2019

Object oriented software architecture, design & implementation

JOHN VOS

Contents

1.1:

Code:

```
public class Main {

    public static void main (String Args[]) {
        System.out.print("Hello Brighton");
    }
}
```

Test output:

```
<terminated> Main (1) |
Hello Brighton
```

1.2:

Code:

```
import java.util.Scanner;

public class Main {

    public static void main(String args[]){

        System.out.println("Enter desired integer (must be between 1 & 25):");
        Scanner sc = new Scanner(System.in);
        int numb = sc.nextInt();

        while (numb < 1 || numb > 25) {
            System.out.println("number was not between 1 & 25:");
            System.out.println("please enter a number between 1 & 25:");
            numb = sc.nextInt();
        }

        for(int i = 1; i <= 12; i++) {
            System.out.printf("%2d * %1d = %3d \n",i, numb, i*numb);
        }

     }
}
```

1.2 Test output:

```
Enter desired integer (must be between 1 & 25):
0
number was not between 1 & 25:
please enter a number between 1 & 25:
8
 1 * 8 =   8
 2 * 8 =  16
 3 * 8 =  24
 4 * 8 =  32
 5 * 8 =  40
 6 * 8 =  48
 7 * 8 =  56
 8 * 8 =  64
 9 * 8 =  72
10 * 8 =  80
11 * 8 =  88
12 * 8 =  96
```

1.3:

Code:

```
1  public class AccountBetter1 extends Account implements Transfer {
2      public boolean transferFrom (Account from, double amount){
3          if (from.getBalance () >= amount && amount > 0) {
4              from.withdraw(amount);
5              this.deposit(amount);
6              return true;
7          }
8          else {
9                  return false;
10         }
11     }
12
13     public boolean transferTo (Account to, double amount){
14         if (this.getBalance() >= amount && amount > 0){
15             this.withdraw(amount);
16             to.deposit(amount);
17             return true;
18         }
19         else {
20             return false;
21         }
22     }
23 }
24
```

1.3 Test output:

```
Cheat: method deposit is in your submitted class
Cheat: method withdraw is in your submitted class
Cheat: method setOverdraftLimit is in your submitted class
Cheat: method getOverdraftLimit is in your submitted class
Then sending messages to these objects - gives:

ab.deposit( 100.00 )
    ab.getbalance() -> 100.00 a.getbalance() ->   0.00

a.deposit(  50.00 )
    ab.getbalance() -> 100.00 a.getbalance() ->  50.00

ab.transferTo( a,  50.00 )  -> returns true
    ab.getbalance() ->  50.00 a.getbalance() -> 100.00

ab.transferTo( a,  40.00 )  -> returns true
    ab.getbalance() ->  10.00 a.getbalance() -> 140.00

ab.transferTo( a,  -1.00 )  -> returns false
    ab.getbalance() ->  10.00 a.getbalance() -> 140.00

ab.transferTo( a,  10.00 )  -> returns true
    ab.getbalance() ->   0.00 a.getbalance() -> 150.00

ab.transferTo( a,   1.00 )  -> returns false
    ab.getbalance() ->   0.00 a.getbalance() -> 150.00

ab.transferTo( a,  -0.01 )  -> returns false
    ab.getbalance() ->   0.00 a.getbalance() -> 150.00

ab.transferFrom( a,  50.00 )  -> returns true
    ab.getbalance() ->  50.00 a.getbalance() -> 100.00

ab.transferFrom( a,  50.00 )  -> returns true
    ab.getbalance() -> 100.00 a.getbalance() ->  50.00

ab.transferFrom( a,  40.00 )  -> returns true
    ab.getbalance() -> 140.00 a.getbalance() ->  10.00

ab.transferFrom( a,  -1.00 )  -> returns false
    ab.getbalance() -> 140.00 a.getbalance() ->  10.00

ab.transferFrom( a,  10.00 )  -> returns true
    ab.getbalance() -> 150.00 a.getbalance() ->   0.00

ab.transferFrom( a,   1.00 )  -> returns false
    ab.getbalance() -> 150.00 a.getbalance() ->   0.00

ab.transferFrom( a,  -0.01 )  -> returns false
    ab.getbalance() -> 150.00 a.getbalance() ->   0.00
```

1.4:

Code:

```
1  public class AccountBetter2 extends AccountBetter1 implements Interest{
2
3      public boolean inCredit() {
4          if (this.getBalance() >= 0) {
5              return true;
6          }
7          else {
8              return false;
9          }
10     }
11
12     public void creditCharge() {
13         double interest = 0.0;
14         if (this.inCredit() == false) {
15             interest = -(this.getBalance() * 0.00026116);
16         }
17         if (this.getOverdraftLimit() >= (this.getBalance() - interest)) {
18             this.setOverdraftLimit(this.getBalance() - interest);
19         }
20         this.withdraw(interest);
21     }
22 }
```

1.4 Test output:

```
ab.deposit(   100.00 )
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.creditCharge()
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.inCredit()  -> returns true
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.withdraw(   100.00 )  -> returns   100.00
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.creditCharge()
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.inCredit()  -> returns true
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.setOverdraftLimit(  -100.00 )
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.creditCharge()
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.inCredit()  -> returns true
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.withdraw(   100.00 )  -> returns   100.00
    ab.getbalance() ->  -100.00 ab.getOverdraftLimit() ->  -100.00

ab.creditCharge()
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() ->  -100.03

ab.inCredit()  -> returns false
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() ->  -100.03

ab.setOverdraftLimit( -1000.00 )
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() -> -1000.00

ab.inCredit()  -> returns false
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() -> -1000.00
ab.creditCharge()
    ab.getbalance() ->  -100.05 ab.getOverdraftLimit() -> -1000.00

ab.withdraw(   899.00 )  -> returns   899.00
    ab.getbalance() ->  -999.05 ab.getOverdraftLimit() -> -1000.00

ab.inCredit()  -> returns false
    ab.getbalance() ->  -999.05 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.31 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.57 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.84 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() -> -1000.10 ab.getOverdraftLimit() -> -1000.10

ab.creditCharge()
    ab.getbalance() -> -1000.36 ab.getOverdraftLimit() -> -1000.36

ab.creditCharge()
    ab.getbalance() -> -1000.62 ab.getOverdraftLimit() -> -1000.62
```

1.5:

Code:

```
1  public class AccountStudent extends AccountBetter2 {
2
3      private int day = 0;
4
5      public boolean inCredit() {
6          if (this.getBalance() >= 0) {
7              return true;
8          }
9          else {
10             return false;
11         }
12     }
13
14     public void creditCharge() {
15         if (day >= 1) {
16             double interest = 0.0;
17             if (this.inCredit() == false && this.getBalance() < -5000) {
18                 interest = -(this.getBalance() * 0.00026116);
19             }
20             if (this.getOverdraftLimit() >= (this.getBalance() - interest)) {
21                 this.setOverdraftLimit(this.getBalance() - interest);
22             }
23             this.withdraw(interest);
24         }
25         day++;
26     }
27
28 }
29
```

1.5 Test output (part 1):

```
Using the declaration:
AccountBetter2 ab = new AccountBetter2();

Then sending messages to ab - gives:

Cheat: method getBalance is in your submitted class
Cheat: method deposit is in your submitted class
Cheat: method withdraw is in your submitted class
Cheat: method setOverdraftLimit is in your submitted class
Cheat: method getOverdraftLimit is in your submitted class
ab.deposit(   100.00 )
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.creditCharge()
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.inCredit()  -> returns true
    ab.getbalance() ->   100.00 ab.getOverdraftLimit() ->     0.00

ab.withdraw(   100.00 )  -> returns   100.00
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.creditCharge()
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.inCredit()  -> returns true
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->     0.00

ab.setOverdraftLimit(  -100.00 )
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.creditCharge()
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.inCredit()  -> returns true
    ab.getbalance() ->     0.00 ab.getOverdraftLimit() ->  -100.00

ab.withdraw(   100.00 )  -> returns   100.00
    ab.getbalance() ->  -100.00 ab.getOverdraftLimit() ->  -100.00

ab.creditCharge()
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() ->  -100.03

ab.inCredit()  -> returns false
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() ->  -100.03

ab.setOverdraftLimit( -1000.00 )
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() -> -1000.00

ab.inCredit()  -> returns false
    ab.getbalance() ->  -100.03 ab.getOverdraftLimit() -> -1000.00
```

1.5 Test Output (part 2):

```
ab.creditCharge()
    ab.getbalance() ->  -100.05 ab.getOverdraftLimit() -> -1000.00

ab.withdraw(   899.00 )  -> returns   899.00
    ab.getbalance() ->  -999.05 ab.getOverdraftLimit() -> -1000.00

ab.inCredit()  -> returns false
    ab.getbalance() ->  -999.05 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.31 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.57 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() ->  -999.84 ab.getOverdraftLimit() -> -1000.00

ab.creditCharge()
    ab.getbalance() -> -1000.10 ab.getOverdraftLimit() -> -1000.10

ab.creditCharge()
    ab.getbalance() -> -1000.36 ab.getOverdraftLimit() -> -1000.36

ab.creditCharge()
    ab.getbalance() -> -1000.62 ab.getOverdraftLimit() -> -1000.62
```

Essay: The role of inheritance in Java

Inheritance:

Firstly, this essay was written in relevance to Java. Furthermore, inheritance is a method used in programming where a new class derives from an already created class and obtains properties and methods from the existing class; i.e. fields, methods and nested classes. However, a subclass won't inherit members set as private from the parent class, unless, said superclass's methods are set to public / protected, in order to, access the private fields via getters and setters and only by its subclass.

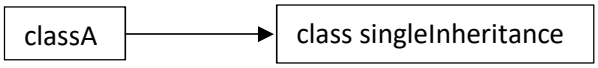
Furthermore, a parent class can be referred to as the "super class" (ancestor) and the class that is inheriting is the "subclass"; except for the object class, which has no super class. Additionally, each class can have only one direct super class (i.e. single inheritance) and therefore, a super class can have *n* number of subclasses; this is due to Java not supporting multiple inheritance. In addition, in the lack of any additional explicit super class, every class is unreservedly a subclass of the Object class.
The keyword being "extends" is used to derive a subclass from the superclass, illustrated by the following syntax: "name_of_subclass *extends* name_of_superclass"

Types of Inheritance:

Single Inheritance:

A class "*extends*" from another class; as shown in the illustration below where class "singleInheritance" extends only "classA" (code demonstrated in appendix single inheritance).

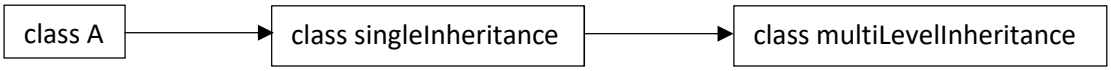
- classA is the super class, class singleInheritance is the sub-class.



Multilevel Inheritance:

A class "*extends*" from a derived class; as shown in the diagram below where the consequential class becomes the "base class" for the new class (code demonstrated in appendix multi-level inheritance).

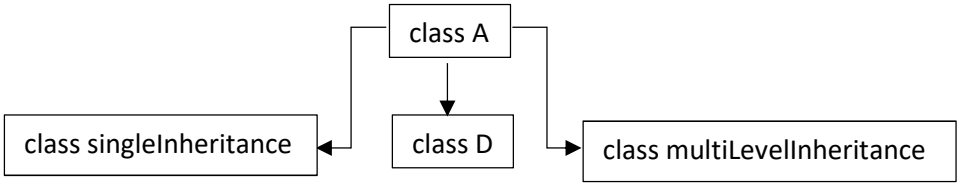
- class multiLevelInheritance is the subclass of singleInheritance, singleInheritance is subclass of class A.



Hierarchical Inheritance:

classA is "inherited" by many sub classes.

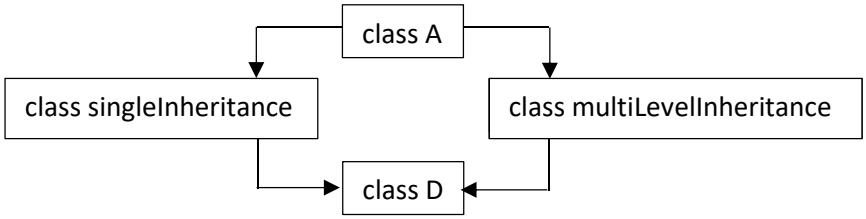
- class singleInheritance, multiLevelInheritance & D inherit from class A.



Hybrid Inheritance:

A classes inheritance is due to a combination of single and multiple inheritance :

- step 1; from class singleInheritance -> step 2; from class multiLevelInheritance.



Advantages of Inheritance:

Firstly, a crucial advantage of inheritance is the ability to diminish the quantity of replica code by distributing shared code amidst multiple subclasses. Furthermore, application code can be made much more flexible to change, due to the fact that classes inheriting from one another can be used interchangeably (if return type = superclass); as a direct result reducing additional repetition and the code being potentially more efficient. Moreover, as a result of being able to share common code, through inheritance, we are able to call upon and re-use public methods without having to rewrite them but also the ability to supersede methods in the base class, in order to, implement more significant implementation to the base class, resulting in a better design to the class being derived. Secondly, through inheritance and the base class we can choose whether to retain particular data that is set to private so that it can't be altered by the derived class.

Disadvantages of Inheritance:

Firstly, one of the major drawbacks to inheritance is the raised time and effort it takes the program to work through all the levels of the classes; for example, if a class has 5 levels above it, it will take 5 jumps to run through a function defined in each of those classes. Additionally, during maintenance and further development of the code, if we add new features it will require amendments to both the base and derived class; for example, if a methods signature is changed then both cases will be affected (inheritance and composition).

In addition, the two classes involved in the inheritance (i.e. base & inherent class) get tightly coupled, resulting in one being unable to be used independently of another. As a result, if a method is deleted in the base class or its collective, re-factoring will have to be done in using that method. Therefore, complicating the code and matters as in the case of inheritance the program will still compile, however, the methods of the subclass will no longer be overriding the base class methods and the methods will become impartial to each other respectively.

Consequences / implication of inheritance:

Firstly, a major consequence inheritance has upon a user's code is when similar code is present in two or more related classes, a user will have the ability to hierarchically refactor common code into a mutual superclass, therefore, resulting in a much better organised, smaller and simpler compilation unit of code. In addition, due to the ability to pull all the variables in common and methods into the superclass with inheritance, more so, and the ability of specialised variables and methods being in the subclass; redundancy can be significantly reduced or even removed as the variables in common and methods are not repeated again in the subclasses.

Key Concepts / Terms:

The role of the superclass & subclass in inheritance:

In object-oriented programming, classes are often organised in a hierarchal manor as to avoid redundancy. Moreover, classes with a lower hierarchal standing inherit all variables (i.e. static attributes) and methods (i.e. dynamic behaviours) from the higher hierarchal classes. Furthermore, a class at the lower end of the hierarchy is called the *subclass*; or is also referred to as being 'derived' or the child or extended class. On the other end, a class at the upper end of the hierarchy is called the *superclass*; also referred to as the base or parent class.

- subclass (child) ➔ class inheriting from another class.
- superclass (parent) ➔ class being inherited from.

Key concepts of Superclass;

- public or protected methods for retrieving private fields can be used in subclass if present in superclass.
- the class from which subclass is derived from, i.e. class that is inherited from.
- each class has a unique explicit superclass (single inheritance); the lack of any other definite superclass, every class is a definite subclass of '*Object*'.

Key concepts of Subclass;

- all public & protected members of the parent are inherited, regardless of the package it is located in.
- members in the parent class set to private cannot be inherited.
  if subclass in same package as parent, also inherits package private members of parent.
- members that are inherited can be used as it, replaced, hidden & supplemented with new members.
- declare fields in subclass with same name as in superclass; resulting in 'hiding' it (though not recommended).
- new static methods can be written in subclass with the same signature in superclass, resulting in 'hiding' it too.
- methods inherited can be used as is.
- new instance method in subclass having the same signature as one in superclass, resulting in 'overriding it'.
- subclass constructors can be created that invoke constructor from the superclass, implicitly by using keyword 'super'.

The role of a constructor in inheritance:

Firstly, when a constructor is used in inheritance it is referred to as 'constructor chaining'; i.e. a subclass's constructor method first task is to call the superclass's constructor method ensuring that the creation of the subclass object begins at initialization of the classes above in the chain of inheritance. Moreover, there can be any number of classes in the chain of inheritance, every individual constructor method will call up said chain until the class at the top of the hierarchy is reached and initialized, once that is done, each subsequent class lower in the hierarchy is initialized as the chain winds back down to the original subclass.

Secondly, a constructer of base class that has no arguments is automatically called in the derived class constructor; however, if we wanted to call the parameterized constructor of the base class then we would need to use super(). Furthermore, the purpose of this is the base class constructor call **must** be the first line in the derived class constructor.

- implicitly calling of the superclass is the same as if the subclass had included the '*super()*' keyword.
- no-args() constructor not being included in a class will result in Java creating one behind the scenes and be conjured;
    - resulting in if the only constructor receives an argument, it must **explicitly** use '*this()*' or '*super()*' as the keyword being invoked.

The role of overriding:

Firstly, if a class inherits a method from its superclass, then there is a chance to override the method provided, in object-oriented terms overriding refers to the ability to override the functionality of an existing method. Moreover, the benefit of this is the ability to clearly define a behaviour that will be specific to the subclass type; i.e. a subclass can implement a parent class method based on its requirements. Secondly, method overriding is used in Java as a means to support runtime polymorphism; this mean that the dynamic method dispatch is the mechanism that calls the overridden method to resolve at run time rather than compile time.

General rules of overriding to follow:

- argument list should be identical in overridden method.
- the return type needs to be identical or a "subtype" of the return type that was declared in the original overridden method in the super class.
- the level of access can't have higher restriction than the methods access level being overridden.
    - e.g. if a variable is declared as "public" it can't then be set as "private" or "protected" when overridden.
- instance methods can only be overridden if inherited by subclass.
- if not inherited it can't be overridden & constructors can't be overridden.

The role of factory method pattern in inheritance:

Firstly, in a class base program "factory method pattern" is the creational pattern that uses factory methods to deal with problems of creating objects without having specific classes for the object that is being created. Furthermore, this is done by creating objects through calling the factory method, i.e. either via being specified in an interface & implemented by child classes or implemented in a base class; optionally, it is overridden by the derived classes rather than by calling constructor. Moreover, factory method in regard to inheritance makes the design more customisable, but, a bit more complicated as other design patterns require new classes, whereas, factory method only requires a new operation.

Appendix Inheritance

singleInheritance class:

```
public class singleInheritance extends classA {

    //a method to display method singleInheritance
    public void dispSingleInheritance() {
        System.out.print("disp() method of single inheritance");
    }

    public static void main(String args[]) {
        //assigns singleInheritance object to singleInheritance reference.
        singleInheritance b = new singleInheritance();

        /*calls method dispA() from classA
         * to display the code and prove inheritance worked.*/
        b.dispA();

        //calls method dispB().
        b.dispSingleInheritance();
    }
}
```

multi-level class:

```
public class multiLevelInheritance extends singleInheritance{

    public void dispmultiLevelInheritance() {
        System.out.println("disp() method of multi-level inheritance");
    }

    public static void main(String args[]) {
        //Assigning multi-level inheritance object to multi-level inheritance reference
        multiLevelInheritance c = new multiLevelInheritance();

        //call dispA() method of ClassA
        c.dispA();

        //call dispB() method of ClassB
        c.dispSingleInheritance();

        //call dispC() method of ClassC
        c.dispmultiLevelInheritance();
    }

}
```

hierarchicalInheritance class:

```
public class hierarchicalInheritance extends classA {

        public void dispHierarchicalInheritance() {
            System.out.println("disp() method of hierarchical inheritance");
        }

    public static void main(String args[]) {
            //Assigning singleInheritance object to singleInheritance reference
            singleInheritance b = new singleInheritance();
            //call dispSingleInheritance() method of singleInheritance
            b.dispSingleInheritance();

            //Assigning multiLevelInheritance object to multiLevelInheritance reference
            multiLevelInheritance c = new multiLevelInheritance();
            //call dispmultiLevelInheritance() method of multiLevelInheritance
            c.dispmultiLevelInheritance();

            //Assigning hierarchicalInheritance object to hierarchicalInheritance reference
            hierarchicalInheritance d = new hierarchicalInheritance();
            //call dispHierarchicalInheritance() method of hierarchicalInheritance
            d.dispHierarchicalInheritance();

            //call dispA() method of ClassA
            d.dispA();
        }
}
```

output:

```
disp() method of single inheritance
disp() method of multi-level inheritance
disp() method of hierarchical inheritance
disp() method of ClassA
```

Works Cited

Anon., n.d. *Oracle.* [Online]
Available at: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
[Accessed Jan 2019].

Anon., n.d. *Techopedia.* [Online]
Available at: https://www.techopedia.com/definition/3226/inheritance-java
[Accessed Jan 2019].

Anon., n.d. *Tutorial Point.* [Online]
Available at: https://www.tutorialspoint.com/java/java_inheritance.htm

Anon., n.d. *Tutorial Point.* [Online]
Available at: https://www.tutorialspoint.com/java/java_overriding.htm
[Accessed Jan 2019].

Anon., n.d. *W3SCHOOLS.* [Online]
Available at: https://www.w3schools.com/java/java_inheritance.asp

Guru, 2019. *Guru99.* [Online]
Available at: https://www.guru99.com/java-class-inheritance.html

Hock-Chuan, C., 2016. *Yet Another Insignificant.... Programming Notes.* [Online]
Available at:
http://www.ntu.edu.sg/home/ehchua/programming/java/j3b_oopinheritancepolymorphism.html
[Accessed Jan 2019].

Leahy, P., 2017. *Thought Co..* [Online]
Available at: https://www.thoughtco.com/constructor-chaining-2034057
[Accessed Jan 2019].

Miglani, G., 2017. *Geek for Geeks.* [Online]
Available at: https://www.geeksforgeeks.org/inheritance-in-java/

Miglani, G., 2017. *Geeks for Geeks.* [Online]
Available at: https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/
[Accessed Jan 2019].

Miglani, T. T. a. G., 2017. *Geeks for Geeks.* [Online]
Available at: https://www.geeksforgeeks.org/overriding-in-java/
[Accessed Jan 2019].

Nizam, A., 2017. *Quora.* [Online]
Available at: https://www.quora.com/What-are-all-the-advantages-of-inheritance-in-Java
[Accessed 2019].

Pavlova, G. F. &. M., n.d. *Source Making.* [Online]
Available at: https://sourcemaking.com/design_patterns/factory_method/java/1
[Accessed Jan 2019].

Shah, C., n.d. *Quora.* [Online]
Available at: https://www.quora.com/What-is-the-main-difference-of-superclass-subclass-and-concrete-class-in-Java

SINGH, C., 2012. *Beginners Book.* [Online]
Available at: https://beginnersbook.com/2013/05/java-inheritance-types/

Waghmare, S., 2014. *Geek For Geeks.* [Online]
Available at: https://www.geeksforgeeks.org/g-fact-67/
[Accessed Jan 2019].