MONKE STRONK TGT

Content Page

USE PAGE BREAKS AND START PAGE COUNT SO WE CAN REFER
BODY TIMES NEW ROMAN FONT SIZE 8 PLS its smaller save ink :)
I love timmy @lolkabash <3 -darren

Revision Quizzes

Quiz 2 - Aspects of Java Programming

Q1. What is printed on the screen? Remember that with objects, == checks for aliasing and .equals() is meant to check for equality of contents.
```
BigDecimal b1 = new BigDecimal("1.23");
BigDecimal b2 = new BigDecimal("1.23");
BigDecimal b3 = b1;
System.out.println(b1 == b2); // false
System.out.println(b1.equals(b2)); // true
System.out.println(b1 == b3); // true
```

Q2. What is printed on the screen? Remember that with objects, == checks for aliasing and .equals() is meant to check for equality of contents.
```
String s1 = "pikachu";
String s2 = "pikachu";
String s3 = new String("pikachu");
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s1.equals(s3)); // true
```
Strings go into the Java string pool and in this pool, there is only one instance of the same string literal to save memory.
Thus in the following example, s1 and s2 point to the same object. It would be a logic error to use == if you intend to compare that their contents are the same, thus you should use s1.equals(s2) to do so.
```
String s1 = "apple";
String s2 = "apple";
```

Q3. What is the final statement that is displayed on the screen? No more green bottles
```
for (int i = 0; i <= 10; i++) {
    String out = "Green Bottle " + i;
    System.out.println(out);
}
String out = "No more green bottles";
System.out.println(out);
```
Variables can have local scope within the braces. String out declared at line 2 is only accessible within the for-loop. Thus, when String out is declared later outside the scope of the for-loop, there is no clash in variable names and the code can compile and be executed.

Q4. Fill in the blanks in the following output on the screen. i = 148, a1 = 148, a2 = 149, b = 148.
```
int i = 147;
int a = ++i;
System.out.println(String.format("i=%d a=%d",i, a));
int b = a++;
System.out.println(String.format("a=%d b=%d",a, b));
```
In Java, ++i (pre-increment) increases the value of i first and then uses the updated value, while i++ (post-increment) uses the current value first and then increments it afterwards. For example, if i = 5, int a = ++i; sets a = 6, but int b = i++; sets b = 5 and then increases i to 6.

Q5. Fill in the blanks in the following output on the screen. i = 191, a1 = 190, a2 = 191, b = 191.
```
int i = 190;
int a = i++;
System.out.println(String.format("i=%d a=%d",i, a));
int b = ++a;
System.out.println(String.format("a=%d b=%d",a, b));
```
In Java, ++i (pre-increment) increases the value of i first and then uses the updated value, while i++ (post-increment) uses the current value first and then increments it afterwards. For example, if i = 5, int a = ++i; sets a = 6, but int b = i++; sets b = 5 and then increases i to 6.
Q6. When you need to use an [**object**] in place of a primitive type, the Java platform provides [**wrapper**] classes for each of the [**primitive**] data types. All of the numerical wrapper classes are [**subclasses**] of the [**abstract**] class Number. The following table lists the primitive types and their corresponding wrapper classes.

Primitive Type	**Wrapper Class**
int	**[Integer]**
double	**[Double]**
[float]	Float
short	Short
long	Long
byte	Byte

The blanks below contain code. Fill in the appropriate code.
Recall that when declaring an ArrayList of integers, you need to specify the Integer wrapper class as a type parameter. This is an example of when the wrapper classes are useful. Fill in the following blank so that arrayList variable points to an ArrayList of Integer objects.
```
ArrayList<Integer> arrayList = new ArrayList()<>;
```
The wrapper classes provide useful constants. For example, if you are trying to access the largest possible (i.e. most positive) int value that the language can support, the code is as follows.
```
int upperBound = Integer.MAX_VALUE;
```
How might you access the lower bound on the int value i.e. the most negative?
```
int lowerBound = Integer.MIN_VALUE;
```
You can also use useful static methods parseInt() and valueOf(). You can read about the difference between them.
Given a string s that has numerical characters, fill in the missing code.

```
String s = "123";
int myInt = Integer.parseInt(s); // write code to execute parseInt to use s to obtain an int
Integer myInteger = Integer.valueOf(s); // write code to execute valueOf to use s to obtain an Integer
```

Q7. Read about Java autoboxing and unboxing.
a) Fill in the following blanks with either autoboxing or unboxing. Your answers are not case-sensitive.
Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called [**autoboxing**].
Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called [**unboxing**].
b) Fill in the following blanks with either autoboxed or unboxed. Your answers are not case-sensitive.

```
Integer a = new Integer(200);
int b = a;   // variable a is [unboxed] to the int primitive type
Integer c = 300;  // int value of 300 is [autoboxed] to the Integer wrapper type
```

Q8. Read Converting Between Strings and Numbers

Given the following code, what are the possible expressions you can write in the blank such that String s points to a string "300"? Select all possibilities.

```
int i = 300;
String s = "" + i ; //  "" + i ; String.valueOf(i); Integer.toString(i);
System.out.println(s);  // 300
```

Answers: "" + i ; String.valueOf(i); Integer.toString(i);

The answers are all found in the linked article. Note that Double.toString(i) returns a string that looks like a double, so it has a decimal point. Observe that Integer.toString(i) means that you are calling a static method in the Integer wrapper class. Thus, this is different from the toString() instance method that you usually must override in any class definition.

Q9. Given the following code, what should BLANK_A and BLANK_B be so that NumberFormatException is not thrown or there is no compile-time error?

```
String s1 = "1.23";
String s2 = "300";
int a = BLANK_A ;
double b = BLANK_B;
```

Answers:

```
BLANK_B: Double.parseDouble(s1);
BLANK_A: Integer.parseInt(s2)
BLANK_B: Double.parseDouble(s2);
```

A NumberFormatException is thrown by the static methods parseInt/parseDouble if they are unable to recognize a string as a int or double.

Q10. Which of the following statements is true regarding the following code snippet?

```
String out = "hello";
for (int i = 0; i < 10; i++) {
    String out = "hello";
    System.out.println(out + i);
}
```

As written above, the code will result in a compilation error.
If in this program, only the for-loop in Lines 2 - 5 shown uses the string "hello", it is good practice to declare variables where it is first used and thus the declaration at Line 1 can be removed for the code to compile.
The for-loop in Lines 2 - 5 can access the variable out defined in Line 1, thus the declaration at Line 3 can be removed for the code to compile.

Quiz 3 - Object-Oriented Programming (Classes, Abstract Classes, Interfaces)

Q1. Which of the following principles of Object-Oriented Programming describes the process of putting methods and attributes together to form a class and that it is a good practice to hide your attributes.
Ans: ENCAPSULATION

Q2.
```
public class ServicePhoneNumber {

    public static String callThePolice() {
        return "Police: 999";
    }

    private String service = "";
    private String number = "";

    public ServicePhoneNumber(String service, String number) {
        this.service = service;
        this.number = number;
    }

    @Override
    public String toString() {
        return service + ": " + number;
    }
}
```
In the following class, which of the following statements executes the method callThePolice() from a main() method located in a different class without compilation error?

Declaring a method with static means that it does not require instantiation to be executed, so the new keyword is not required. Simply type ClassName.staticMethodName() to execute it.

Ans: String number = ServicePhoneNumber.callThePolice();

Q3.
```
public class ServicePhoneNumber {
    public static String callThePolice() {
        return "Police: 999";
    }
    private String service = "";
    private String number = "";
    public ServicePhoneNumber(String service, String number) {
        this.service = service;
        this.number = number;
    }
    @Override
    public String toString() {
        return service + ": " + number;
    }
}
```
In the following class, which of the following statements instantiate a ServicePhoneNumber object without compilation error?
Since instantiation is required, you need to use the new keyword. Merely declaring the variable is insufficient. However, since a constructor with three arguments is defined, Java does not implicitly provide a no-arg constructor. Hence, there is no no-arg constructor in this class definition. In this case, if you want a no-arg constructor in the class, you must write the code for it. It is only when there is no constructor defined in the class, then Java implicitly provides a no-arg constructor.

Ans: `ServicePhoneNumber servicePhoneNumber = new ServicePhoneNumber("Non-Emergency Ambulance", "1777");`

Q4.
```
package p1;
public class A {
    private int a;
    public A(int a) {
        this.a = a;
    }
    protected int getA() {
        return a;
    }
    int get2A() {
        return 2 * a;
    }
    public int get3A() {
        return 3 * a;
    }
}
package p2;
import p1.A;
public class Test {
    public static void main(String[] args) {
        A a = new A(2);
        System.out.println(a.get3A());
    }
}
```

Consider the class A in package p1. Class A has methods getA(), get2A() and get3A() with different access modifiers. When in package p2, class A is instantiated in the main() method. Which of the methods of A can be executed?

Since Test and its main method are in a different package, they can only execute methods of A which allow access from another package. One of the methods is package-private (i.e. no access modifier specified) and the other can only be accessed in a subclass of A in package p2. Thus there is only one option remaining.

Ans: a.get3A()

Q5.

```
public abstract class Dog {
    public abstract void bark();
    public void drool() {
        System.out.println("drool");
    }
}

public class Hound extends Dog {
    @Override
    public void bark() {
        System.out.println("bark");
    }
    public void drool(int x) {
        System.out.println("drool" + x);
    }
    public void sniff() {
        System.out.println("sniff");
    }
}
```

Given the following statement written in the same package as the classes above: Dog g = new Hound();

The methods that can be executed on variable g depend on the declared type Dog

Ans:
What will you see on the screen for g.bark() ? bark
What will you see on the screen for g.drool(1) ? Error
What will you see on the screen for g.drool() ? drool
What will you see on the screen for g.sniff() ? Error

Q6

```
class A {
    void f(int x) {
        System.out.println("Af");
    }
    void h(int x) {
        System.out.println("Ah");
    }
}
class B extends A {
    @Override
    void f(int x) {
        System.out.println("Bf");
    }
    void g(int x) {
        System.out.println("Bg");
    }
}
```

Given the following declaration in another class in the same package.
A x = new B();
Which of the following can subsequently be executed?
x.f(1); //statement (i)
x.g(1); //statement (ii)
x.h(1); //statement (iii)

The methods that can be executed for an object depend on the declared type i.e. A x
If there is overriding, the method that the JVM chooses to execute depends on the actual type new B();

Ans: (i) and (iii)

Q7.

```
public interface AnimalSound{
    String makeSound();
}
public class DogSound [a]{
    @Override
   // Fill in a method signature below.
   [b] {
   return "woofwoof";
```

```
    }
}
```

The class above implements this interface and the makeSound() method returns a string "woofwoof". Fill in the blanks.

- An abstract method is a method signature without any body.
- An interface is a special kind of Java "class" which contains only abstract methods.
- An interface exists to ensure that a group of classes has the same behaviour / methods, and acts as a supertype to those classes who promise to implement the interface.
- To promise that a class will implement an interface's abstract method(s), put implements InterfaceName at the class header.
- Then the writer has to keep the promise by providing an implementation of that method(s).
- The signature of the method must be identical to the abstract method(s). Also, realize that all abstract methods in interfaces are understood to have public access.

Ans: a = implements AnimalSound
b = public String makeSound()

Q8.

```
public interface I {
    void m(int x);
}
public class K implements I {
    @Override
    public void m(int x) {
        System.out.println(x);
    }
}
```

Which of the following statements will not result in a compile-time error?
An interface acts as a supertype to all concrete classes that implement it.
Because an interface has abstract methods only, it cannot be instantiated.

Ans: K x = new K();
I x = new K();

Quiz 4 - Exceptions and Exception Handling (Theory)

Checked and Unchecked Exceptions in Java

Checked Exceptions

Checked exceptions represent errors outside the control of the program. For example, the constructor of FileInputStream throws FileNotFoundException if the input file does not exist. Java verifies checked exceptions at compile-time. Therefore, we should use the throws keyword to declare a checked exception:

```
private static void checkedExceptionWithThrows() throws FileNotFoundException {
    File file = new File("not_existing_file.txt");
    FileInputStream stream = new FileInputStream(file);
}
```

We can also use a try-catch block to handle a checked exception:

```
private static void checkedExceptionWithTryCatch() {
    File file = new File("not_existing_file.txt");
    try {
        FileInputStream stream = new FileInputStream(file);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Some common checked exceptions in Java are IOException, SQLException and ParseException.

The Exception class is the superclass of checked exceptions, so we can create a custom checked exception by extending Exception:

```
public class IncorrectFileNameException extends Exception {
    public IncorrectFileNameException(String errorMessage) {
        super(errorMessage);
    }
}
```

Unchecked Exceptions

If a program throws an unchecked exception, it reflects some error inside the program logic.
For example, if we divide a number by 0, Java will throw ArithmeticException:

```
private static void divideByZero() {
    int numerator = 1;
    int denominator = 0;
    int result = numerator / denominator;
}
```

Java does not verify unchecked exceptions at compile-time. Furthermore, we don't have to declare unchecked exceptions in a method with the throws keyword. And although the above code does not have any errors during compile-time, it will throw ArithmeticException at runtime. Some common unchecked exceptions in Java are NullPointerException, ArrayIndexOutOfBoundsException and IllegalArgumentException.

The RuntimeException class is the superclass of all unchecked exceptions, so we can create a custom unchecked exception by extending RuntimeException:

```
public class NullOrEmptyException extends RuntimeException {
    public NullOrEmptyException(String errorMessage) {
        super(errorMessage);
    }
}
```

When to Use Checked Exceptions and Unchecked Exceptions

"If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception."

For example, before we open a file, we can first validate the input file name. If the user input file name is invalid, we can throw a custom checked exception:

```
if (!isCorrectFileName(fileName)) {
    throw new IncorrectFileNameException("Incorrect filename : " + fileName );
}
```

In this way, we can recover the system by accepting another user input file name.
However, if the input file name is a null pointer or it is an empty string, it means that we have some errors in the code. In this case, we should throw an unchecked exception:

```
if (fileName == null || fileName.isEmpty())  {
    throw new NullOrEmptyException("The filename is null or empty.");
}
```

Quiz 4 - Exceptions and Exception Handling

Q1. For the following Exception objects, state whether they are Checked or Unchecked.

Exception	**Enter Checked or Unchecked into the box.**
Arithmetic Exception	Unchecked
ClassNotFoundException	Checked
FileNotFoundException	Checked
IllegalArgumentException	Unchecked
IOException	Checked
SQLException	Checked

Q2. Which of the following code will compile? A, B, C. Not D. Since IllegalArgumentException is an unchecked exception, JVM does not enforce the "handle or declare rule". This means that the programmer is not required to declare it in the method header with the throws keyword or put the method call in a try-catch block. Should a programmer choose to do so nonetheless, it will not result in a compilation error.
Also note that the keyword used in the method header is throws, while another keyword throw is used in a method body to throw an exception.
A:

```
public class TestExceptionOne{
    public static void main(String[] args){
        methodOne();
    }
    public static void methodOne(){
        // has a line which could throw IllegalArgumentException
    }
}
```

```
B:
public class TestExceptionOne{
    public static void main(String[] args){
        methodOne();
    }
    public static void methodOne() throws IllegalArgumentException{
        // has a line which could throw IllegalArgumentException
    }
}
```

```
C:
public class TestExceptionOne{
    public static void main(String[] args){
        try{
            methodOne();
        }catch(IllegalArgumentException ex){
            ex.printStackTrace();
        }
    }
}
```

```
D: (Wrong)
public class TestExceptionOne{
    public static void main(String[] args){
        methodOne();
    }
    public static void methodOne() throw IllegalArgumentException{
        // has a line which could throw IllegalArgumentException
    }
}
```

Q3. Which of the following code will compile? Since FileNotFoundException is a checked exception, JVM enforces the "handle or declare rule" to compile. In this case, since the Exception is not handled in methodOne(), JVM requires the programmer to declare it in the method header with the throws keyword and put the method call in a try-catch block to handle the exception thrown.
Also note that the keyword used in the method header is throws, while another keyword throw is used in a method body to throw an exception.

```
public class TestExceptionOne{
    public static void main(String[] args){
        try{
            methodOne();
        }catch(FileNotFoundException ex){
            ex.printStackTrace();
        }
    }
    public static void methodOne() throws FileNotFoundException{
        // has a line which could throw FileNotFoundException
    }
}
```

Q4. What is printed out? Ans: S. When a method call is placed in a try-block and has an exception being thrown, execution of the program is immediately diverted to the catch-block.

```
public class TestExceptions1 {
    public static void main(String[] args){
        try{
            f(-1);
            System.out.print("R");
        }catch(Exception e){
            System.out.print("S");
        }
    }
    static void f(int x) throws Exception {
        if (x < 0) throw new Exception();
        System.out.print("P");
    }
}
```

Q5. What is printed out? Ans: QR. In method f, the exception thrown in the try-block is handled by the catch-block. Thus, no exception object is handled in the main() method. So, the rest of the lines in the try-block in the main() method are executed.

```
public class TestExceptions2 {
    public static void main(String[] args){
        try{
            f(-1);
            System.out.print("R");
        }catch(Exception e){
            System.out.print("S");
        }
    }
    static void f(int x) throws Exception {
        try{
            if (x < 0) throw new Exception();
            System.out.print("P");
        }catch (Exception e){
            System.out.print("Q");
        }
    }
}
```

Problem Set 1X

Q1a - Adding Time

Write a function that takes two positive hour and minute values, adds them up and returns a string describing the result, in this format:
Example: For hour1 = 1, min1 = 30, hour2 = 1, min2 = 31
`addTime(1, 30, 1, 31)`
Expected output: 3 hours 1 minutes
Note the words ‘hours’ and ‘minutes’ do not need to be adjusted for singular/plural
Assumes:

- Min1, min2 are values between 0 and 59, both values inclusive
- Hour1, hour2 are values between 0 and 59, both values inclusive

Test Case:	**Expected Output:**
`addTime(0, 20, 0, 30)`	`0 hours 50 minutes`
`addTime(0, 40, 0, 30)`	`1 hours 10 minutes`
`addTime(1, 30, 1, 31)`	`3 hours 1 minutes`
`addTime(6, 30, 4, 45)`	`11 hours 15 minutes`

```
 static String addTime(int hour1, int min1, int hour2, int min2){
        int totalMinutes = min1 + min2;
        int extraHours = totalMinutes / 60;
        int remainingMinutes = totalMinutes % 60;
        int totalHours = hour1 + hour2 + extraHours;
        return totalHours + " hours " + remainingMinutes + " minutes";
    }
```

Q1b - Leap Year

The current Gregorian calendar in use was introduced in the year 1582.
The rule for determining if a year is a leap year is as follows:

- A year divisible by four is a leap year.
- But, if the year can also be divided by 100, it is not a leap year, unless it is also divisible by four.

Examples.
The following years are leap years: 1600, 1604, 1996, 2000, 2004.
The following years are not leap years: 1601, 1700, 1800, 1900, 1990, 1993.

The method isLeapYear() takes in a year value that is an int and returns a boolean value.
Assumes:

- year is equal or larger than 1583 and less than 2100.

Test Case:	**Expected Output:**
`isLeapYear(1600)`	`true`
`isLeapYear(1700)`	`false`

```
static boolean isLeapYear(int year){
        if (year % 4 != 0) { // if divisible by 4, its not a leap year
            return false;
          } else if (year % 400 == 0) { // if divisible by 100, and also divisible by 4, its a leap year
            return true;
          } else if (year % 100 == 0) { // if only divisible by 100, its not a leap year
            return false;
          } else {
            return true;
          }
    }
```

Q1c - Pythagoras’ theorem

A method distance() takes in coordinate values x and y that are of double type and returns the distance from the origin also as a double.
Java has no exponentiation operator. By default, the Math library is imported and you can use the methods directly e.g. Math.sqrt(2.0), Math.pow(2.0, 2)
Assume:

- x, y will always be valid double values.

Test Case:	**Expected Output:**
`distance(3.0, 4.0)`	`5.0`

```
    static double distance(double x, double y){
        return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
    }
```

Q2a - Sum up to a given number

Write a function that takes in positive integer n and returns the sum of 1 + 2 + 3 + ... + n - 1 + n
Output is type int.
Assume: n is greater than 0
You may use a for-loop or the sum of series formula
(Test case inputs: 9 Expected output: 45)

Test Case:	**Expected Output:**
`sumUpTo(9)`	`45`

```
    static int sumUpTo( int n){
        return ((n * (n+1))/2); // sum of series formula
    }
```

Q2b - Sum all elements in Array

Suppose that array is a variable of type int[]. Write a function that returns the sum of all elements in an array.
Output is type int.
Assume: Integer overflow will not happen, and the array will have zero or more elements.
(Test case inputs: -12, 3, 20, 21, -15, 2 Expected output: 19)

Test Case:
int a[] = {-12, 3, 20, 21, -15, 2};
int b[] = {};
int c[] = new int[10];
sumIntArrayAll(a); sumIntArrayAll(b); sumIntArrayAll(c);

Expected output:
19 0 0

```
    static int sumIntArrayAll( int[] array){
        int sum = 0;
        for (int num : array) {
            sum += num;
        }
        return sum;
    }
```

Q2c - Sum all elements in Array that are larger than 20

Suppose that array is a variable of type int[]. Write a function that returns the sum of all elements in an array that are larger than 20.
Output is type int.
Assume: Integer overflow will not happen. array can have zero elements or more.
(Test case inputs: -12, 3, 20, 21, -15, 22 Expected output: 43)

Test Case:
int a[] = {-12, 3, 20, 21, -15, 22};
int b[] = {};
sumIntArrayTwenty(a)
sumIntArrayTwenty(b)

Expected output:
43
0

```
    static int sumIntArrayTwenty( int[] array){
        int sum = 0;
        for (int num : array) {
            if (num > 20){
                sum += num;
            }
        }
        return sum;
    }
```

Q2d - Count all even numbers in Array

Suppose that array is a variable of type int[]. Write a function that returns the count of the number of even numbers (which could be positive or negative) stored in array. Output is type int.
Assume:
· Integer overflow will not happen
· 0 is considered an even number
· array will have zero elements or more
(Test case inputs: -12, 3, 0, 21, -15, 2 Expected output: 3)

Test Case (0 is considered even number):
int a[] = {-12, 3, 0, 21, -15, 2};
int b[] = {};

countEvenNumbers(a); countEvenNumbers(b);

Expected output:
3
0

Test Case (array only contains positive integers):
int a[] = {2, 3, 9, 5, 4, 1, 1}
countEvenNumbers(a)

Expected output:
2

```
    static int countEvenNumbers(int[] array){
        int count = 0;
        for (int num : array){
            if (num % 2 == 0){
                count += 1;
            }
        }
        return count;
    }
```

Q3 - Number of terms required

The sum of series 1/1^2 + 1/2^2 + 1/3^2 + … + 1/n^2 converges to pi^2/6:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Write a function that takes in p, a double that can be assumed to be between 0 and 0.999, and returns the number of terms N needed in this series to achieve a sum that is equal or larger than a fraction p of pi^2/6:
Find N for:

$$\frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{(N-1)^2} + \frac{1}{N^2} = \sum_{n=1}^{N} \frac{1}{n^2} \geq \frac{p\pi^2}{6}$$

Output is type int.
Assume: p is between 0 and 0.999
(Test case inputs: 0.9 Expected output: 6)
Hints. The constant pi in Java can be accessed with Math.PI. Use a while loop.

Test Case:
termsRequired(0.9)

Expected output:
6

```
    static int termsRequired(double p){
        double target = p * (Math.PI * Math.PI) / 6;
        double sum = 0.0;
        int n = 1;
        while (sum < target){
            sum += 1.0 /(n*n);
            n++;
        }
        return n - 1;
    }
```

Q4 - Binary to Decimal

Write a function that:
· takes in a String s that contains only the characters ‘0’ and ‘1’.
· s represents a positive binary number
· this function returns the decimal equivalent of this binary number as an integer value
Output is type int.
Assume: Integer overflow will not happen when calculating the result. s has at least one element and will only contain 1 or 0.
(Test case inputs: “1011” Expected output: 11)

Background Information:
1011 (base 2) = 1×23 + 0×22 + 1×21 + 1×20 = 11 (base 10)
Pseudocode (Loop solution)

```
1. Given a string s of length n with characters 1 and 0 only
2. Initialize total to 0
3. For index i from n -1 to 0
   a. Get the character at position i and determine its equivalent int value
   b. Calculate the power of 2 for that position (e.g. power of 2 at position n-1 is 0)
   c. Calculate the equivalent decimal value and add it to the total.
```

Note. If you can, explore more than one way to solve this.

Test Case:
binaryToDecimal(“1011”) binaryToDecimal(“0”)

Expected output:
11
0

```
    static int binaryToDecimal( String s){
        int total = 0;
        for(int i = s.length()-1; i >= 0; i--){
            // get the character at index 1
            // get the value in int type s.charAt(i)
            int value = s.charAt(i) - '0'; // '1' -> 1
            total += value * Math.pow(2, s.length()-1-i);
        }
        return total; // return Integer.parseInt(s, 2) <- also works

    }
```

Q5a - MiddleArray

A method middleArray() takes in an int array a and returns a new int array with the element at index 0 and the last element removed. You will recall that this can be done in Python with list slicing, but this has to be tackled differently in Java.
Explore using
· Arrays.copyRangeOf() (read the documentation yourself) which is available since java.util.Arrays has been imported for you.
· Using a for-loop to copy the elements from one int[] array to another.
o First, determine the length of the new int array. Use this to initialize a new int array. Write a for loop to copy the elements from array a to the new int array.
Assume:
· *a* will have at least two elements.

Test Case:
int[] a = {10, 20, 30};
middleArray(a)

Expected Output:
{20}

```
    static int[] middleArray(int[] a) {
        int[] new_a = Arrays.copyOfRange(a, 1, a.length - 1);
        return new_a;
    }
```

Q5b - Reverse A String

A method reverse() takes in a String a and returns a new string with the order of letters reversed.
Hint: You will need to write a for-loop and use the charAt() and size() method of a String object, as well as concatenation using the + operator.
You will recall that this can be done in Python with an appropriate slice, but this has to be tackled differently in Java.
Assume:
· *a* will have at least two characters.

Test Case:
reverse(“pikachu”)

Expected Output:
uhcakip

```
    static String reverse(String a) {
        String r = "";
        for (int i = a.length() - 1; i >= 0; i--) {
            r += a.charAt(i);
        }
        return r;
    }
```

Problem Set 1A

Q1 - Fibonacci Numbers Generator

Write a JAVA program that returns the first n numbers in the fibonacci sequence, in this format:
Eg. For n = 5 the output is
0,1,1,2,3
When submitting, return these numbers in this format as a string, instead of printing.

```
public class Fibonacci{
    public static String fibonacci( int n) {
        if (n <= 0) {
            return "";
        }
        String result = "";
        int a = 0, b = 1;
        for (int i = 0; i < n; i++) {
            result += a;
            if (i < n - 1) {
                result += ",";
            }
            int temp = a + b;
            a = b;
            b = temp;
        }
        return result;
    }
}

public class Main {
    public static void main(String[] args) {
        int n;
        n = Integer.parseInt(args[0]);
        String ans = "";

        Fibonacci fibo = new Fibonacci();
        ans = fibo.fibonacci(n);

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

Q2a - Iterating with Iterator

Suppose that *integers* is a variable of type List<Integer>. Write a program that uses an iterator to compute the sum of all integer values in the List.
(Test case inputs: (1, 2, 3, 4, 5) Expected output: 15)

```
import java.util.*;
public class IteratingExamples {
    public static int Act2Iterator(List<Integer> integers) {
        int sum = 0;
        // Insert code here to sum up input using an Iterator ...
        Iterator<Integer> it = integers.iterator();
        while(it.hasNext()) {
            sum += it.next();
        }
        return sum;
    }
}


import java.util.*;
public class Main {
    public static void main(String[] args){
        int n = 10;
        // int n = Integer.parseInt(args[0]); // use this if passing command-line arguments

        List<Integer> integerList = new ArrayList<>(); // Generate the input ArrayList
        for( int i = 1; i <= n; i++){
            integerList.add(i);
        }
        //Recall that 1 + 2 + .. + n = n(n+1)/2.
        String ans = "" + IteratingExamples.Act2Iterator(integerList);
        ans = "Iterator Sum = " + ans;
        System.out.println(ans);
    }
}
```

Q2b - Iterating with For-Each

Write a second program that does the same thing as in the previous question but using a for-each loop.
(Test case inputs: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) Expected output: 55)

```
import java.util.*;
public class IteratingExamples {
    public static int Act2ForEach(List<Integer> integers) {
        int sum = 0;
        for (int num : integers) {
            sum += num;
        }
        return sum;
    }
}

import java.io.*;
import java.util.*;
public class Main{
    public static void main(String[] args){
        int n = 10;
        // int n = Integer.parseInt(args[0]); // use this if passing command-line arguments
        List<Integer> integerList = new ArrayList<>(); // Generate the input ArrayList
        for( int i = 1; i <= n; i++){
            integerList.add(i);
        }
        //Recall that 1 + 2 + .. + n = n(n+1)/2.
        String ans = "" + IteratingExamples.Act2ForEach(integerList);
        ans = "ForEach Sum = " + ans;
        System.out.println(ans);
    }
}
```

Q3 - The Account Class

Design a class name Account that contains:
Private int data field named id for the account (default 0)
Private double data field named balance for the account (default 0)
Private double data field named annualInterestRate that stores the current interest rate (in percentage, default 0). Assume all accounts have the same interest rate i.e. declare this data field static.
A private Date data field named dateCreated that stores the date when the account was created. (use java.util.Date)
A no-arg constructor that creates a default account
A constructor that creates an account with the specified id and initial balance
The accessor and mutator methods for id, balance, and annualInterestRate
The accessor method for dateCreated
A method named getMonthlyInterestRate() that returns the monthly interest rate
A method named getMonthlyInterest() that returns the monthly interest
A method named withdraw that withdraws a specified amount from the account
A method named deposit that deposits a specified amount to the account
Write a test program that creates an Account object, with withdraw and deposit method to withdraw / deposit the amount, and print the balance, monthly interest and the date when the account was created. Note that the account balance is allowed to be negative.

Test case:

```
public class TestAccount{
        public static void main (String[] args) {
                Account account = new Account(1122, 20000);
                Account.setAnnualInterestRate(4.5);
                account.withdraw(2500);
                account.deposit(3000);
                System.out.println("Balance is " + account.getBalance());
                System.out.println("Monthly interest is " + account.getMonthlyInterest());
        }
}
```

Expected output:

```
Balance is 20500.0
Monthly interest is 76.875
```

```
import java.util.Date;
public class Account {
    private int id = 0;
    private double balance = 0;
    private static double annualInterestRate = 0;  //all accounts have the same interest rate -> static
    private Date dateCreated;

    public Account() {
        this.dateCreated = new Date();
    }
    public Account(int id, double balance) {
        this.id = id;
        this.balance = balance;
        this.dateCreated = new Date();
    }
    public int getId() {
        return id;
    }
```

```java
    public void setId(int id) {
        this.id = id;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
    public static double getAnnualInterestRate() {
        return annualInterestRate;
    }
    public static void setAnnualInterestRate(double rate) {
        annualInterestRate = rate;
    }
    public Date getDateCreated() {
        return dateCreated;
    }
    public double getMonthlyInterestRate() {
        return annualInterestRate / 12;
    }
    public double getMonthlyInterest() {
        return balance * (getMonthlyInterestRate() / 100); // Monthly interest in percentage
    }
    public void withdraw(double amount) {
        balance -= amount;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount; // Add amount to balance
        } else {
            System.out.println("Deposit must be positive.");
        }
    }
}
```

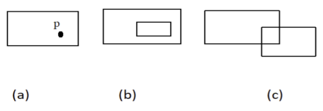
Q1 - Prime Number Checker

Write a static method, that reads in a number (you can assume that the input number is always >=3). Return 1 if it is prime, return 0 if it is not prime.
Hints: use %. a%b= remainder of a/b. e.g. 13%5=3, 4%2=0
Test case inputs: 4, 7, 14, 23, 99
Expected outputs: 0, 1, 0, 1, 0

```java
public class PrimeNumberChecker{
    public static void main(String[] args) {
        System.out.println(isPrime(14));
        int[] checkPrimesList = {2,12,13,45,47};
        for (int i : checkPrimesList){
            System.out.println(isPrime(i));
        }
    }
    public static int isPrime(int num){
        for (int i = 2; i <= Math.sqrt(num); i++){
            if (num % i ==0){
                return 0;
            }
        }
        return 1;
    }
}
```

Q2 - The MyRectangle2D class

Define the MyRectangle2D class that contains:
• Two double data fields named x and y that specify the center of the rectangle with get and set methods: getX, setX, getY, setY. (Assume that the rectangle sides are parallel to x- or y- axes.)
• The double data fields width and height with get and set methods: getWidth, setWidth, getHeight, setHeight.
• A no-arg constructor that creates a default rectangle with (0, 0) for (x, y) and 1 for both width and height.
• A constructor that creates a rectangle with the specified x, y, width, and height: MyRectangle2D(double x, double y, double width, double height)
• A method getArea() that returns the area of the rectangle.
• A method getPerimeter() that returns the perimeter of the rectangle.
• A method contains(double x, double y) that returns true if the specified point (x, y) is inside this rectangle. See Figure 1(a).
• A method contains(MyRectangle2D r) that returns true if the specified rectangle is inside this rectangle. See Figure 1(b).
• A method overlaps(MyRectangle2D r) that returns true if the specified rectangle overlaps with this rectangle. See Figure 1(c).



Implement data fields, all constructors, methods getArea(), getPerimeter(), and contains(double x, double y), contains(MyRectangle2D r), and overlaps(MyRectangle2D r)
Test case:
MyRectangle2D b = new MyRectangle2D(10,20,60,20);
b.contains(-18,12); // true

```java
public class MyRectangle2D {
    private double x;
    private double y;
    private double width;
    private double height;

    public MyRectangle2D() {
        this.x = 0;
        this.y = 0;
        this.width = 1;
        this.height = 1;
    }
    public MyRectangle2D(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public double getX(){
        return x;
    }
    public void setX(double x){
        this.x = x;
    }
    public double getY(){
        return y;
    }
    public void setY(double y){
        this.y = y;
    }
    public double getWidth(){
        return width;
    }
    public void setWidth(double width){
        this.width = width;
    }
    public double getHeight(){
        return height;
    }
    public double setHeight(double height){
        this.height = height;
        return height;
    }
    public double getArea(){
        return width * height;
    }
    public double getPerimeter(){
        return width + width + height + height;
    }
    public boolean contains(double x, double y) {
        return Math.abs(x - this.x) <= width / 2 && Math.abs(y - this.y) <= height / 2;
    }
    public boolean contains(MyRectangle2D secondRectangle) {
        double secondRectangleX = secondRectangle.getX();
        double secondRectangleY = secondRectangle.getY();
        double secondRectangleWidth = secondRectangle.getWidth();
        double secondRectangleHeight = secondRectangle.getHeight();
        return secondRectangleX - secondRectangleWidth / 2 >= this.x - this.width / 2 &&
                secondRectangleX + secondRectangleWidth / 2 <= this.x + this.width / 2 &&
                secondRectangleY - secondRectangleHeight / 2 >= this.y - this.height / 2 &&
                secondRectangleY + secondRectangleHeight / 2 <= this.y + this.height / 2;
    }
    public boolean overlaps(MyRectangle2D secondRectangle) {
        return !(secondRectangle.getX() - secondRectangle.getWidth() / 2 > this.x + this.width / 2 ||
                secondRectangle.getX() + secondRectangle.getWidth() / 2 < this.x - this.width / 2 ||
                secondRectangle.getY() - secondRectangle.getHeight() / 2 > this.y + this.height / 2 ||
                secondRectangle.getY() + secondRectangle.getHeight() / 2 < this.y - this.height / 2);
    }
}
```

Q3 - Range of Numbers

Write a static method getRangeOfInts() that takes in three ints: start, end, increment. It returns an ArrayList<Integer> object whose elements form a sequence of integers from start, up to but not including end, increasing by increment. Assume that the end is always larger than the start and the increment is positive, and there is no integer overflow.

Test Case:
getRangeOfInts(2, 10, 1)
getRangeOfInts(2, 10, 2)
Output:

```
[2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8]
```

```java
import java.util.ArrayList;
import java.util.List;

public class RangeOfNumbers {
    public static void main(String[] args) {
        System.out.println( getRangeOfInts(0, 10, 1));
    }
    public static List<Integer> getRangeOfInts (int start, int end, int increment){
        List<Integer> outlist = new ArrayList<>();
        for (int i = start; i < end; i+=increment){
            outlist.add(i);
        }
        return outlist;
    }
}
```

Q2 - Triangle class

Design a class named Triangle that extends GeometricObject. The class contains:
Three double data fields named side1, side2, side3 with default value 1.0 to denote three sides of the triangle.
A no-arg constructor to create a default triangle
A constructor that creates a triangle with the specified side1, side2, and side3
A method named getArea() that returns the area
A method named getPerimeter() that returns the perimeter
A method named toString() that returns description of the triangle
Triangle: side1 = 1.0 side2 = 2.0 side3 = 3.0
Write a test program to test the code. The program should create the Triangle object with these sides and color and filled properties set.

Test case:

```
public class TestTriangle {
        public static void main(String[] args) {
                Triangle myTri = new Triangle();
                myTri.setColor("red");
                myTri.setFilled(true);
                System.out.println(myTri);
                System.out.println(myTri.isFilled());
                Triangle myTri2 = new Triangle(2.0, 4.0, 5.5);
                System.out.println(myTri2);
                System.out.println("area : " + myTri2.getArea() + " perimeter: " + myTri2.getPerimeter());
        }
}
```

Output:

```
Triangle: side1 = 1.0 side2 = 1.0 side3 = 1.0
true
Triangle: side1 = 2.0 side2 = 4.0 side3 = 5.5
area : 3.0714155938264036 perimeter: 11.5
```

```java
class Triangle extends GeometricObject {
    private double side1;
    private double side2;
    private double side3;

    public Triangle(){
        this.side1 = 1.0;
        this.side2 = 1.0;
        this.side3 = 1.0;
    }
    public Triangle(double side1, double side2, double side3){
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }
    public double getArea() {
        double s = getPerimeter() / 2;
        return Math.sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }
    public double getPerimeter(){
        return (side1 + side2 + side3);
    }
    @Override
    public String toString() {
        return "Triangle: side1 = " + side1 + " side2 = " + side2 + " side3 = " + side3;
    }
    public double getSide1() {
        return side1;
    }
    public void setSide1(double side1) {
        this.side1 = side1;
    }
    public double getSide2() {
        return side2;
    }
    public void setSide2(double side2) {
        this.side2 = side2;
    }
    public double getSide3() {
        return side3;
    }
    public void setSide3(double side3) {
        this.side3 = side3;
    }
}
```

Q3 - Subclasses of Account

In Week-1, the Account class was defined to model a bank account. An account has the properties: account id, balance, annual interest rate, and date created, and methods to deposit and withdraw funds. Create a subclass for checking account CheckingAccount. A checking account has an overdraft limit of 5000. Provide constructors for CheckingAccount similar to Account. Override withdraw() to print out "over limit" if the amount withdrawing exceeds the overdraft limit.
Test Case:

```
public class TestCheckingAccount {
        public static void main(String[] args) {
                CheckingAccount myCheckAcc = new CheckingAccount(1024, 8000.0);
                myCheckAcc.deposit(2000);
                myCheckAcc.withdraw(15000);
                System.out.println(myCheckAcc.getBalance());
                myCheckAcc.withdraw(200);
                System.out.println(myCheckAcc.getBalance());
                myCheckAcc.deposit(7000);
                myCheckAcc.withdraw(200);
                System.out.println(myCheckAcc.getBalance());
        }
}
```

Output:

```
-5000.0
over limit
-5000.0
1800.0
```

```
public class CheckingAccount extends Account{
    private double overdraftLimit = 5000.0;
    public CheckingAccount(){
        super();
    }
    public CheckingAccount(int id, double balance){
        super(id, balance);
    }
    @Override
    public void withdraw(double amount){
        double lastBalance = getBalance() - amount;
        if (lastBalance >= -overdraftLimit) {
            setBalance(lastBalance);
        } else {
            System.out.println("over limit");
        }
    }
}
```

Q4 - String Operation

(Part-I) Design and implement a static method to determine if an input string has all unique characters. Assume the character set is ASCII, which encodes 128 characters into 7-bit binary integers.
(Part-II) Design and implement a static method to determine if two input strings are permutation of each other. Assume the character set is ASCII, which encodes 128 characters into 7-bit binary integers.
Test case

```
public static void main(String[] args) {
        System.out.println(Pset1.isAllCharacterUnique("abcdefghijklmnopqrstuvABC"));
        System.out.println(Pset1.isAllCharacterUnique("abcdefgghijklmnopqrstuvABC"));
        System.out.println(Pset1.isPermutation("@ab", "a@b"));
        System.out.println(Pset1.isPermutation("abcd", "bcdA"));
}
```

Output:

```
true
false
```

```
true
false
```

```java
import java.util.Arrays;
public class Pset1 {
    public static boolean isAllCharacterUnique(String sIn) {
        for (int i = 0; i < sIn.length(); i++) {
            for (int j = i + 1; j < sIn.length(); j++) {
                if (sIn.charAt(i) == sIn.charAt(j)) {
                    return false;
                }
            }
        }
        return true;
    }
    public static boolean isPermutation(String s1, String s2) {
        int n1 = s1.length();
        int n2 = s2.length();
        if (n1 != n2){
            return false;
        }
        char ch1[] = s1.toCharArray();
        char ch2[] = s2.toCharArray();
        Arrays.sort(ch1);
        Arrays.sort(ch2);

        for(int i = 0; i < n1; i++){
            if (ch1[i] != ch2[i]){
                return false;
            }
        }
        return true;
    }
}
```

Problem Set 1B

Q1 - Encapsulation

Create a class named Person.
· Implement the class with the following private attributes: name (type String), age (type int), gender (type char), sharingConsent (type boolean).
· Provide constructor that takes 4 arguments (name, age, gender, and sharing consent.) Inputs are used to initiate the attributes of the same name.
· Provide public getter and setter methods to access and modify age attribute.
· Provide public getter for name and sharingConsent attributes. If sharingConsent is true, name getter will return the name, otherwise return a String "Anonymous".
· Provide the toString() method which returns a String. See the test case below. The String returned does not have any leading or trailing spaces.
Additionally, create a class named Filter. Implement a static method named seniorFilter. The method takes an array of Person objects and return an ArrayList of String containing the names of Person objects whose age equals to or above 60 years old.

```java
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Objects;
public class FilterPersonExample {
    // No need to modify this class
    public static void main(String[] args) {
        Person p1 = new Person("A", 90, 'F', false);
        Person p2 = new Person("B", 60, 'M', true);
        Person p3 = new Person("C", 30, 'F', true);
        Person[] p = {p1, p2, p3};

        System.out.println( Filter.seniorFilter(p) );
    }
}
class Filter {
    public static ArrayList<String> seniorFilter(Object p) {
        ArrayList<String> resultList = new ArrayList<>();

        for (int i = 0; i < Array.getLength(p); i++) {
            Person thisPerson = (Person) Array.get(p, i);
            // Person[] persons = (Person[]) p;
            // Person thisPerson = persons[i];
            if (thisPerson.getAge() >= 60) {
                resultList.add(thisPerson.getName());
            }
        }
        return resultList;
    }
}
class Person {
    private String name;
    private int age;
    private char gender;
    private boolean sharingConsent;

    public Person(String name, int age, char gender, boolean sharingConsent) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.sharingConsent = sharingConsent;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return this.age;
    }
    public String getName() {
        if (!getSharingConsent()) {
            return "Anonymous";
        }
        return this.name;
    }
    public boolean getSharingConsent() {
        return this.sharingConsent;
    }
    public String toString() {
        return "Person: " + this.name + " " + this.gender + " " + this.age;
    }
}
```

Q2 - Polymorphism

· Create a class Animal with a method makeSound. There is no need to define a constructor. This method takes no input and return a String object “I am just an animal” .
· Create 3 subclasses: Dog, Cat, and Cow. Each subclass should override the makeSound() method to represent the unique sound that each animal makes. The String each subclass return for Dog, Cat, and Cow are “Woof”, “Meow”, and “Moo”.
· Create a subclass of Cat named SiberianCat. The class does not contain any method.
· Create a class called AnimalConcert that has a static method named performConcert.
o The performConcert method should take an array of Animal objects as a parameter and return a String object as its output. The String is the sound of each animal joined with a comma and a space “, “.

```java
import java.util.ArrayList;
public class TestSound {
    // You are not required to modify this class
    public static void main(String[] args) {
        Animal[] animals = {new Dog(), new Cat(), new Cow(), new SiberianCat()};

        System.out.println( AnimalConcert.performConcert(animals) );
    }
}

// define all classes (AnimalConcert, Animal, Dog, Cat, Cow, SiberianCat) below this line
class Animal {
    public String makeSound() {
        return "I am just an animal";
    }
}

class Dog extends Animal {
    @Override
    public String makeSound() {
        return "Woof";
    }
}

class Cat extends Animal {
    @Override
    public String makeSound() {
        return "Meow";
    }
}

class Cow extends Animal {
    @Override
    public String makeSound() {
        return "Moo";
    }
}

class SiberianCat extends Cat {

}

class AnimalConcert {
    public static String performConcert(Animal[] animals) {
        ArrayList<String> concert = new ArrayList<>();

        for (Animal animal : animals) {
            concert.add(animal.makeSound());
        }

        return String.join(", ", concert);
    }
}
```

Q3 - BaseInteger

An integer of arbitrary base can be represented as follows.
· "1,14,4" in Base 17 would be calculated in decimal value as: $1\times17^2+14\times17^1+4\times17^0=531$
· "43,5" in Base 60 would be: $43\times60^1+5\times60^0=2585$
In this problem, we assume that the base is an integer and $2 \leq \text{base} \leq 60$, and the integer represented is 0 or positive.
The constructor takes in two inputs:
· representation - a string with digits separated by commas and no spaces eg. "22,43,5"
· base - a positive integer from 2 to 60
These are assigned to the instance variables representation and base respectively.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class BaseInteger {
    private String representation;
    private int base;
    private int[] digits;
    private int decimalValue;

    BaseInteger( String representation, int base){
        this.representation = deleteSpaces(representation);
        this.base = base;
        convertRepresentationToArray();
        setDecimalValue();
    }
    private void convertRepresentationToArray(){
        String[] stringDigits = this.representation.split(",");
        this.digits = new int[stringDigits.length];

        for (int i = 0; i < stringDigits.length; i++) {
            this.digits[i] = Integer.parseInt(stringDigits[i]);
        }
    }
    private void setDecimalValue(){
        int i = this.digits.length - 1;
        int total = 0;

        for (int x : this.digits) {
            total += (x * Math.pow(this.base, i));
            i -= 1;
        }
        this.decimalValue = total;
    }
    public int getDecimalValue() {
        return decimalValue;
    }
    public String getDigitsString(){
        return Arrays.toString(digits);
    }
    public BaseInteger add(BaseInteger other, int base){
        int sum = this.getDecimalValue() + other.getDecimalValue();
        String newRepresentation = convertBase(sum, base);
        return new BaseInteger(newRepresentation, base);
    }
    private String convertBase(int decimalValue, int base){
        ArrayList<String> newRepresentation = new ArrayList<String>();

        if (decimalValue == 0) {
            newRepresentation.add("0");
        }

        while (decimalValue > 0) {
            int remainder = decimalValue % base;
            newRepresentation.add(Integer.toString(remainder));
            decimalValue /= base;
        }

        Collections.reverse(newRepresentation);
        return newRepresentation.toString().replaceAll("[\\[\\] ]", "");
    }

    private String deleteSpaces( String representation){
        return representation.replaceAll("\\s+","");
    }

    @Override
    public String toString() {
        return representation + " Base " + base;
    }
}
```

Q4 - RSA Algorithm

RSA (Rivest–Shamir–Adleman) algorithm is one of the algorithms for data encryption. It involves 4 steps:
1. Public and Private key generation by a server. The key generation steps are:

- Choose 2 prime numbers p and q
- Compute modulus, n=pq
- Compute Carmichael function, l=lcm(p-1, q-1), where lcm is the least common multiple. The formula of lcm(a,b) is |ab| / gcd(a,b), where gcd is the greatest common divisor.
- Choose an integer e such that 2<e<l , gcd(e, l)=1, and e and l are coprime, their only common factor/divisor is 1.
- Calculate d, the modular multiplicative inverse of e modulo l. Private key consists of modulus n and d

2. Public key (n,e) distribution to a client
3. Data encryption by the client. Ciphertext c = me mod n. The value of n and e are the public key, and m is the message character as an integer.
4. Data decryption by the server. The integer character m can be recovered using the following formula, m = cd mod n. The value of n and d are private key, and c is the encrypted character in integer.

In this program, we will simulate a text message encryption and decryption. Server generates public key and private key. Public key is distributed to browser (client) and private key is kept secret by the server. Client can use the public key to encrypt a message while server can decrypt the message using the private key.

1. class Server

a) Complete the generatePublicPrivateKey method. There are 6 TODOs here:

i. Compute modulus n
ii. Compute lambda l
iii. Compute e using computE method.
iv. Compute d using computeModInverse method.
v. Set (n,e) as the publicKey attribute. publicKey is an integer array of size 2. First element is the modulus n and the second is e
vi. Set (n,d) as the privateKey attribute. privateKey is an integer array of size 2. First element is the modulus n and the second is d

b) Complete the lcm (least common multiple) and gcd (greatest common divisor) methods.

c) Complete the decryptMessage method. This method has BigInteger array as the input parameter. Each element of the BigInteger array represents the encrypted character. Each character needs to be decrypted and then its ASCII value is converted to a character. Concatenate all characters to output a String, which is the original message from the client (Browser).

2. class Browser

a) Complete the establishConnection method. This method has 1 input parameter: a Server object. There are 2 TODOs here:

i. Invoke generatePublicPrivateKey of the Server object.
ii. Get the publicKey from the server and set it as the Browser object's publicKey attribute.

b) Complete the encryptMessage method. This method has 1 input parameter: a String object to be encrypted. The output of this method is a BigInteger array, where each item represents the encrypted character of the input message. Hint: You can loop through each character of the String message, cast it to integer (you can convert character into its ASCII value in Java by casting char to int), and then using modPow method from BigInteger object to encrypt the character

```
import java.math.BigInteger;

public class Server {
    // DO NOT CHANGE THIS PART OF THE CODE ======================
    private final int[] publicKey = new int[2];
    private final int[] privateKey = new int[2];
    private int p;
    private int q;

    public void setP(int p) {
        this.p = p;
    }

    public void setQ(int q) {
        this.q = q;
    }

    public int[] getPublicKey() {
        return publicKey;
    }

    public int[] getPrivateKey() {
        return privateKey;
    }

    private int computeModInverse(int e, int lambda) {
        for (int d=1; d<lambda; d++) {
            if ( ((e%lambda)*(d%lambda))%lambda==1 ) {
                return d;
            }
        }
        return 1;
    }
    private int computeE(int lambda) {
        for (int i=lambda-1; i>2; i-- ) {
            if (lambda%i!=0 && isPrime(i)) {
                return i;
            }
        }
        return 0;
    }
    private boolean isPrime(int a) {
        for (int i = 2; i<a/2; i++) {
            if (a%i==0) {
                return false;
            }
        }
        return true;
    }
    // =========================================

    // Start your answer from here onwards
    public void generatePublicPrivateKey() {
        // TODO 1: Compute modulus n
        int n = p*q;
        // TODO 2: Compute lambda λ
        int lambda = lcm(p-1, q-1);
        // TODO 3: Compute e
        int e = computeE(lambda);
        // TODO 4: Compute d
        int d = computeModInverse(e, lambda);
        // TODO 5: Set (n,e) as the public key
        this.publicKey[0] = n;
        this.publicKey[1] = e;
        // TODO 6: Sset (n,d) as the private key
        this.privateKey[0] = n;
        this.privateKey[1] = d;
    }

    public String decryptMessage(BigInteger[] encryptedIntMessage) {
        BigInteger[] decryptedIntMessage = new BigInteger[encryptedIntMessage.length];
        String outputString = "";

        // Hint:
        // 1. Decrypt each character of the message. Use .modPow from BigInteger
        // 2. Decrypted character is an ASCII value (integer). Convert to char
        // 3. Concatenate characters into string
        // 4. Return the decrypted string message
        for (int i = 0; i < encryptedIntMessage.length; i++) {
            decryptedIntMessage[i] = encryptedIntMessage[i].modPow(BigInteger.valueOf(this.privateKey[1]),
BigInteger.valueOf(this.privateKey[0]));
            int ascii = decryptedIntMessage[i].intValue();
            char c = (char) ascii;
            outputString += c;
        }
        return outputString;
    }

    private int lcm(int a, int b) {
        // Return the least common multiple of a and b

        return (Math.abs(a*b) / gcd(a,b));
    }

    private int gcd(int a, int b) {
        // Return the greatest common divisor of a and b

        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
}

import java.math.BigInteger;

public class Browser {
    private final int[] publicKey = new int[2]; // DO NOT CHANGE THIS DATA FIELD

    public void establishConnection(Server s) {
        // TODO 1: Generating public and private key by the server
        // TODO 2: Get the public key from the server and use it to set Browser object's public key
        s.generatePublicPrivateKey();
        this.publicKey[0] = s.getPublicKey()[0];
        this.publicKey[1] = s.getPublicKey()[1];
    }

    public BigInteger[] encryptMessage(String message) {
        BigInteger[] encryptedIntMessage = new BigInteger[message.length()];

        // Hint:
        // 1. Loop through each character of the message
        // 2. Convert to its ASCII value in integer
        // 3. Encrypt the value. Use .modPow method from BigInteger
        for (int i = 0; i < message.length(); i++) {
            char c = message.charAt(i);
            int ascii = (int) c;
            encryptedIntMessage[i] = BigInteger.valueOf(ascii).modPow(BigInteger.valueOf(this.publicKey[1]),
BigInteger.valueOf(this.publicKey[0]));
        }
        return encryptedIntMessage;
    }
}
```

Problem Set 2A

Q1 - Abstract Classes

Write a concrete class Triangle that is a sub-class of GeometricObjectMod. It also initializes the instance variables filled and colour with their default values using the superclass constructor. Search online for Heron’s formula to calculate the area of a triangle from the length of three sides.

```
public abstract class GeometricObjectMod {
    private String colour;
    private boolean filled;

    GeometricObjectMod(){
        colour = "white";
        filled = false;
    }
    GeometricObjectMod(String colour, boolean filled){
        this.colour = colour;
        this.filled = filled;
    }
    public abstract double getArea();

    public abstract double getPerimeter();

    public String getColour() {
        return colour;
    }

    public boolean isFilled() {
        return filled;
    }
}

public class Triangle extends GeometricObjectMod {

    private double a;
    private double b;
    private double c;

    Triangle( double a, double b, double c){
        // call the superclass-constructor here according to the requirements of the question
        super();
        this.a = a;
        this.b = b;
        this.c = c;}

        @Override // Not a must, but v recommended to put Override. Must have matching signatures
        public double getArea(){
            double s = 0.5 * (a + b + c);
            double area = Math.sqrt(s * (s -a) * (s - b) * (s-c)); // Heron’s formula
            return area;
        }
        @Override
        public double getPerimeter(){
            double perimeter = a + b + c;
            return perimeter;
        }
}
```

Q2 - Interface

You are given the following interface Dog. The programmer envisions a family of Dog classes, all of which have the same behaviour i.e. makeSound and wagtail. Write two concrete classes Husky and Retriever, both of which implement Dog. The constructor of Husky and Retriever each take in and initialize a private String instance variable name.

```
public interface Dog {
    String makeSound();
    String wagTail();
}
public class Husky implements Dog{
    private String name;
    Husky(String name){
        this.name = name;
    }
    @Override
    public String makeSound() {
        return name + " howls";
    }
    @Override
    public String wagTail() {
        return name + " wags tail";
    }
}
public class Husky implements Dog{
    private String name;
    Husky(String name){
        this.name = name;
    }
    @Override
    public String makeSound() {
        return name + " howls";
    }
    @Override
    public String wagTail() {
        return name + " wags tail";
    }
}
```

Q3 - Comparable

Modify this Octagon class to implement the Comparable<Octagon> interface to allow sorting of Octagon objects based on their perimeters.
An Octagon class implementing the Comparable<Octagon> interface allows a List of Octagon objects to be sorted by a natural ordering (in this case, the perimeter).

```
public class Octagon implements Comparable<Octagon> {
    private double side;
    public Octagon(double side){
        this.side = side;
    }
    public double getSide() {
        return side;
    }
    public double getPerimeter() {
        return side*8;
    }

    @Override // compareTo: -1 returned if left < right, 0 if equal, 1 if left > right
    public int compareTo(Octagon s) {
        return Double.compare(this.getPerimeter(), s.getPerimeter());
    }
}
```

Q4 - Comparator

Implement a OctagonComparator class to implement the Comparator<Octagon> interface to allow sorting of Octagon objects based on their perimeters. In this case, the Octagon does not implement any interface. You design a separate class called OctagonComparator which implements the Comparator<Octagon> interface. The OctagonComparator class specifies how a List of Octagon objects is sorted (in this case, the perimeter) and an object of this class is passed to the Collections.sort() method.

```
import java.util.*;
public class OctagonComparator implements Comparator<Octagon> {
    //then implement the method(s) in the interface
    @Override
    public int compare(Octagon o1, Octagon o2) {
        double perimeter1 = o1.getSide() * 8;
        double perimeter2 = o2.getSide() * 8;
        return Double.compare(perimeter1, perimeter2);
    }
}
```

Difference between, Q3 Comparable and Q4 Comparator

Features	Comparable	Comparator
Definition	It defines natural ordering within the class.	It defines external sorting logic.
Method	compareTo()	compare()
Implementation	It is implemented in the class.	It is implemented in a separate class.
Sorting Criteria	Natural order sorting	Custom sorting
Usage	It is used for single sorting order.	It is used for multiple sorting orders.

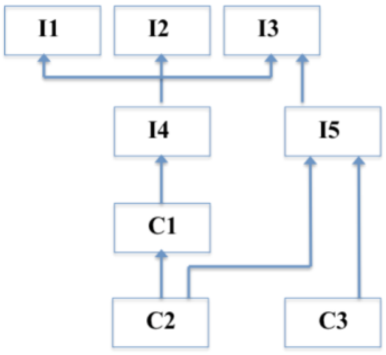
Q5 - Sieve of Eratosthenes (Efficient algorithm for finding list of Prime Numbers up to a number)

ArrayList<T> and LinkedList<T> are concrete implementations of the List<T> interface. Since the List<T> interface specifies abstract methods add(), get() and so on, you have the guarantee that for an ArrayList<T> object or LinkedList<T> object, you are able to execute the methods add() and get(). In the class Sieve, write a static method getPrimeNumbers that returns any List<Integer> object (i.e. ArrayList<Integer> or any other) that contains all the prime numbers from 2 up to and including n. Assume that $2 \leq n \leq 100$.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Sieve {

    public static void main(String[] args) {

        List<Integer> list = getPrimeNumbers(25);
        System.out.println(list);
    }

    public static List<Integer> getPrimeNumbers(int n){

        boolean prime[] = new boolean[n+1]; // Creating an Array of size n filled with trues
        for (int i = 0; i <= n; i++){
            prime[i] = true;
        }
        for (int p = 2; p*p <= n; p++){
            if (prime[p] == true){
                for (int i = p*p; i <= n; i += p){ // Algorithm summary: Mark out all multiples of this number
                    prime[i] = false; // until n. Repeat for next prime number until p*p <= n.
                }
            }
        }
        List<Integer> numberList = new ArrayList<>();
        for (int s = 2; s <=n; s++){
            if (prime[s] == true){
                numberList.add(s); // Add all prime numbers marked true to list to be outputted
            }
        }
        return numberList;
    }

}
```

Q6 - Inheritance and Interface

Develop the software using inheritance and interfaces as shown in the following figure. Note that an arrow from P to Q means that P is a subclass/sub-interface of Q, or P implements Q.



```java
public class TestClass {
    //DO NOT MODIFY THIS METHOD
    public static void main(String[] args) {
        C2 x = new C2();
        C3 y = new C3();
    }
}
interface I1{
    int p1();
}
interface I2{
    int p2();
}
interface I3{
    int p3();
}
interface I4 extends I1, I2, I3 {
    int p4();
}
interface I5 extends I3 {
    int p5();
}
abstract class C1 implements I4 {
    public abstract int q1();
}
class C2 extends C1 implements I5 {
    @Override
    public int p1() {
        return 0;}
    @Override
    public int p2() {
        return 0;}
    @Override
    public int p3() {
        return 0;}
    @Override
    public int p4() {
        return 0;}
    @Override
    public int p5() {
        return 0;}
    @Override
    public int q1() {
        return 0;}
}
class C3 implements I5 {
    @Override
    public int p5() { return 0;}
    public int p3() { return 0;}
}
```

Q7 - Recursion

Complete the static methods to implement the recursive algorithms for these static methods.

```
public class Recursion {
    public static void main(String[] args) {
        System.out.println(isPalindrome("abca"));
        System.out.println(factorial(5));
    }

    public static boolean isPalindrome( String s ){
        int length = s.length();
        if (length == 1 || length == 0) {
            return true;
        }
        if (s.charAt(0) != s.charAt(length - 1)){
            return false;
        }
        return isPalindrome(s.substring(1, length -1));
    }

    public static int factorial( int n){
        if (n == 0 || n == 1){
            return 1;
        }
        else {
            return n * factorial(n-1);
        }
    }
}
```

Q8 - Permutation

A class Permutation takes in either a String or an int at its constructor and initializes the in instance variable. The client of this class calls the permute() method to generate all the permutations of the characters in String or the digits in int, which is stored in the instance variable a. Then by calling getA(), a List object containing all the permutations is obtained. If permute() is not called first, then getA() returns an empty List object. Your task in this question is to complete the constructor Permutation(int number) so that in is initialized. You are also to complete the private methods permuteString() and swap().

```java
import java.util.ArrayList;
import java.util.List;

public class Permutation {
    private final String in;
    private final List<String> a = new ArrayList<>();

    // additional attribute if needed
    Permutation(String str){
        in = str;
    }

    Permutation(int number){
        // complete this constructor correctly
        in = String.valueOf(number);
    }

    public Permutation permute(){
        // calls the recursive function to produce the permuted sequence of 'in' and store in 'a', recursively
        permuteString(in, 0);
        return this;
    }

    private void permuteString(String s, int i){
        int n = s.length();
        if (i == n) {
            a.add(s);
            return;
        }
        else {
            for (int j = i; j < n; j++) {
                s = swap(s, i, j);
                permuteString(s, i+1);
                s= swap(s, i, j);
            }
        }
    }

    /* complete this function to swap the characters at position i and j of s
    and return a new string*/
    private String swap(String s, int i, int j){
        char[] arr = s.toCharArray();
        char temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        return new String(arr);
    }

    public List<String> getA(){
        return a;
    }
}
```

Q9 - Exceptions

In readLineFromFileOne(), put a try-catch block around the appropriate line(s) of code. If there is no exception thrown, the method should return the string out. If an exception is thrown, return the output of the getMessage() method of the exception object. Remember that the catch block should catch FileNotFoundException.
In readLineFromFileTwo(), modify the header to specify that FileNotFoundException is thrown.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TestExceptions {

    public static void main(String[] args) {
        System.out.println( readFileByLineOne("apple"));
        try{
            System.out.println( readFileByLineTwo("orange"));
            System.out.println( readFileByLineTwo("apple"));
        }catch(FileNotFoundException ex){
            ex.printStackTrace();
        }
        /* You should see this:
        apple (No such file or directory) <-- this is from readFileByLineOne("apple")
        Africanorangevalenciaorange <-- this is from readFileByLineTwo("orange")
        at java.base/java.io.FileInputStream.open0(Native Method) <-- this is from readFileByLineTwo("apple")
        at java.base/java.io.FileInputStream.open(FileInputStream.java:213)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:152)
        at java.base/java.util.Scanner.<init>(Scanner.java:645)
        at TestExceptions.readFileByLineTwo(TestExceptions.java:38)
        at TestExceptions.main(TestExceptions.java:11) */

    }

    /* put a try-catch block in this method at the appropriate location
     * in the catch block, return the output of the getMessage() method of the exception object
    */
    public static String readFileByLineOne(String filename){
        File file = new File(filename);
        String out = "";
        try{
            Scanner scanner = new Scanner(file);
            while(scanner.hasNext()){
                out = out + scanner.next();
                }
            scanner.close();
            }
        catch (FileNotFoundException ex) {
                return ex.getMessage();
        }
        return out;
    }

    /* simply modify the method signature to declare that a specific checked exception is thrown*/
    public static String readFileByLineTwo(String filename) throws FileNotFoundException{
        File file = new File(filename);
        Scanner scanner = new Scanner(file);
        String out = "";
        while(scanner.hasNext()){
            out = out + scanner.next();
        }
        scanner.close();
        return out;
    }
}
```

Problem Set 2B

Q1 - Recursive Fibonacci

Write a class called Fibonacci. This class is meant to implement a cached version of the recursive algorithm of the Fibonacci series. You will see that this reduces the number of recursive calls.

Constructor. The constructor takes in one int parameter max which initializes the number of elements of the int array data. Thus, in the constructor

- Initialize the instance variable max
- Instantiate the array data to have max elements
- data[0] shall be 0 and data[1] shall be 1

public int getFibonacciNumber(int n).

- Initialize the calls instance variable to 0. This instance variable tracks the number of times getFibRecursive() is called.
- Returns -1 if n is greater than or equal to max, otherwise this method returns the n-th Fibonacci number by returning the result of executing fibRecursive(n).

private int fibRecursive(int n). This is the recursive method which implements the cached version of the recursive algorithm of the Fibonacci series. Here's the pseudocode.

```
1. Increase calls by 1
2. If n is 0 or 1, return data[n], else
    a. If data[n] is not zero, return data[n]
    b. Else, data[n] = fibRecursive(n-1) + fibRecursive(n-2) and return data[n]
```

```java
public class Fibonacci {
    private int[] data;
    private int max;
    private int calls;

    Fibonacci(int max){
        this.max = max;
        this.data = new int[max];
        data[0] = 0;
        data[1] = 1;
    }
    public int getFibonacciNumber(int n){
        this.calls = 0;
        if (n >= max) {
            return -1;
        }
        else {
            return fibRecursive(n);
        }
    }
    private int fibRecursive(int n){
        this.calls += 1;
        if (n == 0 || n == 1) {
            return data[n];
        }
        else {
            if (data[n] != 0) {
                return data[n];
            }
            else {
                data[n] = fibRecursive(n-1) + fibRecursive(n-2);
                return data[n];
            }
        }
    }
    public int getCalls(){
        return calls;
    }
    public int[] getData(){
        return data;
    }
}
```

Q2 - Custom Stack and StackImpl

You are given the following interface CustomStack which takes in a type parameter T.

```
public interface CustomStack<T> {
    void push(T t);
    T pop();
    int size();
    T peek();
    boolean isEmpty();
}
```

Write a concrete class StackImpl<T> which implements CustomStack. Write a no-arg constructor to initialize the List<T> myList instance variable with any concrete class that implements the List<T> interface (eg. ArrayList).

Implement the abstract methods.

- push() adds element t to the top of the stack
- pop() removes the element at the top of the stack and returns it. If the stack has no elements, return null
- peek() returns the element at the top of the stack, but does not remove it. If the stack has no elements, return null.
- size() returns the number of elements in the stack.
- isEmpty() returns true if the stack has no elements, and false otherwise.

```
import java.util.*;
public class StackImpl<T> implements CustomStack<T>{
    public List<T> dataStorage;
    public StackImpl() {
        dataStorage = new ArrayList<>();
    }
    @Override
    public void push(T t) {
        dataStorage.add(t);
    }
    @Override
    public T pop() {
        if (dataStorage.isEmpty()) {
            return null;
        }
        return dataStorage.remove(getLastIndex());
    }
    @Override
    public int size() {
        return dataStorage.size();
    }
    @Override
    public T peek() {
        if (dataStorage.isEmpty()) {
            return null;
        }
        return dataStorage.get(getLastIndex());
    }
    private int getLastIndex(){
        return dataStorage.size() - 1;
    }
    @Override
    public boolean isEmpty() {
        return dataStorage.isEmpty();
    }
}
```

Q3 - Check Balanced Brackets

Write a static method isBalancedBrackets() to take in a String that contains open brackets, close brackets, and characters which are letters of the alphabet. Open brackets are "({[" and closing brackets are ")}]". You can assume that this string is at most 20 characters long. If the brackets in the string are balanced, return true, otherwise return false.

Implement a set of static helper methods, public static boolean isOpenBracket(), isCloseBracket() and isMatchBracket(). You'll need to use a Stack implementation. Pseudocode:

```
1. Initialize a stack.
2. For every char c in the string
    a. If c is an opening bracket, push to stack
    b. If c is a closing bracket,
        i. pop a character d from the stack
        ii. If both characters are matching open/close brackets, continue, else brackets are not balanced, return false
3. If the stack is empty, return true, else return false otherwise.
```

```java
public class CheckBalancedBrackets {
    public static boolean isBalancedBrackets( String expression ){
        StackImpl<Character> stack = new StackImpl<>();
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            if (isOpenBracket(c)) {
                stack.push(c);
            } else if (isCloseBracket(c)) {
                if (stack.isEmpty()) {
                    return false;
                }
                char open = stack.pop();
                if (!isMatchBracket(open, c)) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
    public static boolean isOpenBracket( char c){
        String brackets = "({[";
        return brackets.indexOf(c) != -1;
    }
    public static boolean isCloseBracket(char c){
        String brackets = ")}]";
        return brackets.indexOf(c) != -1;
    }
    public static boolean isMatchBracket(char c1, char c2){
        if (c1 == '(' && c2 == ')') {
            return true;
        }
        if (c1 == '{' && c2 == '}') {
            return true;
        }
        if (c1 == '[' && c2 == ']') {
            return true;
        }
        return false;
    }
}
```

Q4 - FixExpression

Abstract class FixExpression is given to you. Abstract class is meant to be a superclass of concrete classes that will either represent InFix or PostFix expressions and calculate their result. Infix expression: 2+3*4, Postfix expression: 234*+

A string containing the Infix or Postfix expression is passed to the constructor. Regardless of whether the expression is an Infix or Postfix expression, the first task is to check that the string contains only valid characters, as defined in the instance variable validChars.

- Declare FixExpression as an abstract class.
- Complete the method isValidString() to return true if expression has valid characters, and false otherwise.
- To test your implementation of isValidChars(), declare a concrete class which is a subclasss of FixExpression and provide a trivial implementation of the abstract method. See next page.

```java
public abstract class FixExpression {
    private String expression;
    private String validChars = "0123456789+-*/";

    // you may encounter issues if this is not declared public
    public FixExpression(String expression){
        this.expression = expression;
    }

    // complete this method
    public boolean isValidString(){
        for (char c: expression.toCharArray()){
            if (validChars.indexOf(c) == -1)
            {
                return false;
            }
        }
        return true;
    }

    public String getExpression() {
        return expression;
    }

    public String getValidChars() {
        return validChars;
    }

    public abstract int evaluateExpression();
}
```

Q5 - PostFixExpression

Write a concrete class PostFixExpression that is a child class of FixExpression. This means you have to implement evaluateExpression() to evaluate a postfix expression using your concrete stack class developed in Question 2.

Make PostfixExpression a child class of FixExpression. Write the constructor accordingly if it is not given to you already. Implement evaluateExpression(). If expression contains invalid characters, return Integer.MIN_VALUE otherwise, you may assume that expression contains a valid postfix expression with the assumptions stated in the previous question. Then, implement evaluateExpression() according to the algorithm below.

```
1.  Initialize a stack.
2.  For each char c in expression,
        a.  If c is an operand, get its int value and push it to stack
        b.  Else c is an operator and do the following,
                i.    pop the stack and assign the result to p1
                ii.   pop the stack and assign the result to p2
                iii.  p3 is the result of the operation p2 c p1.
                iv.   push p3 to the stack 3. pop the stack and return the result
```

You may also need the following private methods. It is up to you how to declare these methods. The autograder does not test your private methods.

- isOperator() – returns true if a char is an operator, and false otherwise.
- isOperand() – returns true if a char is an operand, and false otherwise.
- getValue() – given operands p1, p2 and operator c, return the result of the operation between them

```java
public class PostfixExpression extends FixExpression {

    public PostfixExpression(String expression){
        super(expression);
    }

    @Override
    public int evaluateExpression(){
        CustomStack<Integer> stack = new StackImpl<>();

        for (char c : getExpression().toCharArray()) {
            if (isOperand(c)) {
                stack.push(Character.getNumericValue(c));
            } else if (isOperator(c)) {
                if (stack.isEmpty()){
                    return Integer.MIN_VALUE;
                    }
                int p1 = stack.pop();
                if (stack.isEmpty()){
                    return Integer.MIN_VALUE;
                    }
                int p2 = stack.pop();
                int result = getValue(p2, p1, c);
                stack.push(result);
            }
        }

        return stack.pop();
    }

    private boolean isOperand(char c) {
        if (Character.getNumericValue(c) >= 0 && Character.getNumericValue(c) <= 9){
            return true;
        }
        else{
            return false;
        }
    }
    private boolean isOperator(char c){
        if (c == '+' || c =='-' || c =='/' || c == '*'){
            return true;
        }
        else{
            return false;
        }
    }
    private int getValue(int p2, int p1, char operator) {
        switch (operator) {
            case '+': return p2 + p1; // why switch hereeeeeeeeeeeeeeee
            case '-': return p2 - p1;
            case '*': return p2 * p1;
            case '/': return p2 / p1; // Assumes no division by zero (valid input assumption)
            default: throw new IllegalArgumentException("Invalid operator: " + operator);
        }
    }
}
```

Lecture Examples

Example 19 - Writing a Class

Write a class for a Student.

- It has instance variables / attribute name (String), id (int) and yearOfStudy(int). which must be declared private due to encapsulation.
- It has one constructor that takes in and initializes these variables. Make it default / package-private.
- It has one no-arg constructor that initializes these instance variables with the default values "Jane Doe", 0, 0. Make it default / package-private.
- It has getters for each of the instance variables. You do not need to write setters.
- It has a public method getStudentInfo() which returns a String in the format [name] [id] Year [yearOfStudy.]

In a main method, instantiate a Student object with any name, ID, yearOfStudy you like and execute the getStudentInfo() method.

```java
public class Student {

    private String name;
    private int id;
    private int yearOfStudy;
    Student(String name, int id, int yearOfStudy){
        // name -local variable, this.name is your attribute / instance variable
        this.name = name;
        this.id = id;
        this.yearOfStudy = yearOfStudy;
    }
    // No-arg constructor -> no arguments
    Student(){
        // calling your three-arg constructor above
        this("Hoshino Ai", 0, 0);
    }

    // Code-> Generate -> Getters
    public String getName() {
        return name;
    }
    public int getId() {
        return id;
    }
    public int getYearOfStudy() {
        return yearOfStudy;
    } // write setter for year of study
    public String getStudentInfo(){ // no need this.name etc, understood to be attribute
        int id = 1000; // id has a name conflict ==> you want to specify attribute, use this
        return String.format("%s %d Year %d",name, this.id, yearOfStudy);
    }
}
```

Example 20 - Superclass/Subclass

Let's say we have a class called Employee. To return textual information about it, you can define a toString() method.
Then, after instantiating an Employee instance, simply pass the reference to System.out.println() , and toString() will be automatically called.
We will explain the @Override annotation later.

You specify the superclass using the extends keyword.
private variables/methods are not inherited since they are accessible only within the same class.
Constructors are not inherited – you must define them.

- The first line in your subclass constructor must be a call to the **super()** method.
- This accesses the constructors in the superclass. e.g. If you supply two parameters to super(), then you call the superclass constructor with two arguments.
- If you do not have a specified call, the JVM puts a call to **super()** by default i.e. the no-arg constructor in the superclass is called by default.

To change the implementation of any accessible method in the superclass, you can override in the subclass.

- You must ensure that the method signature is the same.
- To help you do so, putting the @Override annotation signals to the compiler to check that the method signature is the same, and if not, to prevent compilation from taking place.
- It is **not compulsory** to put this **@Override** annotation, but it is **highly recommended**.

Since private variables/methods are not inherited since they are accessible only within the same class, you are not able to override any private method in the parent class.

```
public class Employee {
    private String name;
    private int id;

    Employee(String name, int id){
        this.name = name;
        this.id = id;
    }
    // Getters not shown
    @Override
    public String toString() {
        return String.format("Employee %s ID %d",name,id);
    }
}

public class FullTimeEmployee extends Employee {
    private int monthlyWage;
    FullTimeEmployee(String name, int id, int monthlyWage){
         super(name, id); System.out.println("FullTimeEmployee 3-arg cons.");
         this.monthlyWage = monthlyWage;
        }
        FullTimeEmployee(){
            System.out.println("FullTimeEmployee No-arg cons.");
            this.monthlyWage = 0;
        }
        public int getMonthlyWage() {
            return monthlyWage;
        }
        @Override public String toString() {
             String fromSuperclass = super.toString();
             return fromSuperclass + String.format("Monthly Wage %d", monthlyWage) ;
            }
}
```

Example 21 - Constructor Chaining

Recall that, no-arg super() is called in a subclass constructor if it is not specified. This means it is a good practice to explicitly write a no-arg constructor if your intention is for any class to be extended i.e. to be a superclass.
Suppose A is the superclass class of B, and B is the superclass of C.

Recall that the first call of every constructor must be to super(), consider the following example. By executing **C c = new C()**, what will you see on the screen?

A) Constructor C Constructor B Constructor A
B) Constructor A Constructor B Constructor C

```
public class A {
    A(){
        System.out.println("Constructor A");
    }
}
class B extends A{
    B(){
        System.out.println("Constructor B");
    }
}
class C extends B{
    C(){
        System.out.println("Constructor C");
    }
}
```

1. C's constructor is called. It implicitly calls B's constructor using super().
2. B's constructor is called. It implicitly calls A's constructor using super().
3. A's constructor is called, which doesn't call any other constructor since it's the topmost class in the hierarchy.
4. Once A's constructor completes, control returns to B's constructor.
5. Once B's constructor completes, control returns to C's constructor.

Output is B.

Example 22 - Overloading

Overloading means to write 2 or more methods with the same name, but take in different sets of parameters.
You see overloading in many places in the Java API e.g. String.valueOf() can take in more than one datatype. From the datatype you pass to this method, the JVM knows which version to call.

In the example below, **pIncrease()** is overloaded and you simply execute the method with the necessary parameters and the JVM will access and execute the correct version accordingly.

```
public class P {
    private int p;
    P(){
    }
    public void pIncrease(){
        p = p + 1;
    }
    public void pIncrease(int v){
         p = p + v;
    }
}

P p = new P();
p.pIncrease(); // valid
p.pIncrease(20); // valid
```

Example 23 - Declare object using superclass

You may declare an object using its superclass as a datatype. This is one form of **polymorphism**.
Suppose A is the superclass class of C.
Thus far you would instantiate an object of class C as follows.
C c = new C();
And you would instantiate an object of class A as follows.
A a = new A();
Since A is a superclass of C, we can also treat classes as datatypes and say that

- A is the supertype of C .
- C is a subtype of A.

Thus the following declaration is legal and will pass compile-time checks.
A a1 = new C();
In the declaration A a1 = new C();

- the declared type of variable a1 is class A,
- the actual type of variable a1 is C.
- variable a1 can access methods of class A, following the declared type

We see that variables declared as type A can be assigned to objects of class A or class C. This is known as **polymorphism**.
Suppose A is the superclass of B, and B is the superclass of C, and C is the superclass of D. toString() is implemented in class A and C. See the example.
In the following declaration, the toString() method is invoked.

Object a2 = new D();
a2.toString();

We have previously said that the methods available to variable a1 depend on the declared type. Since the Object class defines toString(), you can execute the toString() method on a2. What methods can you NOT execute?

Which toString() method will be executed?

The JVM will search for the implementation of toString() up the inheritance hierarchy, starting from the most specific class (i.e. class D) until it finds an implementation.
Thus, the JVM will start looking for toString() in class D, then in class C then in class B and so on.

```java
public class A {
    A(){}
    public void apple(){
        System.out.println("apple");
    }
    @Override
    public String toString() {
        return "Class A";
    }
}
class B extends A {
    public void orange(){
        System.out.println("orange");
    }
}
class C extends B{
    @Override
    public String toString() {
        return "Class C";
    }
}
class D extends C{
}

A a1 = new C(); // can be executed
a1.apple(); // can be executed
//a1.orange(); // cannot be executed
a1.toString(); // can be executed
Object a2 = new D(); // can be executed
System.out.println(a2.toString()); // Class C will be printed
```

Example 24 - Visibility Modifiers / Accessibility Modifiers

There are four visibility modifiers. Recall that private restricts access to within the same class.
When you override a method in a subclass, you cannot decrease its visibility. Thus, a method declared public must remain public in a subclass.

Access is allowed…				
Modifier	From the same case	From any class in the same package	From a subclass in a different package	From any class in a different package
`public`	yes	yes	yes	yes
`protected`	yes	yes	yes	
(no keyword) package-private	yes	yes		
`private`	yes			

Example 30 - Recall Instantiation

You create an instance of a class using the new keyword. This is called instantiation, and creates an object in memory, which has attributes (also called instance variables) and methods. How you instantiate depends on the constructor(s) defined in the class. Once you have an object after instantiation, you can execute any methods that are defined in the class.
The keyword static allows a method to be executed without creating an instance of the class.

```
public class Orange {
    public static String getFact(){
        return "Vitamin C";
    }
    private String type;
    Orange(){ this.type = "Navel Orange"; }
    Orange(String type){ this.type = type; }
    public String getType(){
        return type;
    }
}

public class TestOrange {
    public static void main(String[] args) {
        // TODO 1. Write code to get the string "Vitamin C".
        Orange.getType();
        // TODO 2. Write code to instantiate an Orange object,
        // and execute getType() to get the string "Navel Orange"
        Orange A = new Orange();
        System.out.println(A.getType());
        // TODO 3. Write code to instantiate an Orange object,
        // and execute getType() to get the string "Tangerine"
        Orange B = new Orange("Tangerine");
        System.out.println(B.getType());
    }
}
```

Example 31 - Inheritance and Polymorphism

Inheritance: A child class can inherit non-private methods and instance variables from a parent class.
Overriding: A child class can provide a new definition of a method defined in the parent class. This is as long as the method signatures remain the same. The parent class method (of the same signature) can be invoked using the super keyword.
Subtype and Supertype: An inheritance relationship results in the two classes having a subtype and supertype relationship. In the declaration below, A is a supertype of B. B is a subtype of A. Thus a variable a declared as type A can refer to an object B, since B is a subclass of A.

```
A a = new B();
```

Dynamic binding: In the declaration above, all methods available to variable a are based on what class A has, ("declared type"). However, remember that due to overriding, the same method can be defined in the subclasses.
Let's suppose both A and B have a method in their class definitions called something().
With the declaration above, if a.something() is executed, the JVM will begin looking for a something() method in the lowest level of the inheritance hierarchy i.e. in class B, before going up.

```
public class TestA {
    public static void main(String[] args) {
        A a = new B(); //what method can you execute? what will you see?
        // Can execute something()
        a.something(); // "Bond In B" printed instead of "Anya in A"
        }
    }
class A{
    public void something(){
        System.out.println("Anya in A");
    }
}
class B extends A{
    @Override
    public void something(){
        System.out.println("Bond In B");
    }
    public void nothing(){
        System.out.println("Nothing here");
    }
}
```

Example 32 - Overriding toString(), equals(), hashCode()

It is good practice for your class to override toString(), equals() and hashCode(). In Java, Object is the superclass of all classes. It has the following methods (and others) defined, which your class needs to override to provide the necessary functionality.

Method	You should override this …
String toString()	to provide an informative string representation of your class
boolean equals(Object obj	If you need to define what makes two objects of your classes identical in contents Note that == checks whether two objects are aliased. See next section on String
int hashCode()	If you override equals(), override this to return a hashCode value, such that identical objects should return the same hashCode value

```java
import java.util.Objects;
public class Coordinate {
    private int x;
    private int y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    //TODO 1 override toString() to return an informative string
    @Override
    public String toString() {
        return "Coordinate: " + x + ", " + y;
    }
    //TODO 2 the implementation using Android studio's wizard
    // (or your own IDE's)
    // will be sufficient for equals() and hashCode()
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Coordinate)) return false;
        Coordinate that = (Coordinate) o;
        return getX() == that.getX() && getY() == that.getY();
    }
    @Override
    public int hashCode() {
        return Objects.hash(getX(), getY());
    }
}
```

Example 33 - equals() vs ==

String objects with identical contents are stored as a single instance in the same pool. Use equals() to compare their values, do not use == . The JVM maintains a pool of String instances and strings declared using literals are interned in this pool. What interning means is that there is only one instance of a string value in the pool, as long as it is declared using a literal. This helps to reduce memory usage.

Thus in the example below,

- s1 and s2 point to the same string in the String pool.
- s3 is declared with the new keyword, it does not go in the pool.

Hence, even though the == operator checks for aliasing, it seems to work for comparing string values. However, this is a logic error if the aim of the code is to check if values are identical.

- Always use the equals() method to compare two String values (and Integer, Double, BigDecimal etc)
- Avoid using the new keyword to instantiate a String, as it does not go into the string pool.

```
public static void main(String[] args) {
    String s1 = "hello";
    String s2 = "hello";
    String s3 = new String("hello"); //avoid doing this
    /** Don't do this */
    System.out.println(s1 == s2); // true
    System.out.println(s1 == s3); // false, since s3 is not interned
    System.out.println(s1 == "hello"); // true
    /** Always use the equals method */
    System.out.println(s1.equals(s3)); // true
    System.out.println(s1.equals("hello")); // true
}
```

Summary: == or .equals() ?

1. For primitive types, use the comparison operator to compare values.

```
int a = 10;
int b = 20;
System.out.println(a == b);
```

2. To check if objects are aliased, use the comparison operator.

```
BigDecimal a = new BigDecimal("12");
BigDecimal b = a;
System.out.println(a == b);
```

3. To check if objects have equal contents or values, use the .equals() method. This includes String objects.

```
BigDecimal a = new BigDecimal("12");
BigDecimal b = new BigDecimal("12");
System.out.println(a.equals(b));

String a = "apple";
System.out.println( a.equals("orange"));
```

Example 34 - Abstract Classes

You have learnt about inheritance. In the design of your class hierarchy, you may envision a situation where

- some methods have the same implementation for all classes
- other methods have to be implemented in the subclasses

Thus, it would not make sense to allow the top-most parent to be instantiated. In such a design situation, abstract classes can be used. They are defined like any other concrete class, but:

- the abstract keyword is used
- there may be zero or more abstract methods defined

Abstract methods merely have the signature defined, but no body exists, and the abstract keyword is used. It will then be the responsibility of the subclasses to provide an implementation. Given the following abstract class, write a concrete subclass Circle that implements the abstract methods.

```java
public abstract class GeometricObjectMod {
    private String colour;
    private boolean filled;
    GeometricObjectMod(){
        colour = "white";
        filled = false;
    }
    GeometricObjectMod(String colour, boolean filled){
        this.colour = colour;
        this.filled = filled;
    }
    // abstract methods, only the signature
    public abstract double getArea();
    public abstract double getPerimeter();

    public String getColour() {
        return colour;
    }
    public boolean isFilled() {
        return filled;
    }
}
public class Circle extends GeometricObjectMod  {
    private double radius;
    Circle(String colour, boolean filled, double radius){
        super(colour , filled);
        this.radius = radius;
    }
    public double getRadius(){
        return radius;
    }
    // implementing the abstract methods in the GeomertricObjectMod, must ensure that signature is the same
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
    @Override
    public double getPerimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

Example 35 - Interface

One reason for having interfaces is to provide a common behaviour for a group of classes, which **makes code flexible**.
If a concrete class A implements an interface B, this provides a guarantee that the abstract methods defined in B will be implemented in A, otherwise, this will generate a compile-time error.
Also, an interface acts as a supertype to the concrete classes that implement it.

Using code in the previous example, you can declare:
Dog doggy = new WelshCorgi();
Since the methods available to doggy depends on the declared type (Dog), you have the guarantee that you can execute doggy.makeSound() and doggy.wagTail().
Furthermore, if you wish to change the implementation of doggy, simply change the declaration to:
Dog doggy = new SiberianHusky();
You STILL have the guarantee that you have the same behaviour – you still can execute doggy.makeSound() and doggy.wagTail().

OOP Design Principle

- Single Responsibility Principle – Each class should have only one reason to change
    - think very carefully about what it is meant to do and avoid specifying unrelated functionality in it
- Program to interfaces, not implementations
    - This means that you should declare a variable as an interface type. Then, you are free to use ANY concrete implementation of that interface, and you can CHANGE the concrete implementation without affecting the rest of the code.

```java
public interface Dog {
    // interface --> public and abstract are implied
    void makeSound();
    void wagTail();
}
public class SiberianHusky implements Dog{
    @Override
    public void makeSound() {
        System.out.println("woo");
    }
    @Override
    public void wagTail() {
        System.out.println("waggers");
    }      // implementing Dog interface --> must provide Dog abstract method
}
public class WelshCorgi implements Dog {
    private String name;
    @Override
    public void makeSound() {   // abstract method from Dog
        System.out.println("woof");
    }
    @Override
    public void wagTail() { // abstract method from Dog
        System.out.println("wags tail");
    }
}
```

Example 36 - Comparable

Java provides the Comparable interface (part of java.util library) for objects to be sorted in a natural ordering. It has one abstract method int compareTo(T o).
Recall your previous lesson on Generics: T is a type parameter, which enables different types to be specified in place of T.
public interface Comparable<T>{ int compareTo(T o); }
The method compareTo should "Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object".
An ArrayList of objects can be sorted by **Collections.sort()** as long as they have the **Comparable<T> interface implemented**.

```java
public class Coordinate implements Comparable<Coordinate> {
    private double x, y;
    Coordinate(){}
    Coordinate(double x, double y){
        this.x = x;
        this.y = y;
    }
    public double getDistance(){
        return Math.sqrt( x*x + y*y);
    }
    public double getX() {
        return x;
    }
    @Override
    public String toString() {
        String template = "(%f, %f)";
        return String.format( template, x, y);
    }
    @Override   // for sorting using Collections.sort() later
    public int compareTo(Coordinate other){
        if (x > other.x) {
            return 1;
        }else if ( Math.abs(x - other.x) < 1e-10){      //(x == other.x) no work -> floating point error
            return 0;
        }else {
            return -1;
        }
    }
}
public class CoordinateComparator implements Comparator<Coordinate> {
    // Comparator used to compare stuff from outside the class
    // sort by distance in ascending order
    // no constructor -> no-arg constructor present by default
    @Override
    public int compare(Coordinate coordinate, Coordinate t1) {
        if (coordinate.getDistance() < t1.getDistance()){
            return 1;
        }else if (Math.abs(coordinate.getDistance() - t1.getDistance()) < 1e-10){
            return 0;
        }else{
            return -1;
        }
    }
}
public class TestCoordinate {
    public static void main(String[] args) {
        System.out.println("before sorting");
        System.out.println(coordinateList); //[(5.00, 12.00), (3.00, 4.00), (7.00, 24.00), (0.00, 0.00)]
        Collections.sort(coordinateList, new CoordinateComparator()); // this by dist
        System.out.println("after sorting");
        System.out.println(coordinateList);//[(7.00, 24.00), (5.00, 12.00), (3.00, 4.00), (0.00, 0.00)]
        Collections.sort(coordinateList); //[(0.00, 0.00), (3.00, 4.00), (5.00, 12.00), (7.00, 24.00)]

    }
}
```

Example 37 - List interface

Another example of this is the List<T> Interface, of which you are familiar with ArrayList<T>. But there are other implementations, such as LinkedList<T> and Vector<T> , which you can choose depending on the performance requirements.

```java
public class TestList {
    public static String ARRAYLIST = "ArrayList";
    public static String LINKEDLIST = "LinkedList";
    public static String VECTOR = "Vector";
    public static void main(String[] args) {
        //String choice = "ArrayList";
        String choice = "Vector";
        List<Integer> integerList = null;
        // TODO Write if/else statements to instantiate integerList based on choice
        if (choice.equals("ArrayList")){
            integerList = new ArrayList<>();
        }else if(choice.equals("LinkedList")){
            integerList = new LinkedList<>();
        }else if (choice.equals("Vector")){
            integerList = new Vector<>();
        }
        // ArrayList, LinkedList, Vector are concrete classes that implement the List interface
        // List has abstract methods add() get() which concrete classes implement
        integerList.add(10);
        integerList.add(20);
        integerList.add(30);
        System.out.println(integerList.get(1));
        System.out.println(integerList);
        System.out.println("List:" + (integerList instanceof List) );
        System.out.println("ArrayList:" + (integerList instanceof ArrayList) );
        System.out.println("Vector:" + (integerList instanceof Vector) );
        /* List<> Interface has an add method, so regardless
         * of the choice of object, I have the guarantee that I can
         * call it */
        System.out.println("List is empty: " + integerList);
        integerList.add( Integer.valueOf("123"));
        integerList.add( Integer.valueOf("456"));
        System.out.println(integerList);
    }
}
```

Example 38 - Stack

You will recall that a stack is a Last-In-First-Out linear data structure, with the following operations
· push – add an item to the top of the stack
· pop – remove the item on the top of the stack and return it
· peek – return the item on the top of the stack
· size – number of elements
· isEmpty – returns true if no elements and false otherwise

```
public interface CustomStack<T> {        // T is type parameter
    void push(T t);

    T pop();

    int size();

    T peek();

    boolean isEmpty();

    // All are abstract methods
}

import java.util.*;
import java.math.*;

public class MyStack<T> implements CustomStack<T>{
    private List<T> stackContents;

    public MyStack(){
        stackContents = new ArrayList<>();
    }
    @Override
    public void push(T t){
        stackContents.add(t);
    }
    @Override
    public T pop(){
        if (isEmpty()){
            return null;
        }
        return (stackContents.remove(stackContents.size()-1));
    }
    @Override
    public int size(){
        return stackContents.size();
    }
    @Override
    public T peek(){
        if (isEmpty()){
            return null;
        }
        return (stackContents.get(stackContents.size() - 1));
    }
    @Override
    public boolean isEmpty(){
        return (stackContents.isEmpty());
    }
}
```

To change the way the stack elements are stored, leave the code for MyStack alone, and write a new class that also implements CustomStack.

.

Example 41 - Try-catch

try-catch is for situations beyond programmers control, otherwise use if else if you can
if (s == null){do smth}
else {do smth else}
Put code to where exceptions are likely to occur in a try-catch block

In the following example, a divide-by-zero could occur at the line return x/y, and if y is 0, the JVM throws an ArithmeticException object. You can write a try-catch block to handle this error.
In the try-block, once an Exception is thrown, execution is diverted to the catch block. The rest of the lines in the try-block are not executed.
ArithmeticException is known as an Unchecked Exception. We will have more to say on this later.

Notice that you have the option to NOT use a try-catch block to handle the error. This is left to you as an exercise.

```
public class TestQuotient {
    public static void main(String[] args) {
        try{
            System.out.println("A");
            System.out.println(divide1(1, 0));
            // divide throws an ArithmeticException -> transfer to catch block
            System.out.println("B");
        }catch( ArithmeticException ex){
            System.out.println("In catch block");
            ex.printStackTrace();
        }
    }

    /** TODO 1 Handle this method in main() using try/catch block */
    public static int divide1( int x, int y) {
        return x/y;
    }

    /** TODO 2 Handle this using if/else */
    public static int divide2( int x, int y){
        if( y == 0){
            System.out.println("cannot divide by zero");
            return Integer.MAX_VALUE;
        }
        return x/y;
    }
}
```

Example 42 - Opening & Reading from Text File (Question 9 2B)

File() object is known to throw a FileNotFoundException. You have two options.
The programmer cannot control whether the file is on the system or not. Thus Exceptions and a try-catch block are an ideal way to handle the scenario.

You are writing a method openFile() to open, read and return the contents of a file in a single string. The File() constructor throws a FileNotFoundException. You have two design choices that you can make:

- Handle the exception in openFile() itself using a try-catch block
- Delegate the task of handling the Exception to code that calls openFile(). You then need to declare that openFile() throws FileNotFoundException object. This is shown below.\

```java
public class TestOpenReadFile {
    public static void main(String[] args) {
        try{
            System.out.println("A");
            openFile("Anya");
            System.out.println("B");
        }catch( FileNotFoundException ex){
            System.out.println("File is not found!");
            ex.printStackTrace();
        }

    }

    public static String openFile(String fileName) throws FileNotFoundException {

        // TODO1 Uncomment and modify this method so that exception is handled by this methods caller

        File file = new File(fileName);
        Scanner scanner = new Scanner(file); // throw an FileNotFoundException
        String out = "";
        while( scanner.hasNext()){
            String s = scanner.nextLine();
            out = out + s;
        }
        return out;
    }


    /* TODO2 Write a version of openFile that handles the exception within it  */


}
```

Example 43 - Catch more specific exception first

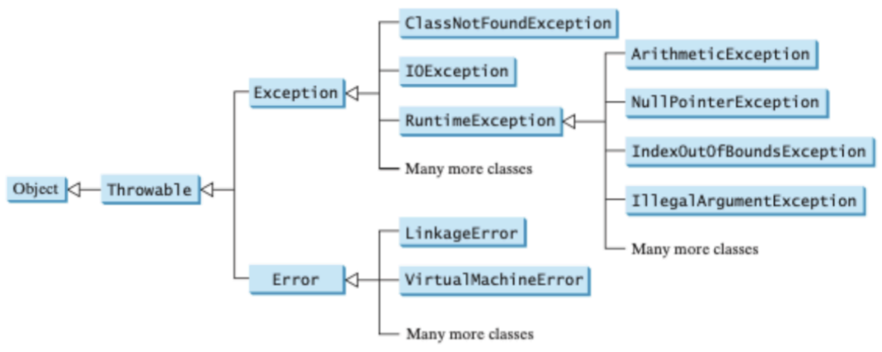
Unchecked Exceptions:
RuntimeException and all its subclasses are known as Unchecked Exceptions.
The compiler does not enforce that you declare in method signature and put code in try-catch block. These are usually thrown due to logic errors, is detected during testing, and can handled by modifying your code.
Thus, for logic errors, you should usually handle them **using if/else** and not use try-catch blocks.
Checked Exceptions:
All other Exceptions are known as Unchecked Exceptions. Compiler enforces the “handle or declare rule”: (You saw this in Example 42)
· If your method executes code that throws a checked exception, you must handle that exception in your method or throw the exception (see next rule)
· **If your method throws a checked exception, you must declare in method signature**

You can instantiate a BigDecimal object using a string, and if the string does not represent a valid number, then a NumberFormatException will be thrown eg.
BigDecimal b = new BigDecimal ("12g4");
A divide() method throws a IllegalArgumentException if the RoundingMode is an invalid value.
You can write more than one catch block to handle each of these exceptions, but you must be aware of the inheritance hierarchy.

```java
import java.math.BigDecimal;
public class TestBigDecimal {

    public static void main(String[] args) {
        // TODO Write a try/catch block to handle the various exceptions
        try{
            System.out.println( divide("12a", "60"));
        }catch( NumberFormatException ex){ // more specific
            System.out.println("NumberFormatException caught");
            ex.printStackTrace();
        }catch( IllegalArgumentException ex){ // more general
            System.out.println("IllegalArgumentException caught");
            ex.printStackTrace();
        }catch( RuntimeException ex){ // parent of IllegalArgumentExec
            // catches any other exception that you didn't anticipate
            System.out.println("RuntimeEx");
        }
    }
    // unchecked exceptions --> Java doesn't force you to declare and handle
    public static String divide (String s1, String s2) {
        // THIS METHOD COULD THROW ILLEGALARGUMENTEXCEPTION and NUMBERFORMATEXCEPTION

        /** The constructors throw a NumberFormatException if the string is not a valid number */
        // eg s2 = "12abc" <-
        BigDecimal b1 = new BigDecimal(s1);
        BigDecimal b2 = new BigDecimal(s2); // <-- throw a NumberFormatException

        /** using the deprecated overloaded version of divide()
         * public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode);
         * It throws NumberFormatException if the Rounding Mode is not a valid number
         */
        BigDecimal b3 = b1.divide(b2,3, 20);
        return b3.toString();
    }
}
```

Example of declaring for checked exceptions:

```java
public static void getURIfromURL(String myURL) throws URISyntaxException {
URI uri = new URI(myURL); //may throw a URISyntaxException
System.out.println("E");
}
```



→ RuntimeException = Unchecked

Example 44 - Methods to throw Exceptions

Coming back to the quotient example in Example 41, you could also decide to throw an exception if y = 0 is passed to the method.
Notice that the keyword used here is throw, together with new and the constructor of an Exception object. You can also pass an message to the constructor.
The throws keyword is **used at the method signature** only.

```
public class TestThrowsException {

    public static void main(String[] args) {

        // Write a try/catch block to handle the exception thrown
        try{
            System.out.println( divide1(10, 0));
        }catch(IllegalArgumentException ex){
            System.out.println(ex.getMessage());
        }
    }
    // unchecked exception - no need to declare and handle
    public static int divide1( int x, int y) throws IllegalArgumentException {

        if (y == 0){
            throw new IllegalArgumentException("Cannot divide by zeroooo"); // check note*
        }
        // TODO throw an Illegal Argument Exception if the denominator is zero
        return x/y;
    }
}
```

***Note that:**

```
if( y == 0) throw new IllegalArgumentException("Message")
```

Also means:

```
try{
        // code
}catch( Exception ex){
        ex.printStackTrace(); // get all the error messages
        String s = ex.getMessage(); // get just "Message"
}
```

Example 45 – Exceptions: Reading data from a URL

If you are accessing a webpage to get data, many things could go wrong: 1. There could be no internet connection. 2. There could be a typo in the URL. In the example below, a webpage can be accessed by Instantiating a URL object with the address, Calling the url.openStream() method to get an InputStream object which provides a stream of data. This stream data is passed to a Scanner object to read this line-by-line
The URL constructor and the openStream() method both throw exceptions. Adjust the method signature accordingly.

```java
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Scanner;
public class TestDownloadURL {
    public static void main(String[] args) {
        // TODO Write a try/catch block
        try{
            System.out.println( queryURL("https://www.google.com.sg"));
        }catch( MalformedURLException ex){
            ex.printStackTrace();
        }catch( IOException ex){
            ex.printStackTrace();
        }
    }
    /** TODO Remove the throws Exception and replace it with the specific exception */
    // checked exception ==> you must declare
    public static String queryURL(String urlString) throws IOException {
        // Create a URL object
        URL url = new URL(urlString);
        // read the URL stream into the Scanner object
        Scanner input = new Scanner(url.openStream());
        String out = "";
        while (input.hasNext()){
            out = out + input.nextLine() + "\n";
        }
        return out;
    }
}
```

Exceptions Summary

Exceptions can be thrown, declared, and handled. This is what happens.

1. method2() has code that results in an Exception object being thrown. It does not handle the exception object thrown.
2. method2() thus declares in the method signature. The compiler enforces this for Checked Exceptions.
3. method1(), who calls method2(), is now responsible for handling the exception thrown by method2() through a try-catch block.

```java
class A{
    void method1(){
        try{
            method2();
        }catch(SomeException ex){
            // handle exception
        }
    }
    void method2() throws SomeException{
        if (error) throw new SomeException();
    }
}
```

Example 50 - Interfaces have only abstract methods, act as a supertype, and make guarantees on classes that implement them

If code could talk:

Animal: “I promise you that any concrete class that implements me must provide code for the method **makeSound()**. By default the method is **public**.”

```
public interface Animal {
    String makeSound();
}
```

Cat: "Since I implement **Animal**, I must keep the promise and implement **makeSound()** exactly according to the signature given. If I do not, I will become an abstract class."

```
public class Cat implements Animal {
    @Override
    public String makeSound() {
        return "meow";
    }
}
```

Animal a: "I promise you that variable **a** can hold a reference to any object that implements **Animal** interface, and thus I promise you that **a** is able to execute the **makeSound()** method."

```
public static void main(String[] args) {
    Animal a = new Cat();
    a.makeSound();
}
```

Example 50a - OOP Design Principle Single Responsibility Principle

Each class should have only one reason to change. Good design would include flexibility to change and safe from bugs

Imagine that you are writing a software to print bus tickets based on the distance travelled on the bus route. You realize that the following tasks could change

- The way bus fares are calculated
- The information printed on the bus ticket

Based on what you have learnt so far, you would use inheritance and overriding to change the implementation of each of the methods when the need arises. You will soon get many classes with various combinations of the implementation of the two tasks, violating the Single Responsibility Principle.

```java
public class BusTicket {
    private String busService;
    private final int SHORTFARE = 100;
    private final int LONGFARE = 120;

    BusTicket(String busService) {
        this.busService = busService;
    }

    public int getBusFare(int distance) {
        return distance < 32 ? SHORTFARE : LONGFARE;
    }

    public String getTicketPrintout(int distance) {
        return "Service:" + busService + " Fare:" + getBusFare(distance);
    }
}
```

Example 51 - Tight Coupling (Depend on another concrete class)

Depending directly on another concrete class is tight coupling and makes your code less flexible. To follow the single responsibility principle, the solution is to give the calculation of bus fares to another object. This is called delegation. Consider the following implementation. However, notice that the BusTicket class depends directly on BusFareConcrete class.
What if you are required to update bus fares? You can:

1. Modify getBusFare() in BusFareConcrete directly
2. Sub-class BusFareConcrete and override getBusFare()

```java
public class BusTicket {
    private String busService;
    private BusFareConcrete busFare;

    BusTicket(String busService, BusFareConcrete busFare){
        this.busService = busService;
        this.busFare = busFare;
    }

    public String getTicketPrintout(int distance){
        return "Service:" + busService + " Fare:" + busFare.getBusFare(distance);
    }
}
public class BusFareConcrete {
    private final int SHORTFARE = 100;
    private final int LONGFARE = 120;

    public int getBusFare(int distance){
        if( distance < 32){
            return SHORTFARE;
        } else {
            return LONGFARE;
        }
    }
}
```

Example 52 – Loosely coupled (Use interfaces/abstract class)

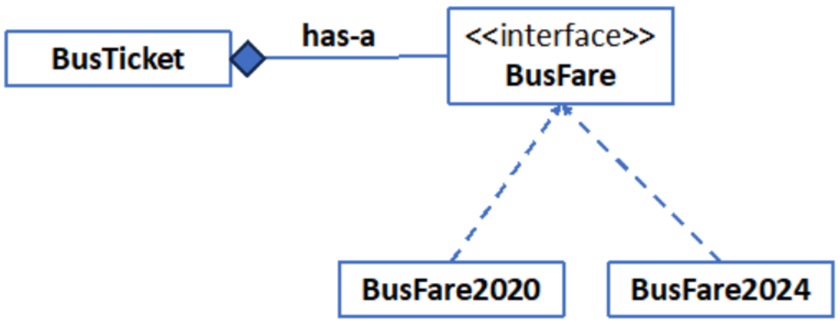
Modifying existing code directly is not encouraged in a production situation as it would have been well-tested and integrated in the software, thus you would not want to risk causing the software to fail and annoy your stakeholders.
Sub-classing and writing a new method would be better, but it would be hard to justify the inheritance relationship. Also, inheritance does not FORCE you to override.
To loosely couple to another class, make your instance variables depend on an Interface instead of a concrete class. This gives you the flexibility to assign ANY concrete class to it, as long as it implements the Interface.
Steps to make the code more flexible.

1. Write an interface BusFare with the necessary abstract method(s) for the calculation of bus fares.
2. Write a concrete class BusFare2020 that implements the BusFare interface.
3. Declare an attribute in BusTicket of type BusFare, and adjust the constructor accordingly. This means that BusTicket can work with ANY object that implements BusFare Interface.
4. When bus fares change, write a new concrete class BusFare2024 that implements the BusFare interface. Then simply change the object passed to the BusTicket constructor when bus fares change, and the information printed on the bus ticket is correct.

```
public interface BusFare {
    int getBusFare(int distance);
}
public class BusFare2020 implements BusFare{
    private final int SHORTFARE = 100;
    private final int LONGFARE = 120;
    @Override
    public int getBusFare(int distance) {
        return distance < 32 ? SHORTFARE : LONGFARE;
    }
}
public class BusFare2024 implements BusFare {
    public final int SHORTFARE = 200;
    public final int MEDIUMFARE = 250;
    public final int LONGFARE = 300;
    @Override
    public int getBusFare(int distance) {
        if( distance < 10){
            return SHORTFARE;
        }else if( distance < 25){
            return MEDIUMFARE;
        }else{
            return LONGFARE;
        }
    }
}
public class BusTicket {
    private String busService;
    private BusFare busFare;
    // ANY object that implements the BusFare interface can be passed
    // to the class BusTicket (compare with previous Example)
    BusTicket(String busService, BusFare busFare){
        this.busService = busService;
        this.busFare = busFare;
    }

    public String getTicketPrintout(int distance){
        return "Service:"+busService + "Fare:" + busFare.getBusFare(distance);
    }
}
public class TestBusTicket {
    public static void main(String[] args) {
        // TODO Write code for 2020 bus ticket
        BusTicket busTicket = new BusTicket("20", new BusFare2020());
        System.out.println("2020"+busTicket.getTicketPrintout(15));
        // TODO Write code for 2024 bus ticket
        BusTicket busTicket1 = new BusTicket("20", new BusFare2024());
        System.out.println("2024"+busTicket1.getTicketPrintout(15));
    }
}
```

Favour Composition over Inheritance (Open/Closed Principle)

Example 52 illustrates the use of composition, a has-a relationship between the classes, to make software more flexible. If you want to change the bus fare calculation, simply write a new class that implements the BusFare interface. (Loose coupling)
In this design, BusTicket only needs the guarantee of methods from the BusFare interface to function correctly, and does not need to depend on any specific concrete class. This is the "Program to an interface, not an implementation" principle. If the calculation of bus fares change, all the software designer needs to do is to write a new class that implements the BusFare interface. Existing concrete classes need not be modified.



This design illustrates the following OOP Design Principle: Open/Closed Principle - classes should be open for extension and closed to modification.

- "Open for extension": change the bus fare calculation? simply write a new class that implements the BusFare interface.
- "Closed to modification": don't modify existing code.

Example 53 - Inheritance can lead to duplicated code

This example illustrates why inheritance is not the best solution to deal with changing requirements. Notice that as some birds can/cannot fly and can/cannot swim, each method is overridden accordingly, and this leads to many classes with duplicated code.
Furthermore, the coder has to deal with changing requirements – in this case, what if a bird can fly but can't do it well (like a chicken) or can fly really long distances (like a migratory bird)?

```java
public class Bird {
    public void fly() {
        System.out.println("I CAN fly");
    }
    public void swim() {
        System.out.println("I can't swim");
    }
}
class DomesticDuck extends Bird {
    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
    @Override
    public void swim() {
        System.out.println("I CAN swim");
    }
}
class CanadaGoose extends Bird {
    @Override
    public void swim() {
        System.out.println("I CAN swim");
    }
}
class Pigeon extends Bird {
    @Override
    public void fly() {
        System.out.println("I CAN fly long distances");
    }
}
class Ostrich extends Bird {
    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
    @Override
    public void swim() {
        System.out.println("I can't swim");
    }
}
```

Example 54 - Strategy Design Pattern – Make classes flexible?

A design pattern is an existing solution to a known problem in OOP. You have seen how composition is used in Example 52. The Strategy Design Pattern uses the same idea. The idea in the Strategy Design Pattern is to “encapsulate what varies” i.e. to make the flying behaviour into separate classes so that you can use composition to specify the behaviour that you want.

1. Write an interface that specify what method FlyBehaviour objects will have

```
public interface FlyBehaviour {
    void doFly();
}
```

2. Write as many concrete classes that implement FlyBehaviour as required. (Single Responsibility Principle)

```
public class CanFly implements FlyBehaviour {
    @Override
    public void doFly() {
        System.out.println("I CAN fly");
    }
}
```

3. Rewrite the Bird class to have (i) instance variable to store FlyBehaviour objects (ii) a setter to assign such objects to the instance variable (this can also be specified at the constructor)

4. Rewrite the fly() method to execute the method that the FlyBehaviour interface guarantees. (Program to an interface)

```
public class Bird {
    private FlyBehaviour flyBehaviour;
    public void setFlyBehaviour(FlyBehaviour flyBehaviour){
        this.flyBehaviour = flyBehaviour;
    }
    public void fly(){
        flyBehaviour.doFly();
    }
}
```

Benefit of Strategy Design Pattern:

- Since you encapsulate the various behaviours, the client can change them during runtime if needed
- If there are new requirements or specifications for the behaviour, a new class can simply be written and passed to the setter (Open/Closed principle)

```
interface FlyBehaviour {
    void doFly();
}

public class Bird {
    // "I guarantee that flyBehaviour will have a doFly() method,
    // since it is of the FlyBehaviour interface
    private FlyBehaviour flyBehaviour;
    public void setFlyBehaviour(FlyBehaviour flyBehaviour) {
        this.flyBehaviour = flyBehaviour;
    }

    // Since FlyBehaviour objects are guaranteed to have a doFly() method,
    // you can execute doFly() on flyBehaviour
    public void fly() {
        flyBehaviour.doFly();
    }

    public static void main(String[] args) {

        Bird bird = new Bird();
        // TODO add a CanFly object and execute the fly Method
        bird.setFlyBehaviour(new CanFly());
        bird.fly();

        //change behaviour in runtime
        // TODO now add a LongDistanceFly and execute the fly method
        bird.setFlyBehaviour(new LongDistanceFly());
        bird.fly();

        // Design Pattern - known solution to OOP problem
        // Strategy Design Pattern - how do you change behaviours at runtime?
        // for algorithms, calculations as well
    }
}
class CanFly implements FlyBehaviour {

    @Override
    public void doFly() {
        System.out.println("I CAN fly!");
    }
}

class ChickenFly implements FlyBehaviour {

    @Override
    public void doFly() {
        System.out.println("I CAN NOT fly really well!");
    }
}

class LongDistanceFly implements FlyBehaviour {

    @Override
    public void doFly() {
        System.out.println("I CAN fly long distances!");
    }
}
```

Lecture Recall Quizzes

~~Week 3 Lecture 1~~
~~Week 3 Lecture 2~~
Week 4 Lecture 1

Which of the following statements will correctly check that the primitive types have the same value?
int a = 16;
int b = 5;
double c = 0.1 + 0.2;

Ans:
- boolean isIntEqual = a == b
- boolean isDoubleEqual = Maths.abs(c - 0.3) < 1e-10;

Which of the following statements checks that two objects are aliased?
BigDecimal a = new BigDecimal("2.45");
BigDecimal b = a;
BigDecimal c = new BigDecimal("3.42");

Ans:
boolean isAliased = a ==b

Which of the following statements checks that two objects have identical contents?
BigDecimal a = new BigDecimal("2.45");
BigDecimal b = a;
BigDecimal c = new BigDecimal("3.42");

Ans:
boolean isAliased = a.equals(c);

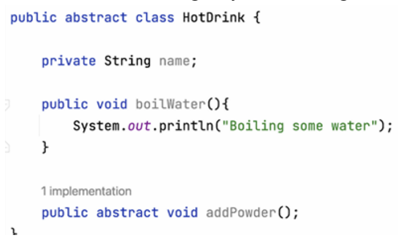
For the code below, which of the following statements is the correct way to check that the contents of String a is "Anya"?
String a = "Anya";

Ans:
a.equals("Anya")

Select all statements that represent correct design decisions.
When brewing hot drinks, the first step is to boil some water, and then to add the necessary powder. We all use a kettle to boil water! The kind of powder you add depends on the hot drink you are brewing e.g. for Matcha you add matcha powder, for "Milo", you add milo powder and sugar, "Milo Kosong" you just add milo powder and for "Milo Dinosaur" you add too much milo powder. For a HotDrink class, you are planning two methods, boilWater() and addPowder().

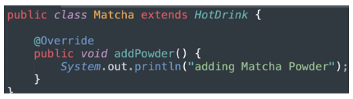
Ans:
- A HotDrink class cannot be instantiated, because it is not specific enough. Thus it should be declared an abstract class.
- boilWater() method should be implemented in HotDrink because water is boiled in the same way regard less of the type of hot drink.
- addPowder() method should be an abstract method in HotDrink because the type of powder(s) added depends on the subclass. Also, specifying this as an abstract method forces the subclass to provide an implementation.

Consider the following abstract class HotDrink.
Which of the following ways of defining a Matcha class that is a subclass of HotDrink will not cause a compilation error?

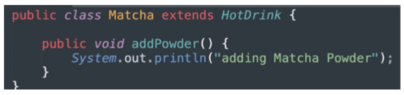


```java
public abstract class HotDrink {

    private String name;

    public void boilWater(){
        System.out.println("Boiling some water");
    }

    1 implementation
    public abstract void addPowder();
}
```

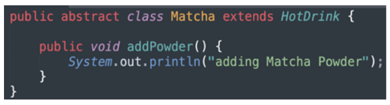
Ans:



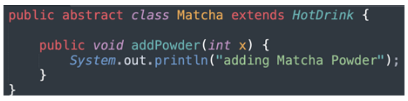
```java
public class Matcha extends HotDrink {

    @Override
    public void addPowder() {
        System.out.println("adding Matcha Powder");
    }
}
```



```java
public class Matcha extends HotDrink {

    public void addPowder() {
        System.out.println("adding Matcha Powder");
    }
}
```



```java
public abstract class Matcha extends HotDrink {

    public void addPowder() {
        System.out.println("adding Matcha Powder");
    }
}
```



```java
public abstract class Matcha extends HotDrink {

    public void addPowder(int x) {
        System.out.println("adding Matcha Powder");
    }
}
```

Week 4 Lecture 2

Which of the following is/are true about an interface?

```
public interface Edible {
        String howToEat();
}
```

Ans:
- A interface can only have abstract methods
- The abstract methods are understood to have public access
- To be a concrete class, a class which declares that it implements an interface (e.g. public class A implements Edible) must provide an implementation for the abstract method(s) in the interface.

Given the following interface, what is the correct way to use it so that a class can become a concrete class without any compilation error?

```
public interface Edible {
        String howToEat();
}
```

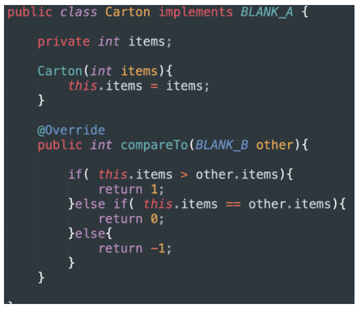
Ans:

```
public class Duck implements Edible{
        @Override
        public String howToEat(){
                return "Roast it"
        }
}
```

Which of the following declarations will compile without error?
Given an interface Edible and concrete classes Duck and Carrot that both implements Edible without compilation error, assume a no-arg constructor exists for all concrete classes.

Ans:
- Edible edible = new Duck()
- Edible edible = new Carrot()

What is the code in each blank?
Consider the following class which implements a particular interface to allow a List of Carton objects to be sorted in ascending order by Collections.sort().



```
public class Carton implements BLANK_A {

    private int items;

    Carton(int items){
        this.items = items;
    }

    @Override
    public int compareTo(BLANK_B other){

        if( this.items > other.items){
            return 1;
        }else if( this.items == other.items){
            return 0;
        }else{
            return -1;
        }
    }
```

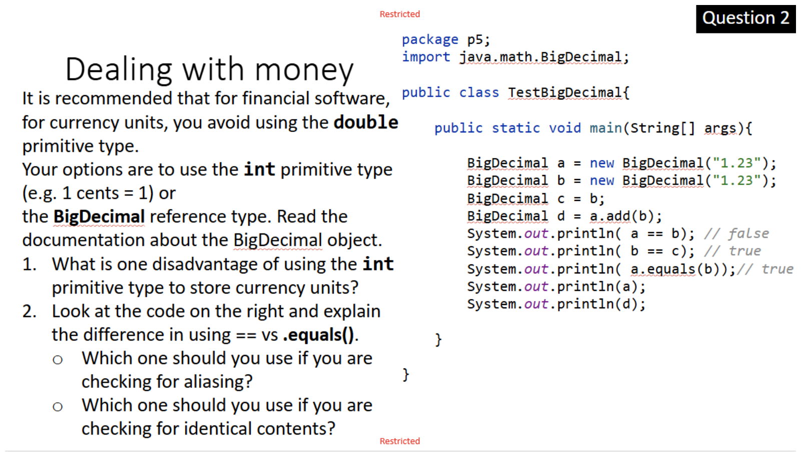
Ans:
- BLANK_A: Comparable
- BLANK_B: Carton

Week 5 Lecture 1

Week 5 Lecture 2

Norman Cohort Questions

Week 3



Restricted

Question 2

Dealing with money

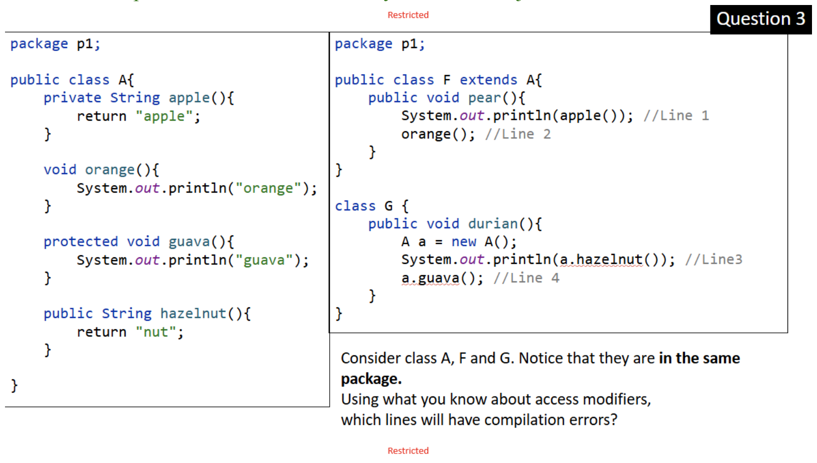
It is recommended that for financial software, for currency units, you avoid using the **`double`** primitive type.
Your options are to use the **`int`** primitive type (e.g. 1 cents = 1) or
the **BigDecimal** reference type. Read the documentation about the BigDecimal object.

1. What is one disadvantage of using the **`int`** primitive type to store currency units?
2. Look at the code on the right and explain the difference in using == vs **.equals()**.
   - Which one should you use if you are checking for aliasing?
   - Which one should you use if you are checking for identical contents?

```java
package p5;
import java.math.BigDecimal;

public class TestBigDecimal{

    public static void main(String[] args){

        BigDecimal a = new BigDecimal("1.23");
        BigDecimal b = new BigDecimal("1.23");
        BigDecimal c = b;
        BigDecimal d = a.add(b);
        System.out.println( a == b); // false
        System.out.println( b == c); // true
        System.out.println( a.equals(b));// true
        System.out.println(a);
        System.out.println(d);

    }
}
```

Restricted

1) It does not make for simple coding, need to convert all dollars to cents.
   a) Primitive double cannot be used because doubles are not stored exactly and floating point errors can accumulate. int cannot be used as its maximum value is 2^32 - 1, that's why there are classes like BigDecimal
2) The main difference is that equals() method compares the content equality of two strings (check if have same content in objecr_) while the == operator compares the reference or memory location of objects



Restricted

Question 3

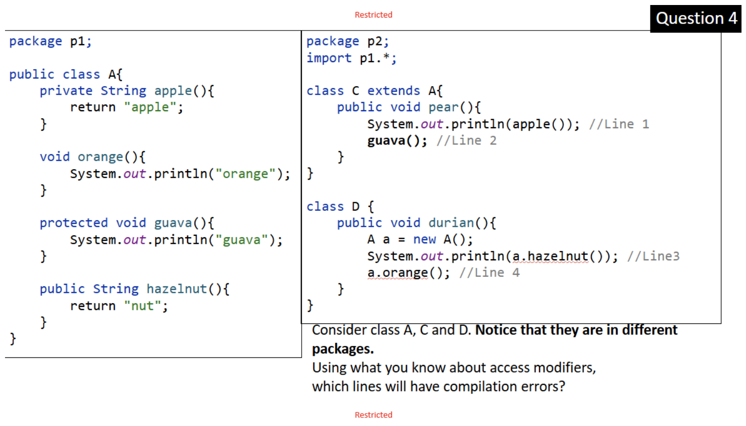
```java
package p1;

public class A{
    private String apple(){
        return "apple";
    }

    void orange(){
        System.out.println("orange");
    }

    protected void guava(){
        System.out.println("guava");
    }

    public String hazelnut(){
        return "nut";
    }

}
```

```java
package p1;

public class F extends A{
    public void pear(){
        System.out.println(apple()); //Line 1
        orange(); //Line 2
    }
}

class G {
    public void durian(){
        A a = new A();
        System.out.println(a.hazelnut()); //Line3
        a.guava(); //Line 4
    }
}
```

Consider class A, F and G. Notice that they are **in the same package.**
Using what you know about access modifiers, which lines will have compilation errors?

Restricted

Line 1 & 4 will have error.
Line 1: apple is private in class A, private is only accessible within same class
Line 2: orange has default (package-private) access, means that it is accessible within same package
Line 3: hazelnut is public, so accessible from anywhere
Line 4: guava is protected, protected = accessible within same package and subclass however, G is not a subclass of A.



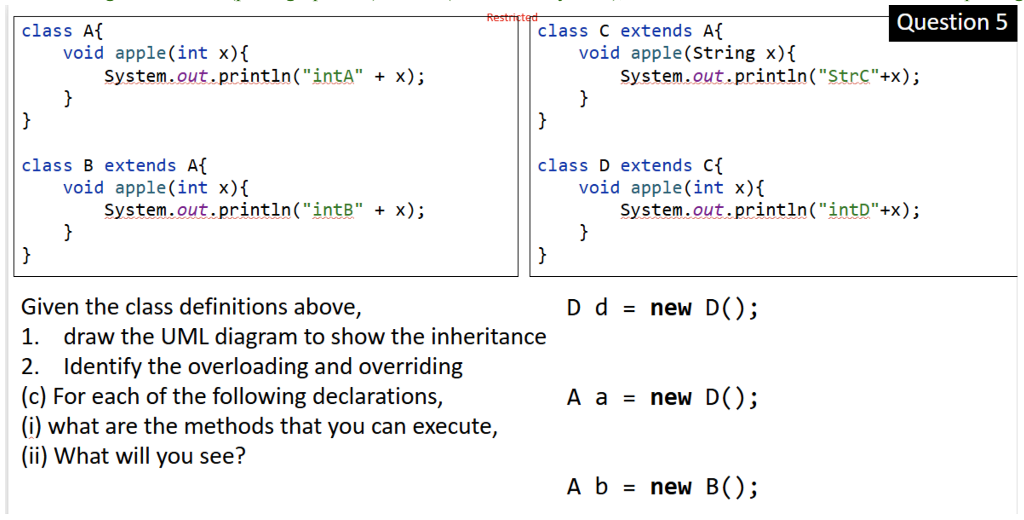
Question 4

```
package p1;

public class A{
    private String apple(){
        return "apple";
    }

    void orange(){
        System.out.println("orange");
    }

    protected void guava(){
        System.out.println("guava");
    }

    public String hazelnut(){
        return "nut";
    }
}
```

```
package p2;
import p1.*;

class C extends A{
    public void pear(){
        System.out.println(apple()); //Line 1
        guava(); //Line 2
    }
}

class D {
    public void durian(){
        A a = new A();
        System.out.println(a.hazelnut()); //Line3
        a.orange(); //Line 4
    }
}
```

Consider class A, C and D. **Notice that they are in different packages.**
Using what you know about access modifiers,
which lines will have compilation errors?

Line 1 and 4 will have error.
Line 1: apple is private in class A, private is only accessible within same class
Line 2: guava is protected, protected = accessible within same package and subclass, C is a subclass of A
Line 3: hazelnut is public, so accessible from anywhere
Line 4: orange has default (package-private) access (no access keyword), means that it is accessible within same package. D is in different package from A.



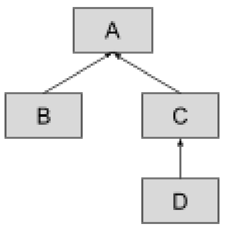
Question 5

```
class A{
    void apple(int x){
        System.out.println("intA" + x);
    }
}

class B extends A{
    void apple(int x){
        System.out.println("intB" + x);
    }
}
```

```
class C extends A{
    void apple(String x){
        System.out.println("StrC"+x);
    }
}

class D extends C{
    void apple(int x){
        System.out.println("intD"+x);
    }
}
```

Given the class definitions above,
1. draw the UML diagram to show the inheritance
2. Identify the overloading and overriding
(c) For each of the following declarations,
(i) what are the methods that you can execute,
(ii) What will you see?

```
D d = new D();

A a = new D();

A b = new B();
```

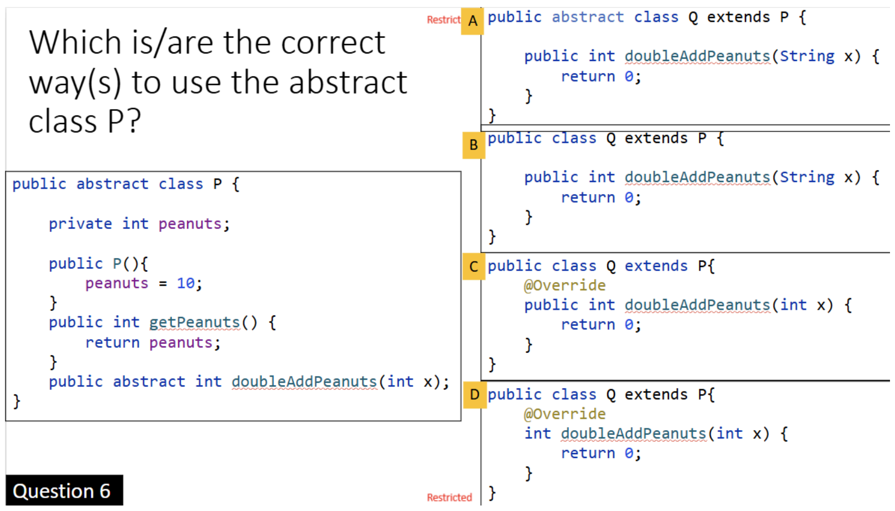


Overriding (Same method signature, different implementations): B, D
Overloading (Same method name, different parameter list): C

D d = new D();
Can call all methods in D, including inherited methods from C and A
Methods: apple(int x) from D, apple(String x) from C

A a = new D();
Can only call methods in A that are overridden in D.
Methods: apple(int x) from D, due to overriding.

A b = new B();
Can only call methods in A that are overridden in B.
Methods: apple(int x) (from B, due to overriding).



Which is/are the correct way(s) to use the abstract class P?

```java
public abstract class P {

    private int peanuts;

    public P(){
        peanuts = 10;
    }
    public int getPeanuts() {
        return peanuts;
    }
    public abstract int doubleAddPeanuts(int x);
}
```

A
```java
public abstract class Q extends P {

    public int doubleAddPeanuts(String x) {
        return 0;
    }
}
```

B
```java
public class Q extends P {

    public int doubleAddPeanuts(String x) {
        return 0;
    }
}
```

C
```java
public class Q extends P{
    @Override
    public int doubleAddPeanuts(int x) {
        return 0;
    }
}
```

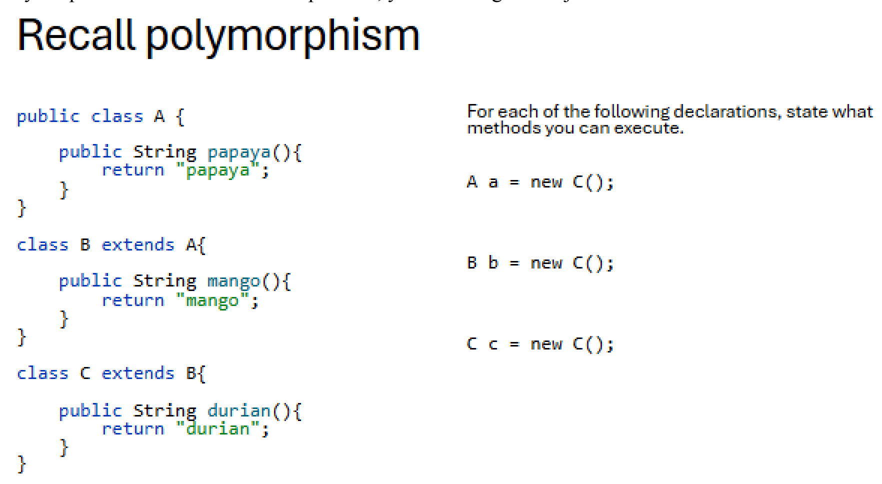
D
```java
public class Q extends P{
    @Override
    int doubleAddPeanuts(int x) {
        return 0;
    }
}
```

Question 6

A - Correct. Q remains abstract and so does not need to implement the abstract method exactly. It is simply a different method.
B - Incorrect. Method signature different.
C - Correct. Implementation is correct. Class Q implements the abstract method in P, so it can be concrete
D - Incorrect. Access modifier different. It is trying to change to default (package-private)

Week 4

Abstraction: in deciding the behaviour of an object, you can focus on what actions it can do and ignore the details of how each action is meant to be done
Encapsulation: methods and attributes together to form a class, good practice to hide attributes
Polymorphism: for a variable of a superclass, you can assign an object of a subclass



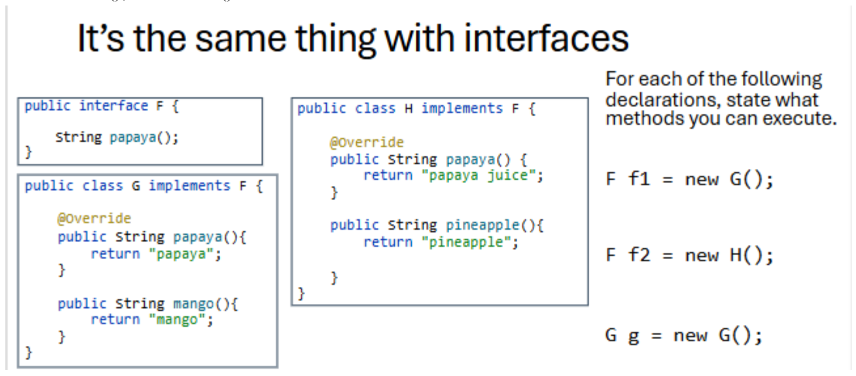
Recall polymorphism

```
public class A {

    public String papaya(){
        return "papaya";
    }
}
class B extends A{

    public String mango(){
        return "mango";
    }
}
class C extends B{

    public String durian(){
        return "durian";
    }
}
```

For each of the following declarations, state what methods you can execute.

```
A a = new C();

B b = new C();

C c = new C();
```

A a = new C(); // a.papaya()
B b = new C(); // b.mango(), b.papaya()
C c = new C(); // c.durian()



It's the same thing with interfaces

```
public interface F {

   String papaya();
}
```

```
public class G implements F {

    @Override
    public String papaya(){
        return "papaya";
    }

    public String mango(){
        return "mango";
    }
}
```

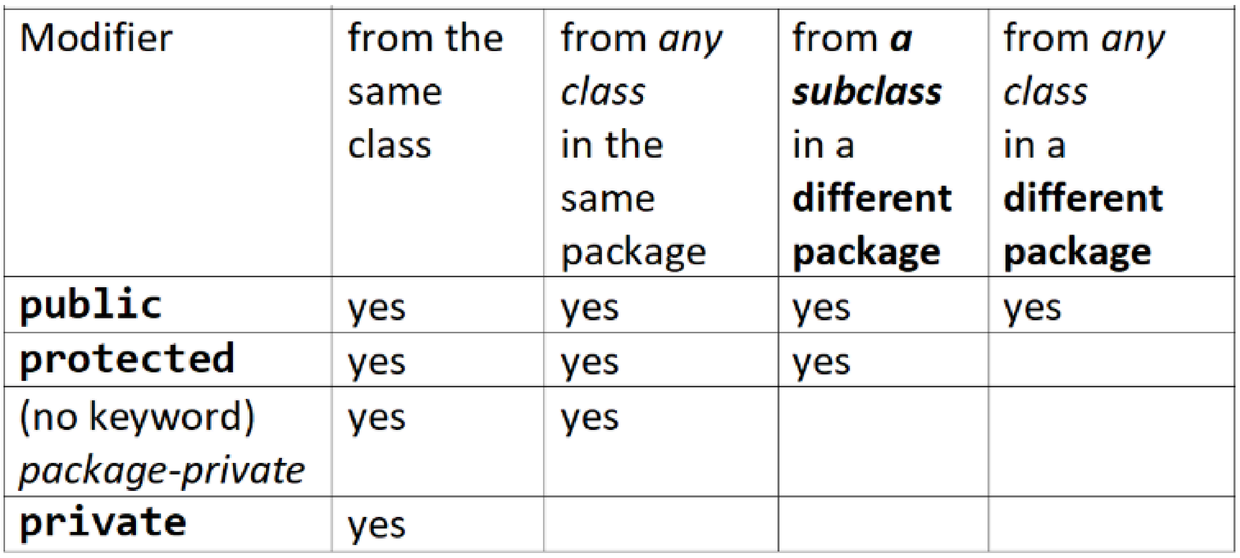
```
public class H implements F {

    @Override
    public String papaya() {
        return "papaya juice";
    }

    public String pineapple(){
        return "pineapple";

    }
}
```

For each of the following declarations, state what methods you can execute.

```
F f1 = new G();

F f2 = new H();

G g = new G();
```

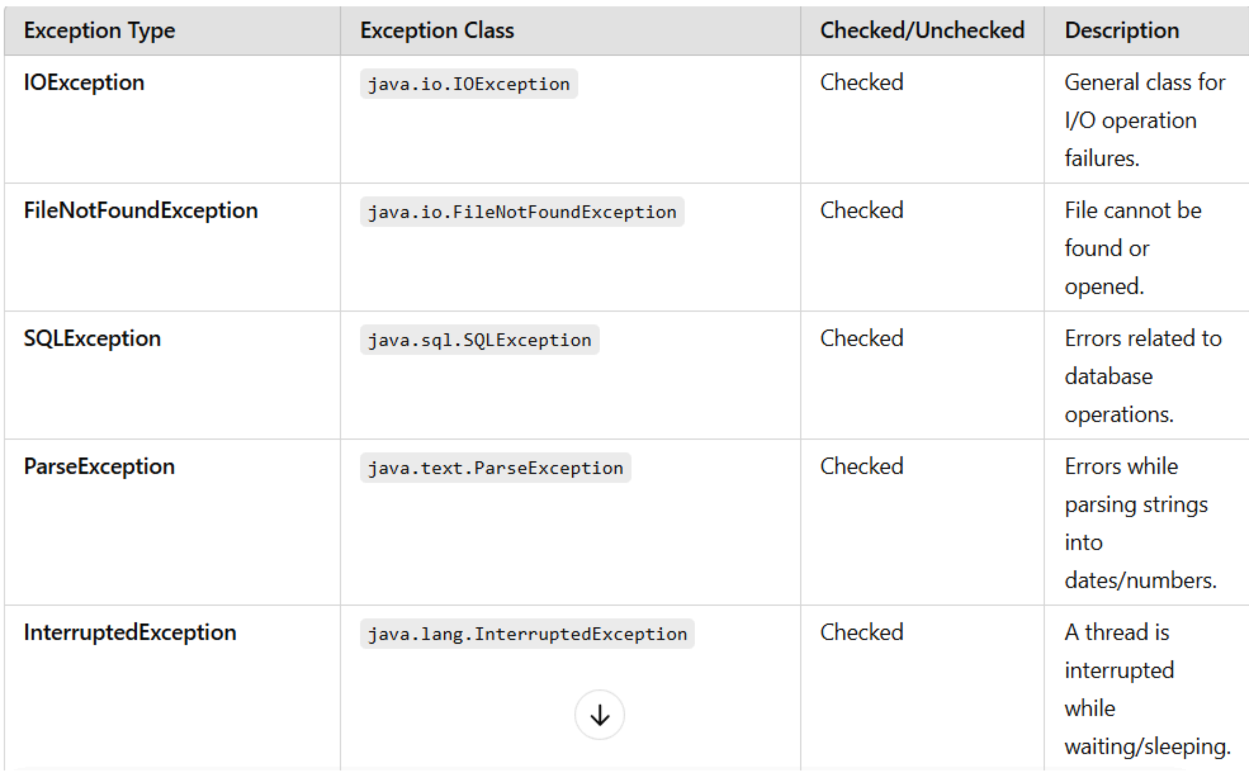
F f1 = new G(); //f1.papaya()
F f2 = new H(); //f2.papaya();
G g = new G(); //g.papaya();, g.pineapple();

Java Syntax



Modifier	from the same class	from *any class* in the same package	from **a *subclass*** in a **different package**	from *any class* in a **different package**
`public`	yes	yes	yes	yes
`protected`	yes	yes	yes	
(no keyword) *package-private*	yes	yes		
`private`	yes			

CHECKED EXCEPTIONS



Exception Type	Exception Class	Checked/Unchecked	Description
IOException	`java.io.IOException`	Checked	General class for I/O operation failures.
FileNotFoundException	`java.io.FileNotFoundException`	Checked	File cannot be found or opened.
SQLException	`java.sql.SQLException`	Checked	Errors related to database operations.
ParseException	`java.text.ParseException`	Checked	Errors while parsing strings into dates/numbers.
InterruptedException	`java.lang.InterruptedException`	Checked	A thread is interrupted while waiting/sleeping.

ClassNotFoundException	`java.lang.ClassNotFoundException`	Checked	Class not found at runtime via reflection.
InstantiationException	`java.lang.InstantiationException`	Checked	Attempt to instantiate an abstract class or interface.
NoSuchMethodException	`java.lang.NoSuchMethodException`	Checked	Requested method does not exist in a class.
NoSuchFieldException	`java.lang.NoSuchFieldException`	Checked	Requested field does not exist in a class.
ReflectiveOperationException	`java.lang.ReflectiveOperationException`	Checked	General superclass for reflection-related exceptions.
EOFException	`java.io.EOFException`	Checked	Unexpected end of file or stream.
MalformedURLException	`java.net.MalformedURLException`	Checked	Invalid URL format.

<u>UNCHECKED EXCEPTIONS</u>

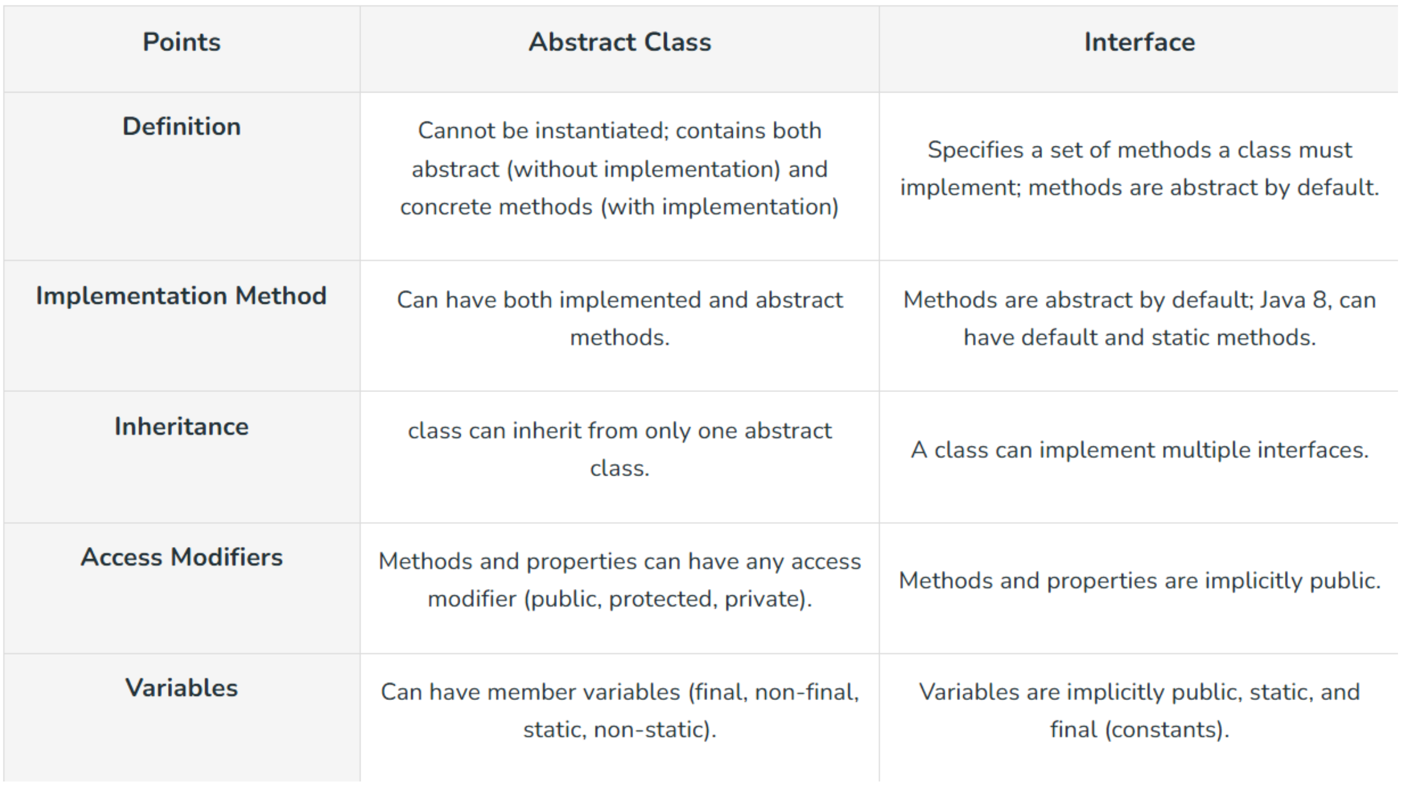
Exception Type	Exception Class	Checked/Unchecked	Description
NullPointerException	`java.lang.NullPointerException`	Unchecked	Attempt to access an object reference that is `null`.
ArrayIndexOutOfBoundsException	`java.lang.ArrayIndexOutOfBoundsException`	Unchecked	Accessing an array index that is out of bounds.
StringIndexOutOfBoundsException	`java.lang.StringIndexOutOfBoundsException`	Unchecked	Accessing an invalid string index.
ArithmeticException	`java.lang.ArithmeticException`	Unchecked	Math errors like division by zero.

IllegalArgumentException	`java.lang.IllegalArgumentException`	Unchecked	Passing an illegal or inappropriate argument to a method.
NumberFormatException	`java.lang.NumberFormatException`	Unchecked	Converting an invalid string to a number.
ClassCastException	`java.lang.ClassCastException`	Unchecked	Invalid casting between incompatible classes.
IllegalStateException	`java.lang.IllegalStateException`	Unchecked	Method is called at an inappropriate time.
UnsupportedOperationException	`java.lang.UnsupportedOperationException`	Unchecked	An unsupported operation is attempted.
ConcurrentModificationException	`java.util.ConcurrentModificationException`	Unchecked	A collection is modified while being iterated over.
StackOverflowError	`java.lang.StackOverflowError`	Unchecked (Error)	Infinite recursion causes stack memory overflow.
OutOfMemoryError	`java.lang.OutOfMemoryError`	Unchecked (Error)	JVM runs out of memory.

Key Differences:

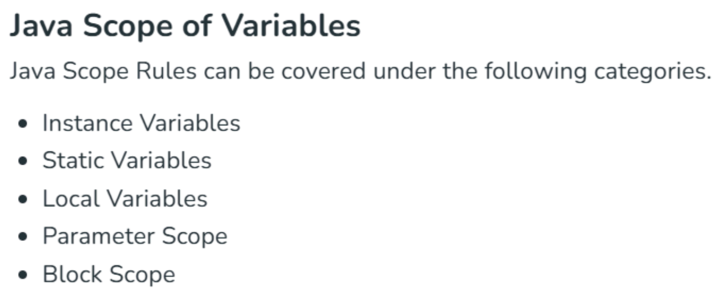
- **Checked Exceptions** must be handled using `try-catch` or declared using `throws` .
- **Unchecked Exceptions** (Runtime exceptions) do not require handling but can be caught if needed.
- **Errors** (`StackOverflowError` , `OutOfMemoryError`) are critical issues that typically should not be caught.

ABSTRACT CLASS VS INTERFACE



Points	Abstract Class	Interface
Definition	Cannot be instantiated; contains both abstract (without implementation) and concrete methods (with implementation)	Specifies a set of methods a class must implement; methods are abstract by default.
Implementation Method	Can have both implemented and abstract methods.	Methods are abstract by default; Java 8, can have default and static methods.
Inheritance	class can inherit from only one abstract class.	A class can implement multiple interfaces.
Access Modifiers	Methods and properties can have any access modifier (public, protected, private).	Methods and properties are implicitly public.
Variables	Can have member variables (final, non-final, static, non-static).	Variables are implicitly public, static, and final (constants).

SCOPE OF VARIABLES (src: Geek for Geeks)



Java Scope of Variables

Java Scope Rules can be covered under the following categories.

- Instance Variables
- Static Variables
- Local Variables
- Parameter Scope
- Block Scope

1. Instance Variables – Class Level Scope

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test {
        // All variables defined directly inside a class
        // are member variables
        int a;
        private String b;

        void method1() {....}
        int method2() {....}

        char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't affect scope of them within a class.
- Member variables can be accessed outside a class.

(see above for access modifiers)

2. Static Variables – Class Level Scope

Static Variable is a type of class variable shared across instances. Static Variables are the variables which once declared can be used anywhere even outside the class without initializing the class. Unlike Local variables it scope is not limited to the class or the block.

Example:

```
// Using Static variables
import java.io.*;

class Test{
      // static variable in Test class
      static int var = 10;
}

class Geeks
{
    public static void main (String[] args) {
          // accessing the static variable
          System.out.println("Static Variable : "+Test.var);
    }
}
```

Output

```
Static Variable : 10
```

3. Method Level Scope – Local Variable

Variables declared inside a method have method level scope and can’t be accessed outside the method.

```
public class Test {
    void method1()
    {
        // Local variable (Method level scope)
        int x;
    }
}
```

Note: Local variables don’t exist after method’s execution is over.

4. Parameter Scope – Local Variable

Here’s another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test {
    private int x;

    public void setX(int x) {
        this.x = x;
    }
}
```

The above code uses this keyword to differentiate between the local and class variables.

Example of Method and Parameter Scope:

```java
// Using Method Scope and Parameter Scope
public class Geeks
{
    // Class Scope variable
    static int x = 11;

    // Instance Variable
    private int y = 33;

    // Parameter Scope (x)
    public void testFunc(int x) {
        // Method Scope (t)
        Geeks t = new Geeks();
        this.x = 22;
        y = 44;

        // Printing variables with different scopes
        System.out.println("Geeks.x: " + Geeks.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }

    // Main Method
    public static void main(String args[]) {
        Geeks t = new Geeks();
        t.testFunc(5);
    }
}
```

Output

```
Geeks.x: 22
t.x: 22
t.y: 33
y: 44
```

5. Block Level Scope

A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

Example:

```
// Using Block Scope
public class Test
{
    public static void main(String args[])
    {
          // Block Level Scope
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

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

Output:

```
10
```

EXAMPLE OF INCORRECT USAGES AND THEIR ERRORS

```
class Test
{
    public static void main(String args[])
    {
          // local variable declared
        int a = 5;

          // for loop variable declared
        for (int a = 0; a < 5; a++)
        {
            System.out.println(a);
        }
    }
}


/*
Output:

6: error: variable a is already defined in method go(int)
       for (int a = 0; a < 5; a++)
                ^
1 error
*/
```

```
// Block Level Variables
class Test
{
    public static void main(String args[])
    {
         // Block Created
        {
             // variable inside block
            int x = 5;

             // Nested Block
            {
                 // Variable inside nested block
                int x = 10;
                System.out.println(x);
            }
        }
    }
}


/*
Output:

./Test.java:8: error: variable x is already defined in method main(String[])
                int x = 10;
                    ^

1 error
/*
```

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}


/*
Output:
11: error: cannot find symbol
System.out.println(x);

*/
```

CORRECT USAGE

Incorrect   Correct

```
// Correcting the error

class Test
{
    public static void main(String args[])
    {
        int x;
        for (x = 0; x < 4; x++) {
            System.out.print(x + " ");
        }

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

Output:

```
0 1 2 3 4
```

```java
// Java Program to demonstrate
// local variables - block level

class Test
{
    public static void main(String args[])
    {
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }

        int i = 20;
        System.out.print(i + " ");
    }
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 20
```