# A comparison of Java and Haskell

*by* Jak Hall

**Submission date:** 11-Jan-2018 09:23PM (UTC+0000)
**Submission ID:** 81499807
**File name:** A_Comparison_of_Java_and_Haskell.pdf (351.18K)
**Word count:** 2068
**Character count:** 10964

# A Comparison of Java and Haskell

## Introduction

Programming languages can often be used to solve similar issues, but can use drastically different methods in completing operations. The features and methods a language uses defines its paradigm.

Java is a multi-paradigm language, it is often defined as object-oriented, however, this is effectively an abstraction of imperative and procedural paradigms which means it allows side effects and will read/run procedures in a defined sequence. Alternatively, Haskell is a purely functional language – "Functional programs contain no assignment statements, so variables, once given a value, never change" [1] they do not allow side effects and performs an evaluation of expressions in order to run.

## Modularity

A program with high modularity is traditionally one which is loosely coupled and has a high level of cohesion. A program with little dependencies between modules has loose coupling. Additionally, a module which methods/functions distinctively solve different tasks has high cohesion.

To reduce tight coupling, different modules must be independent of each other. In Java, an interface can be used to ensure that if class A relies on class B, class A must implement the methods of class B seen in figure 1. Likewise, Haskell programs are a collection of modules that can be defined by typeclasses which function similarly to javas interface – as shown in figure 2.

Haskell has the additional benefit of allowing functions as input for other functions this is called high order functions, this reduces duplicity of code and ensures each function is as independent as possible. Haskell implements lazy evaluation which means it waits until the last possible moment in compilation to evaluate an expression. This means compound functions can be synchronously executed with no temporary memory storing and also separates termination conditions from loop bodies.

Data encapsulation is used to separate interface from implementation, it also reduces the amount each module and function knows about the rest of the program and therefore promotes modularity. Encapsulation is an advantage of java and other OO languages as objects can be created that stores private data related only to the functionality of that class. 'getter' and 'setter' methods can then be implemented to controllably read/write the objects data an example of this can be seen in figure 3.

However, Haskell is also capable of encapsulation – as a program is composite of modules, each data type used can be capsulated in a separate module. This means, externally, all that is visible of the modules data is its typeclasses. Additionally, applicative functors, which are functors that can be used to sequence a series of expressions can use polymorphism to create higher levels of abstraction. It's also worth noting that in respect to separation of concern, both languages enforce unique module/class naming as well as a modules functions.

## Code Reuse

Subclasses which inherit from a superclass enable a hierarchical modularity - this in itself reduces code redundancy as methods attributed to both classes are only required in the superclass. Although a data type may inherit another, certain functionality may vary; in this case polymorphism is used to determine what is done with each type. Each language uses different forms of polymorphism to take advantage of modular inheritance.

Polymorphism found in both languages is 'parametric' polymorphism and 'ad-hoc' polymorphism. 'Parametric' polymorphism is often referred to as generics in OO design such as java, In figure 4 you can see how objects can be used to populate an array list despite being of different types since they inherit from the superclass 'Reptile'. Likewise, In figure 5, a Haskell polymorphic function 'f' accepts any concrete datatype in place of the unconstraint type variable 'a', this operates at compile time.

'Ad-hoc' polymorphism works by changing the functionality of an inherited method to adhere to the input type parameters of the subclass using 'overloading'. In Java, this can be implemented by simply including the superclasses method as seen in figure 6 with different arguments to the original method. Similarly, in Haskell, type classes can be used to specify the effect of adding different type inputs into a function, in the example "elem :: (Eq a) => a -> [a] -> Bool" the constraint variable type a shows ad-hoc polymorphism. Haskell can go further and change the functionality of its operators based on the used data types. This is also done at compile time.

Java has the added benefit of allowing subtype or inclusion polymorphism. Since the language is so flexible it allows for different objects to be accepted as the arguments of a method requesting their superclass. This is demonstrated in figure 7; all methods which are similarly inherited from the superclass reptile can be called collectively. Java is also exclusive in allowing overriding, since it can perform late binding – this means a subclass can change a superclasses method functionality entirely, independent on the number or type of parameters as shown in Figure 8 - this is performed at run time.

## Input/Output

The process of IO relies on functions being mutable at runtime and able to interact with the real world. Imperative languages allow state changes and side effects to occur. As java uses the imperative paradigm, it has no problem in completing IO requests.

Haskell however Is a purely functional language; its functions are deterministic as they only ever change based on their arguments and no side effects can be produced. As a result, the Haskell compiler can use the results of another call of a function if the parameters are the same, since the result will be identical. Figure 9 shows an example of this. The function only needs to be called once, and can therefore significantly improve performance.

The problem is purely function programs can't handle impure requests; such as dealing with the real world where external interaction and exceptions occur.

As a solution to this Haskell has implemented monads. Monads are a way of obtaining imperative results in functional programming; it isolates a function or type variable from side effects and non-deterministic values by wrapping it in a high order function. it will insist on an order of execution to read/write data in the correct sequence and call the function more than once even if it has the same arguments, as the outcome could be affected by external interaction.

Built in Haskell monads such as 'IO' and 'Maybe' wrap a primitive data type so that the compiler knows that the value may or may not be available. A basic demonstration of Haskell's IO monad is shown in figure 10. It is a powerful tool that allows the program to continue being purely functional and still usefully interact with the real world.

## Type systems

Type systems are used to reduce the likelihood of errors occurring by creating interfaces between different functions of the program and ensuring that each section has been connected consistently and correctly. It works by assigning a type property to each element of the language. Both Java and Haskell are examples of statically typed systems in which all types are evaluated at compile time and cannot be changed.

Type inference is the inbuilt automated detection of datatypes in the case where type has not been explicitly declared at compile time. It does this by obtaining atomic values categorically through recognising the reduction of expressions. Generics in Java are an example of implicit typing inferred by the compiler however generally a declaration of type is required before the program will run.

Haskell however is built almost exclusively from expressions, meaning its more intuitive for type to be distinguished. GHC does a great job of reduction and logical exclusion of expressions – however it will isolate impure parts of the program into IO actions as well as any occurrence of polymorphic recursion, which commonly requires explicit typing.

## Concurrency

Both Haskell and Java allow for multithread programming, this means on multiprocessor machines computations can be performed on a number of tasks at the same time. To reduce allocated memory and ensure the threads work conjointly the data is shared. This introduces a few integrity issues; if two threads access the same data simultaneously they could obfuscate it in a way which breaks or effects the outcome of both operations.

Java handles this by implementing synchronizers; synchronizers will perform atomic operations which enforces a task to happen all at once or not at all, ensure mutual exclusion so that only one thread can access a data field at a time, coordinate threads to run computations in a logical order and apply barrier synchronizations which halt a thread until others have reached the same point.

Java synchronization works by using monitors which will protect a portion of code so that only one thread can access it at a time until it exits the synchronized block an example of this is shown in figure 11. This can however introduce a new problem called deadlock, in which two (or more) threads are waiting for each other to finish with a task to continue, since neither of them can finish, the threads become stuck and unable to terminate.

The referential transparency of Haskell ensures immutability between shared resources meaning for the most part, many light-weight GHC threads can be created and run no risk of causing corruption. Concurrency is "a structuring technique for effectful code; in Haskell, that means code in the `IO` monad."[2] The inclusion of the library Control.Concurrency allows for new threads to be created/destroyed along with sleeping. Additionally, the IO monad is handled using synchronized mutable variables MVars, as seen in figure 12, which allow communication between synchronous threads of the program.

Generally speaking concurrency is far more efficient and consistent when used in Haskell, the threads are so light-weight hundreds can be used simultaneously on every expression to process a task. Additionally, it runs less risk in causing concurrent data corruption and run-time errors such as deadlock.

## Implications

Both Java and Haskell are very powerful language – it's hard to explicitly declare one better than the other, as they both have preferred use cases.

Haskell is concise and efficient, it can be used to solve complex concurrency/parallelism problems easily using its purely functional style making it ideal for web based and server-side applications, as well as compilers which benefit from elegant, light weight code.

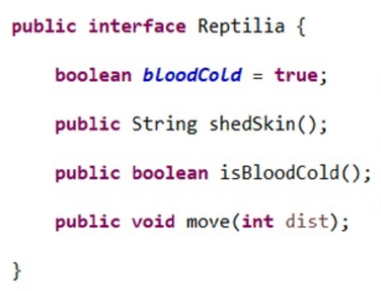
Alternatively, Java is the most popular object-oriented language supported by a vast number of libraries and tools allowing for quick and effective development. Much larger programs benefit from Java's flexibility, hierarchical modularity and extensive polymorphism making it great for desktop/mobile applications.

I believe as a team, Java would be more intuitive and accessible resulting in quicker development. However, given more time to plan and implement, Haskell could be used to create an incredibly reliable and efficient solution.

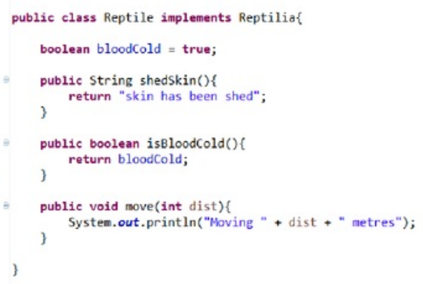
From my research, it seems purely functional languages are hard to ignore in the modern age of computing, more traditional languages are beginning to integrate functional tools such as lambda functions introduced in Java 8. With that in mind it would be wise to continue learning Haskell and skills which constitute functional programming and apply them in the future.

## Appendix

Figure 1)

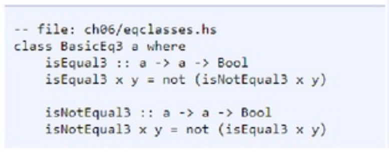


```
public interface Reptilia {

    boolean bloodCold = true;

    public String shedSkin();

    public boolean isBloodCold();

    public void move(int dist);

}
```

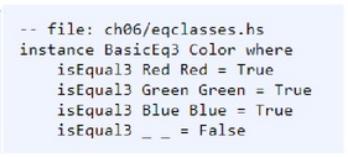


```
public class Reptile implements Reptilia{

    boolean bloodCold = true;

    public String shedSkin(){
        return "skin has been shed";
    }

    public boolean isBloodCold(){
        return bloodCold;
    }

    public void move(int dist){
        System.out.println("Moving " + dist + " metres");
    }

}
```

Figure 2)



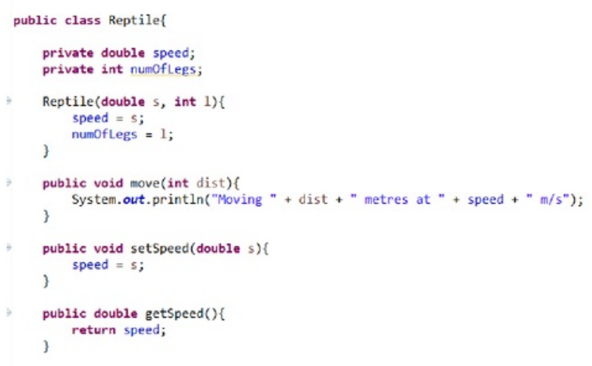
```
-- file: ch06/eqclasses.hs
class BasicEq3 a where
    isEqual3 :: a -> a -> Bool
    isEqual3 x y = not (isNotEqual3 x y)

    isNotEqual3 :: a -> a -> Bool
    isNotEqual3 x y = not (isEqual3 x y)
```



```
-- file: ch06/eqclasses.hs
instance BasicEq3 Color where
    isEqual3 Red Red = True
    isEqual3 Green Green = True
    isEqual3 Blue Blue = True
    isEqual3 _ _ = False
```

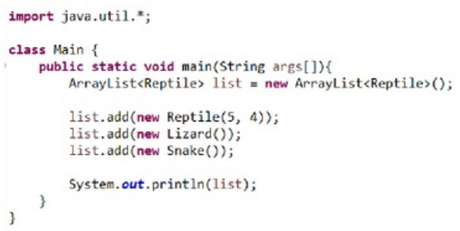
*from source 2*

Figure 3)



```
public class Reptile{

    private double speed;
    private int numOfLegs;

    Reptile(double s, int l){
        speed = s;
        numOfLegs = l;
    }

    public void move(int dist){
        System.out.println("Moving " + dist + " metres at " + speed + " m/s");
    }

    public void setSpeed(double s){
        speed = s;
    }

    public double getSpeed(){
        return speed;
    }
```

Figure 4)

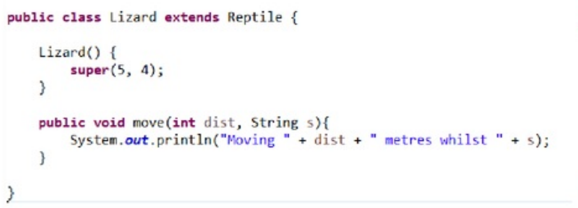


```java
import java.util.*;

class Main {
    public static void main(String args[]){
        ArrayList<Reptile> list = new ArrayList<Reptile>();

        list.add(new Reptile(5, 4));
        list.add(new Lizard());
        list.add(new Snake());

        System.out.println(list);
    }
}
```

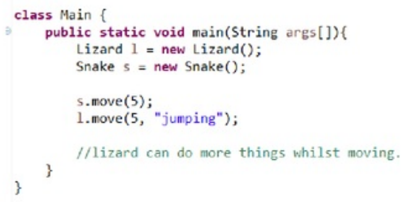
Figure 5)

```haskell
example :: (forall a. a -> a) -> (Char, Bool)
example f = (f 'c', f true)
```

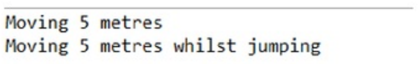
Figure 6)



```java
public class Lizard extends Reptile {

    Lizard() {
        super(5, 4);
    }

    public void move(int dist, String s){
        System.out.println("Moving " + dist + " metres whilst " + s);
    }

}
```

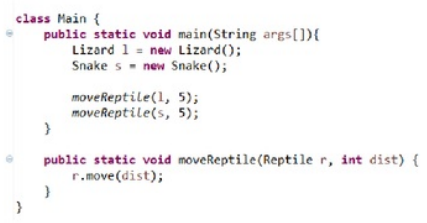


```java
class Main {
    public static void main(String args[]){
        Lizard l = new Lizard();
        Snake s = new Snake();

        s.move(5);
        l.move(5, "jumping");

        //lizard can do more things whilst moving.
    }
}
```



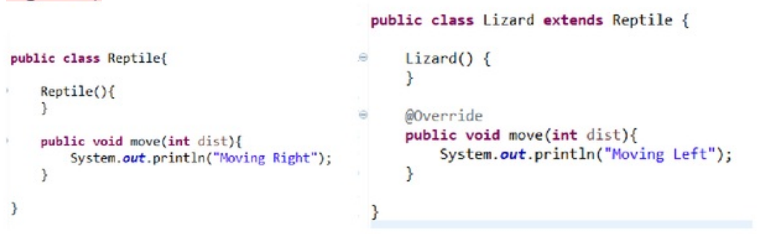
```
Moving 5 metres
Moving 5 metres whilst jumping
```

Figure 7)



```java
class Main {
    public static void main(String args[]){
        Lizard l = new Lizard();
        Snake s = new Snake();

        moveReptile(l, 5);
        moveReptile(s, 5);
    }

    public static void moveReptile(Reptile r, int dist) {
        r.move(dist);
    }
}
```

Figure 8)



```java
public class Reptile{

    Reptile(){
    }

    public void move(int dist){
        System.out.println("Moving Right");
    }

}
```

```java
public class Lizard extends Reptile {

    Lizard() {
    }

    @Override
    public void move(int dist){
        System.out.println("Moving Left");
    }

}
```

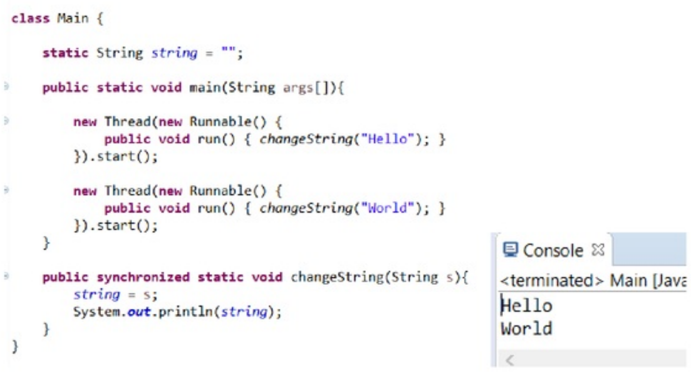
Figure 9)

```haskell
f :: Int -> Int
f x = x^2

y :: Int
y = f 2

z :: Int
z = f 2
```

Figure 10)

```haskell
putStr :: String -> IO ()
putStr s =  sequence_ (map putChar s)
```

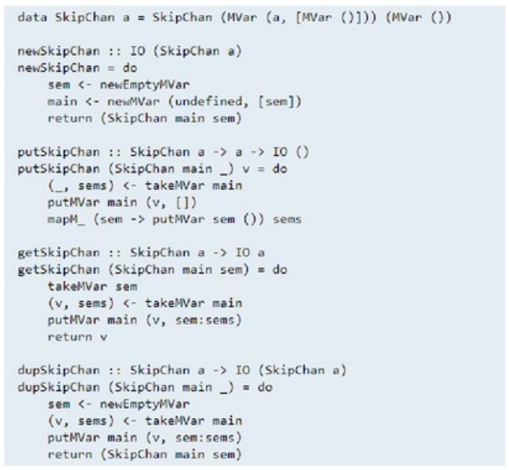
Figure 11)



```
class Main {

    static String string = "";

    public static void main(String args[]){

        new Thread(new Runnable() {
            public void run() { changeString("Hello"); }
        }).start();

        new Thread(new Runnable() {
            public void run() { changeString("World"); }
        }).start();
    }

    public synchronized static void changeString(String s){
        string = s;
        System.out.println(string);
    }
}
```

```
Console
<terminated> Main [Java
Hello
World
```

Figure 12)



```
data SkipChan a = SkipChan (MVar (a, [MVar ()])) (MVar ())

newSkipChan :: IO (SkipChan a)
newSkipChan = do
    sem <- newEmptyMVar
    main <- newMVar (undefined, [sem])
    return (SkipChan main sem)

putSkipChan :: SkipChan a -> a -> IO ()
putSkipChan (SkipChan main _) v = do
    (_, sems) <- takeMVar main
    putMVar main (v, [])
    mapM_ (sem -> putMVar sem ()) sems

getSkipChan :: SkipChan a -> IO a
getSkipChan (SkipChan main sem) = do
    takeMVar sem
    (v, sems) <- takeMVar main
    putMVar main (v, sem:sems)
    return v

dupSkipChan :: SkipChan a -> IO (SkipChan a)
dupSkipChan (SkipChan main _) = do
    sem <- newEmptyMVar
    (v, sems) <- takeMVar main
    putMVar main (v, sem:sems)
    return (SkipChan main sem)
```

*from source 4*

## Bibliography

1. Book
HUGHES, J.
Why functional programming matters
In-text: (Hughes, 1984)
Bibliography: Hughes, J. (1984). Why functional programming matters. 1st ed. Göteborg: Chalmers Tekniska Högskola/Göteborgs Universitet. Programming Methodology Group.

2. Website
CHAPTER 6. USING TYPECLASSES
In-text: (Book.realworldhaskell.org, 2018)
Book.realworldhaskell.org. (2018). Chapter 6. Using Typeclasses. Available at:
http://book.realworldhaskell.org/read/using-typeclasses.htm

3. Book
MARLOW, S.
Parallel and concurrent programming in Haskell
In-text: (Marlow, 2013)
Bibliography: Marlow, S. (2013). Parallel and concurrent programming in Haskell. 1st ed.

4. Website
CONTROL.CONCURRENT.MVAR
In-text: (Hackage.haskell.org, 2018)
Bibliography: Hackage.haskell.org. (2018). Control.Concurrent.MVar. [online] Available at:
https://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Concurrent-MVar.html

A comparison of Java and Haskell

ORIGINALITY REPORT

| 7% | 3% | 4% | 6% |
|---|---|---|---|
| SIMILARITY INDEX | INTERNET SOURCES | PUBLICATIONS | STUDENT PAPERS |

PRIMARY SOURCES

| | | |
|---|---|---|
| 1 | linux.sys-con.com<br>Internet Source | 1% |
| 2 | Submitted to University of Brighton<br>Student Paper | 1% |
| 3 | Submitted to University of Central England in Birmingham<br>Student Paper | 1% |
| 4 | community.haskell.org<br>Internet Source | 1% |
| 5 | Submitted to De Montfort University<br>Student Paper | 1% |
| 6 | Lima, Luis Gabriel, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and Joao Paulo Fernandes. "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language", 2016 IEEE 23rd International Conference on Software Analysis Evolution and Reengineering (SANER), 2016.<br>Publication | 1% |

| | | |
|---|---|---|
| 7 | Submitted to Heriot-Watt University<br>Student Paper | 1% |
| 8 | www.mokehehe.com<br>Internet Source | 1% |
| 9 | "Central European Functional Programming School", Springer Nature, 2012<br>Publication | <1% |

| | | | |
|---|---|---|---|
| Exclude quotes | Off | Exclude matches | Off |
| Exclude bibliography | Off | | |

# A comparison of Java and Haskell