Assignment 2:

1. is Interface and what is an abstract class? What are the differences between them?
   Abstract Class & Interface

2. What are the differences between **overriding** and **overloading**?
   Overriding: when the method signature(method name and parameter types) are the same in superclass and child class. Resolved at run time

Example:
```
public class Person{
    public String name;
    public String getNameCard(){
        return name;
    }
}
public class Employee extends Person{
    public String company;

    @Override
    public String getNameCard(){
       return name + "is a employee in " + company;
    }
}
```

Overload: when two ore more methods in same class with same function name but different parameter typers. Resolved at compile time.

Example:
```
public String getNameCard(String name, String company){
       return name + "is a employee in " + company;
    }
}

public String getNameCard(String name, String company,
int id){
       return name + id+ "is a employee in " + company;
    }
}
```

Override vs Overload

- Overriding implements Runtime Polymorphism whereas Overloading implements Compile time polymorphism.
- The method Overriding occurs between superclass and subclass. Overloading occurs between the methods in the same class.
- Overriding methods have the same signature i.e. same name and method arguments. Overloaded method names are the same but the parameters are different.
- With Overloading, the method to call is determined at the compile-time. With overriding, the method call is determined at the runtime based on the object type.
- If overriding breaks, it can cause serious issues in our program because the effect will be visible at runtime. Whereas if overloading breaks, the compile-time error will come and it's easy to fix.

Polymorphism: perform single actions in different ways
Runtime Polymorphism: When you override a function from a super class, you cannot decided which function are you invoking, JVM will run and decide
Compile time Polymorphism: when function overloading is happening, you can choose which one you want to invoke by passing different parameters

3. What is the **final** keyword? (Filed, Method, Class)
   Generally means " one assigned, cannot be changed"
   final class: a class cannot be derived
   final Field: a variable that is once assigned, cannot be reassigned
   final Method: a method cannot be overridden

4. What is Java garbage collection?
   garbage collection is a mechanism to automatically detect/delete unused objects, so that developers don''t need to care too much about each individual object's lifecycle

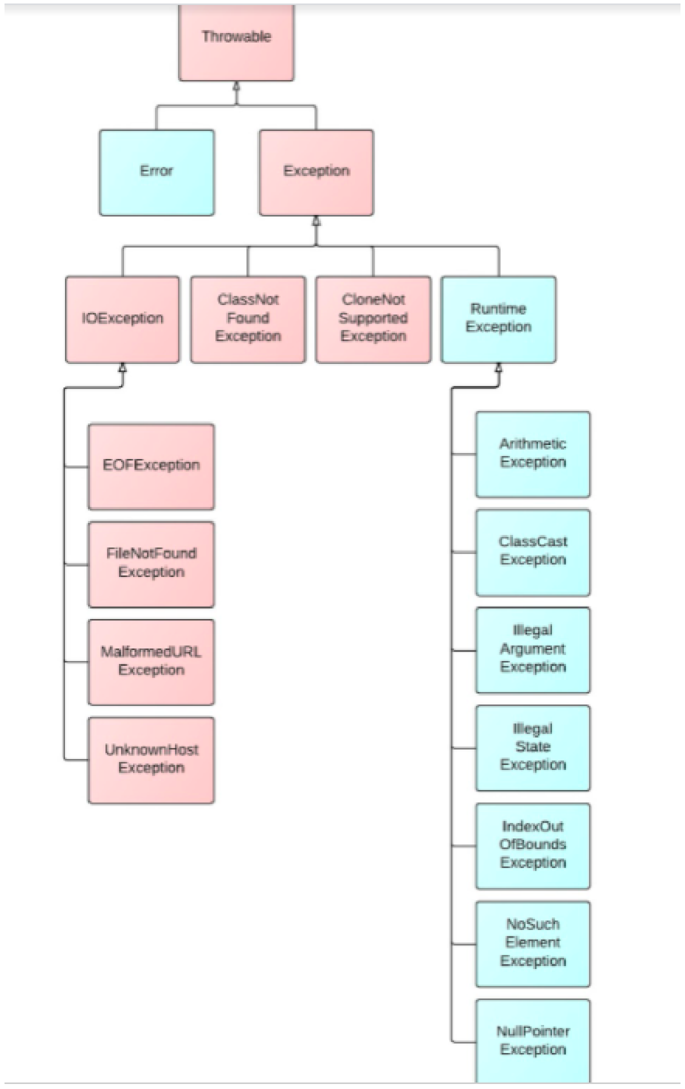
5. What are the differences between **super** and **this**?

key	this	super
represent & reference	this mainly represents current instance of current class	super mainly represents current instance of a parent class
interaction with class constructor	this used to call default constructor of current class	super is used to call default constructor of parent class
Method accessibility	this keyword is used to access methods of current class as it has	super is used to access methods of the parent class

	reference of current class	
Static	his keyword can be referred from static context i.e can be invoked from static instance. For instance we can write System.out.println(this.x) which will print value of x without any compilation or runtime error.	super keyword can't be referred from static context i.e can't be invoked from static instance. For instance we cannot write System.out.println(super.x) this will leads to compile time error.

6. Can we use **this** keyword in the constructor and why?
   yes. We can use this keyword in the constructor. Because this refers to the current class.

7. What is a Runtime/unchecked exception? what is Compile/Checked Exception?



Runtime/unchecked exception: unchecked exceptions are not mandatory to be handles, JVM can take care of it.
Compile/Checked Exception: checked exceptions must be handled by developer during the compile time, otherwise it will cause compile error

Main difference between RuntimeException and checked Exception is that It is mandatory to provide try-catch or try finally block to handle checked Exception and failure to do so will result in a compile-time error, while in the case of RuntimeException this is not mandatory.

8. what is the difference between **throw** and **throws**?
   throw is keyword to explicitly throw an exception in a method or a block of code

Example:
```
public void foo(){
    try{
        throw new NullPointerException;
    }catch(NullPointerException e){
        Systems.out.println("caught in foo");
    }finally(){
        Systems.out.println("nothing caught");
    }
}
```

throws is a keyword used in the head of method to indicate that this method might throw one of the listed exceptions

Example:
```
public void foo() throws NullPointerException{

}

general form: function head+ function signature(parameter
types) throws list of Exceptions
```

9. Run the below three pieces codes, Noticed the printed exceptions. why do we put the Null/Runtime exception before Exception?

   when dealing with exceptions, we need to put more specific exceptions before more general ones. The more specific exception we catch, the better or accurate we can deal with the exceptiona and do more specific responses. Also, if more specific exception is caught, the easier to fix the bug.

```
public class Main {
    public static void main(String[] args) {
        int a = 0;
        int b = 3
        String s = null;
        try {
            System.out.println(b / a); // ArithmeticException, since 0 is
invalid
            System.out.println(s.equals("aa"));// NullPointerException, since s
is null, when dereferencing null will cause NullPointerException
            throw new RuntimeException();
        } catch (ArithmeticException e) {
            e.printStackTrace();
        } catch (NullPointerException e) {
            e.printStackTrace();
        } catch (RuntimeException e) {
            e.printStackTrace(); // RuntimeException, because try block throw a
new RuntimeException and this catch block caught it
        } catch (Exception e) {
            e.getMessage();
        }

        System.out.println("End ...");
    }
}

public class Main {
    public static void main(String[] args) {
        int a = 0;
        int b = 3
        String s = null;
        try {
            // System.out.println(b / a);
            System.out.println(s.equals("aa")); // NullPointerException, since
s is null, when dereferencing null will cause NullPointerException
            throw new RuntimeException();
        } catch (ArithmeticException e) {
            e.printStackTrace();
        } catch (NullPointerException e) {
            e.printStackTrace();
        } catch (RuntimeException e) {
            e.printStackTrace(); // RuntimeException, because try block throw a
new RuntimeException and this catch block caught it
        } catch (Exception e) {
            e.getMessage();
        }

        System.out.println("End ...");
    }
}

public class Main {
    public static void main(String[] args) {
        int a = 0;
        int b = 3
        String s = null;
        try {
            // System.out.println(b / a);
            // System.out.println(s.equals("aa"));
            throw new RuntimeException();
        } catch (ArithmeticException e) {
            e.printStackTrace();
        } catch (NullPointerException e) {
            e.printStackTrace();
        } catch (RuntimeException e) {// RuntimeException, because try block
throw a new RuntimeException and this catch block caught it
            e.printStackTrace();
        } catch (Exception e) {
            e.getMessage();
        }

        System.out.println("End ...");
    }
}
```