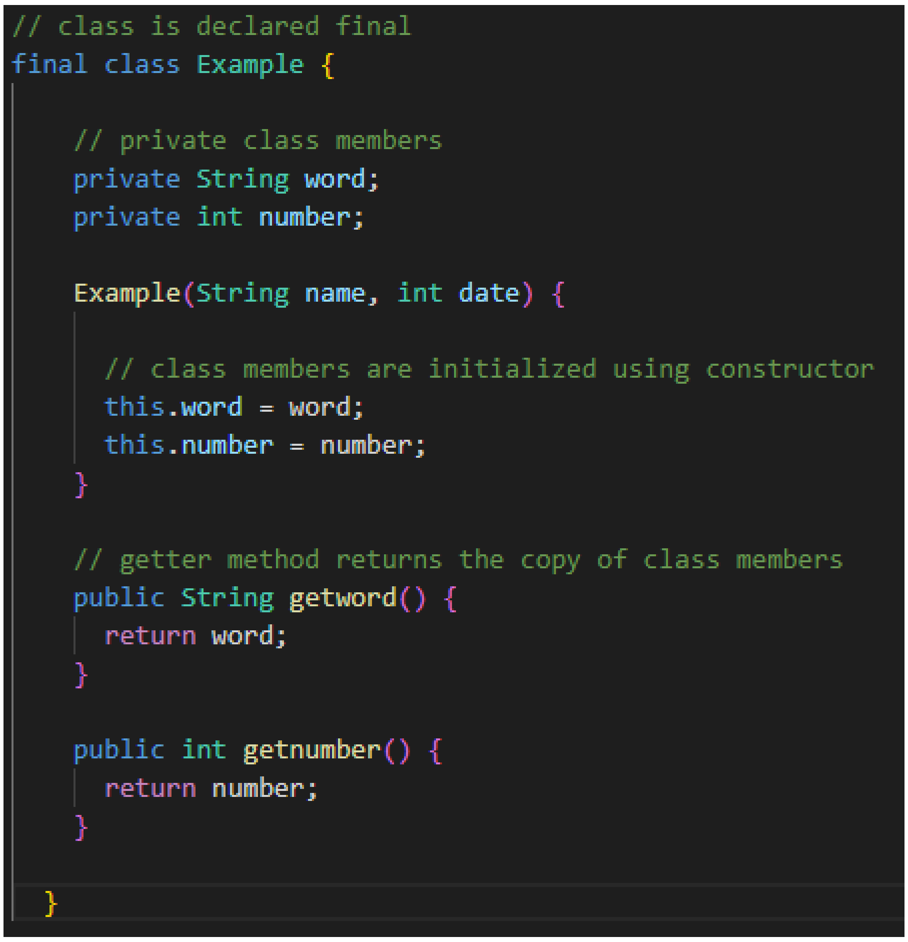
JournalDev. 2022. *Thread Safety in Java - JournalDev*. [online] Available at: <https://www.journaldev.com/1061/thread-safety-in-java> [Accessed 1 April 2022].

O'Reilly Online Learning. 2022. *Parallel and Concurrent Programming in Haskell*. [online] Available at: <https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/ch07.html> [Accessed 8 April 2022].

Appendices

Appendix A

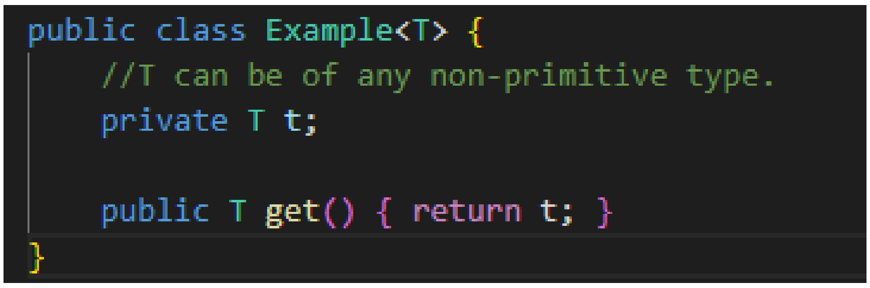
Immutable class in Java:



```java
// class is declared final
final class Example {

    // private class members
    private String word;
    private int number;

    Example(String name, int date) {

      // class members are initialized using constructor
      this.word = word;
      this.number = number;
    }

    // getter method returns the copy of class members
    public String getword() {
      return word;
    }

    public int getnumber() {
      return number;
    }

  }
```

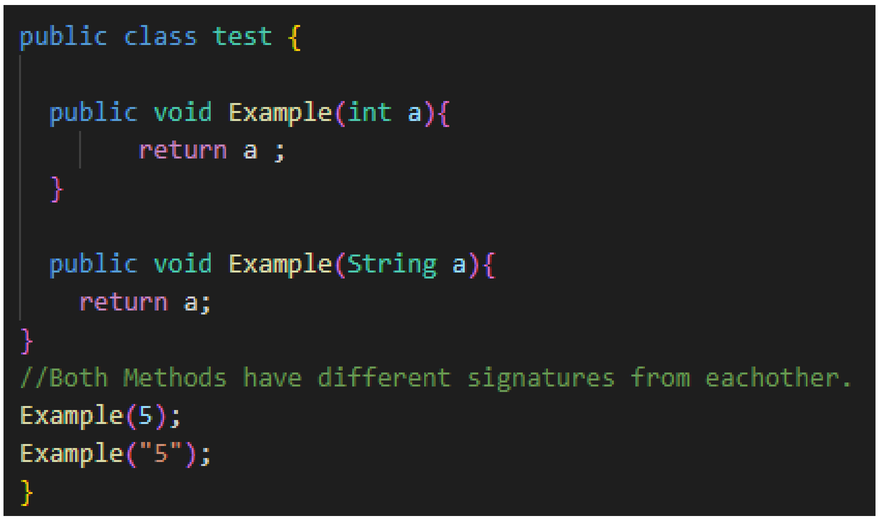
Appendix B

Example of a Java Generic class:



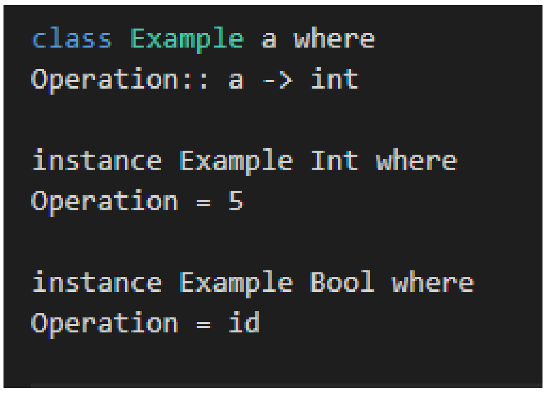
```java
public class Example<T> {
    //T can be of any non-primitive type.
    private T t;

    public T get() { return t; }
}
```

Example of method overloading in Java:



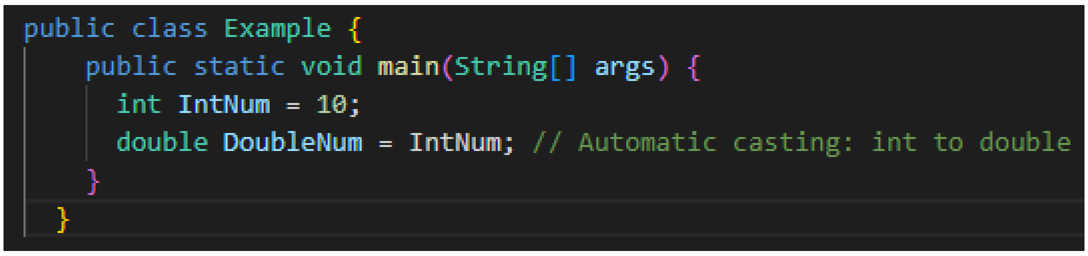
```java
public class test {

  public void Example(int a){
        return a ;
  }

  public void Example(String a){
    return a;
}
//Both Methods have different signatures from eachother.
Example(5);
Example("5");
}
```

Example of Ad-hoc polymorphism in Haskell:



```haskell
class Example a where
Operation:: a -> int

instance Example Int where
Operation = 5

instance Example Bool where
Operation = id
```

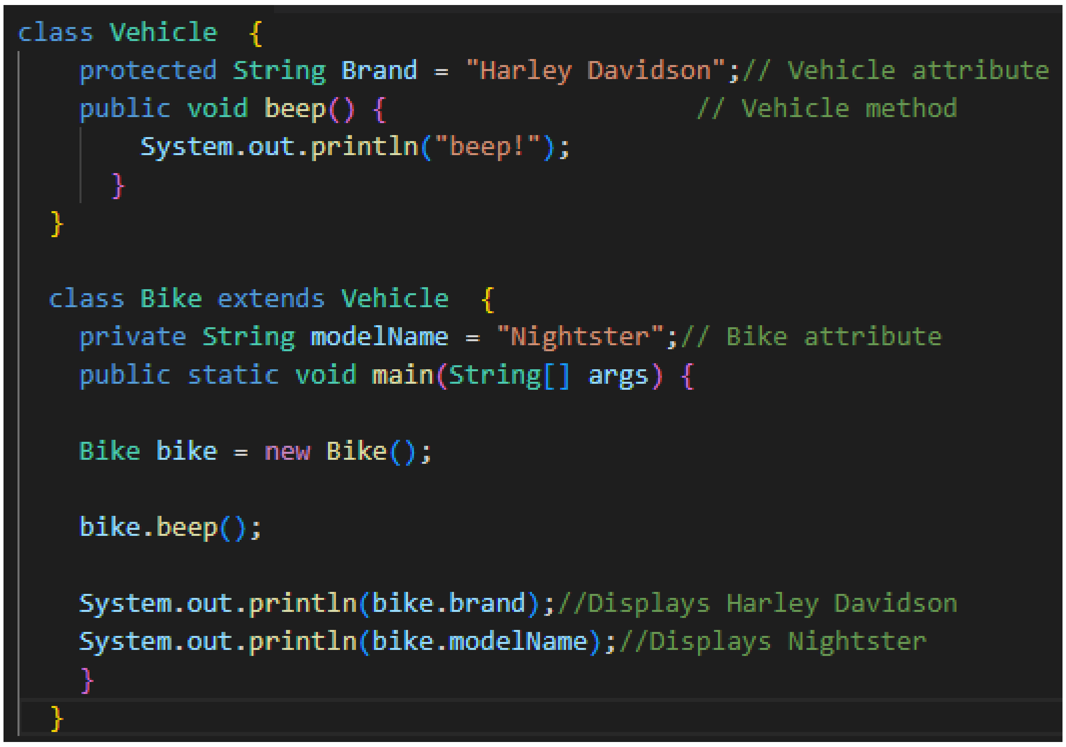
Example of casting types in Java:



```java
public class Example {
    public static void main(String[] args) {
      int IntNum = 10;
      double DoubleNum = IntNum; // Automatic casting: int to double
    }
  }
```

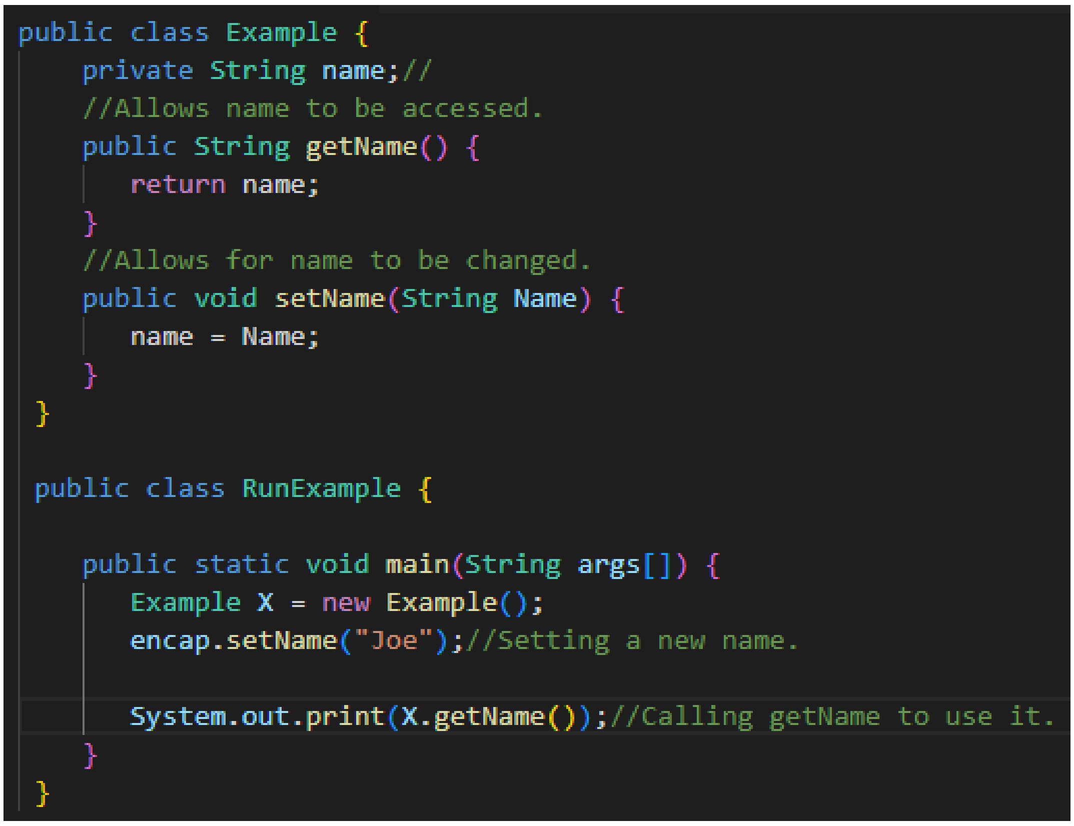
Appendix C

Example of inheritance in Java:



```
class Vehicle  {
    protected String Brand = "Harley Davidson";// Vehicle attribute
    public void beep() {                       // Vehicle method
        System.out.println("beep!");
      }
  }

  class Bike extends Vehicle  {
    private String modelName = "Nightster";// Bike attribute
    public static void main(String[] args) {

    Bike bike = new Bike();

    bike.beep();

    System.out.println(bike.brand);//Displays Harley Davidson
    System.out.println(bike.modelName);//Displays Nightster
    }
  }
```

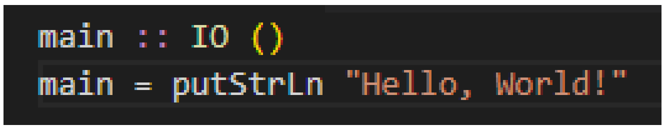
Example of encapsulation in Java:



```
public class Example {
    private String name;//
    //Allows name to be accessed.
    public String getName() {
       return name;
    }
    //Allows for name to be changed.
    public void setName(String Name) {
       name = Name;
    }
 }

 public class RunExample {

    public static void main(String args[]) {
       Example X = new Example();
       encap.setName("Joe");//Setting a new name.

       System.out.print(X.getName());//Calling getName to use it.
    }
 }
```

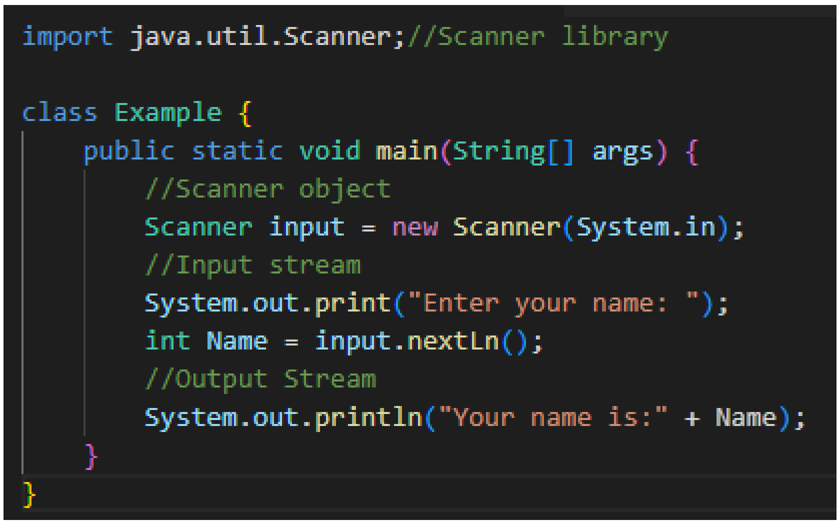
Appendix D

Example of I/O implementation within Haskell:



```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

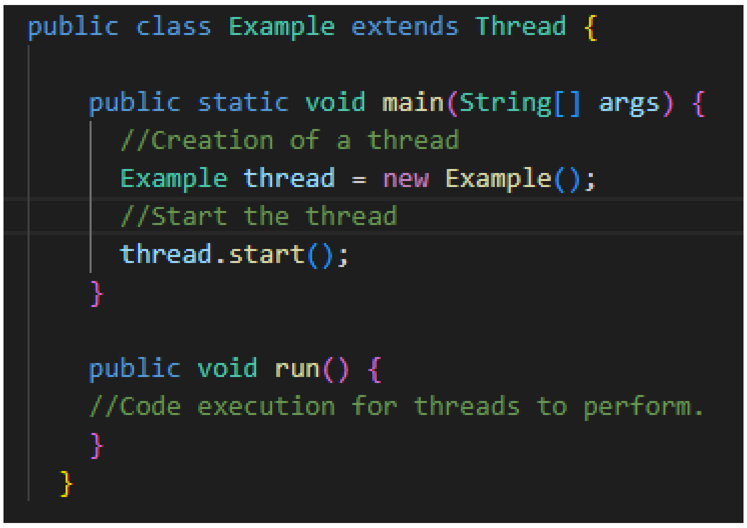
Example of I/O implementation within Java:



```java
import java.util.Scanner;//Scanner library

class Example {
    public static void main(String[] args) {
        //Scanner object
        Scanner input = new Scanner(System.in);
        //Input stream
        System.out.print("Enter your name: ");
        int Name = input.nextLn();
        //Output Stream
        System.out.println("Your name is:" + Name);
    }
}
```

Appendix E

Example of creating threads in Java:



```
public class Example extends Thread {

    public static void main(String[] args) {
      //Creation of a thread
      Example thread = new Example();
      //Start the thread
      thread.start();
    }

    public void run() {
    //Code execution for threads to perform.
    }
  }
```