Flow Control and Exception Handling

5) Write code using the if and switch statements and Identify legal argument types for these statements.

if : General decision making

- The *if* conditional statement is used when you want to execute different bits of code based on the result of a Boolean test expression.
- They contain the keyword *if*, followed by a boolean test, followed by either a single statement or a block statement to execute if the test is true. An optional *else* keyword provides the alternative statement to execute if the test is false.
- The test must return a boolean value (true or false) and not 0 and 1 like in other languages.

```
if (x y) System.out.println("x is smaller than y");
```

An optional else keyword provides the alternative statement to execute if the test is false:

```
if (x y)

         System.out.println("x
is smaller than y");

     else System.out.println("y is bigger");
```

if you want to do more than just one thing, you can enclose those statements inside a block:

```
  if (x y )

         System.out.println("x
is smaller");

     else {

         System.out.println("y
is smaller");

         if (y z)


System.out.println("y is smaller");

         else System.out.println("z
is smaller");

     }
```

- An alternative for using the if and else keywords in a conditional statement is to use the conditional operator, sometimes called the ternary operator (ternary means three; the conditional operator has three parts). This operator is basically short−hand for an if−else statement. This is most useful for very short or simple conditionals

```
expression ? op1 : op2
```

The ?: operator evaluates expression and returns op1 if it's true and op2 if it's false.

switch: selection from a list of alternatives.

The if/else construct can be cumbersome when you have to deal with multiple selections with many alternatives.

```
switch (expression) {
   case value1: statement(s) ; break;
   case value2: statement(s) ; break;
   default: statement(s);
}
```

- The argument to a switch statement must be a byte, char, short or int and must be a constant or constant expression. It takes only one argument.
- If you omit a "break" at the end of a branch, after that branch is executed control falls through to execute all the remaining branches.

```
int x=2;
switch(x) {
  case 1: System.out.println("1");
  case 2:
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

ouput :

```
3
4
```

- There can only be one "default" branch and it doesn't have to be last. The "default" branch can be omitted. If it is preset, it is executed when none of the "case" values match the expression.

```
char myChar = 'c';
switch(myChar) {
   default :
   case 'a': System.out.println("I am in case a"); break;
   case 'b': System.out.println("I am in case b"); break;
```

output :

I am in case a

- If none of the cases match, and there is no default case, the statement does nothing.

---

6) Write code using all forms of loops including labeled and unlabeled use of break and continue and state the values taken by loop counter variables during and after loop execution.

Loops are of two forms. They are "Indeterminate loops" and "Determinate loops"

Indeterminate loops:
while:
This executes the body of the loop while a condition is true. It may never execute it if the condition is false.

```
while(x

  System.out.println(x);

  x++;

}
```

A while loop tests at the top. Therefore, the code in the block may never be executed.

do:
If you want the block to be executed at least once, you have to use the do version of the while loop

```
do {

  System.out.println(x);

} while(false);
```

Determinate loop:
for:
The for loop, repeats a statement or block of statements until a condition is matched. for loops are frequently used for simple iterations in which you repeat a block of statements a certain number of times and then stop.

```
for (int i = 1; i

   System.out.println(i);

}
```

for loop statement has three parts:

- initialization : is an expression that initializes the start of the loop. If you have a loop index variable to keep track of how many times the loop has occurred, this expression might declare and initialize it. Variables that you declare inside the for loop are local to the loop itself; they cease existing after the loop is finished executing.

condition : Boolean expression that tests the loop control variable. If the outcome of that test is true, the loop executes. Once the test is false, the loop stops executing.

- iteration : determines how the loop control variable is changed each time the loop iterates.

A comma separated for loop is allowed in the initial and increment sections, so that you can string together several initializations or increments

```
for(i=0, j=0; i
```

eg,

```
public class Test {

   public static void main(String arg[]) {

      int tot = 0;

      for(int i =0, j=10; tot >30; ++i, --j)
{

          System.out.println("i="
+ i +"j=" + j);

          tot +=(i+j);

      }

      System.out.println("Total : " tot);

   }

}
```

<u>ouput :</u>
Total : 0

It won't enter the loop at all because the condition says do the for loop when tot > 30, when tot = 0

An infinite loop will look like

```
for (;;) {....}
```

<u>break :</u>

The break keyword, when used with a loop halts execution of the current loop. If you've nested loops within loops, the break statement causes the control to pass to the next outer loop; otherwise, the program merely continues executing the next statement after the loop.

<u>break (labeled):</u>

```
 out: for (int i = 1; i

        for (int j = 1;
j


System.out.println("i is " + i + ", j is " + j);


if (( i + j) > 4)
break out; -----------------------

        } System.out.println("end")
;        |

   } System.out.println("the end");
```

output:

i is 1, j is 1
i is 1, j is 2
i is 1, j is 3
end
i is 2, j is 1
i is 2, j is 2
i is 2, j is 3
the end

break (unlabeled):

```
int a=0;

int b=0;

int c=0;

while (c 10) {
a = a + b;
System.out.println("a is " + a + ",c is" + c);
if ( a > c) break; ---------------------
b++; |
c++; |
} |
System.out.println("the end") ;
```

output :
a is 0, c is 0
a is 1, c is 1
a is 3, c is 2
the end

continue :

- Continue statements occur only in loops.
- When a continue statement is executed, it causes the flow of control to pass to the next iteration of the loop.
- It is used when you have nested loops and you want to break out of an inner one and start with the next iteration of the outer loop.

continue (labeled) :

```
 -------> months: for(int m=1;m

|

for(int d=1;d

 ------------------ if(m==2  d==1)
continue months;


System.out.println("m: " + m + ", d: " + d);


}

           }
```

output :

```
m: 1, d: 1
m: 1, d: 2
m: 3, d: 1
m: 3, d: 2
```

continue (unlabeled) :

```
int c1=0;

int c2=0;

while(c1 3) {

       c2++;
|

       System.out.println("c1: " +
c1 + ", c2: " + c2);      |

       if(c2==2) continue; ---------------------------------------

       c1++;

}
```

output :

```
c1: 0, c2: 1
c1: 1, c2: 2
c1: 1, c2: 3
c1: 2, c2: 4
```

Remember : In all the loops (for, while, and do), the loop ends when the condition you're testing for is met. If you want to exit the loop earlier than that, you can use the break and continue keywords.

7) Write code that makes proper use of exceptions and exception handling clauses (try catch finally) and declares methods and overriding methods that throw exceptions.

- If you mistyped a method name or made a mistake in your code, chances are that your program quit and spew a bunch of mysterious errors to the screen. Those mysterious errors are exceptions. Exceptions can be thrown by the system or thrown by you, and they can be caught as well.
- An exception is always an instance of a class derived from Throwable. Throwable can be split into two, they are: Error and Exception
- The Error class exceptions are handled by the system. Instances of Errors are internal errors and resource exhaustion inside the Jave run time system. There's not much you can do about them, beyond notifying the user and trying to terminate the program gracefully. These errors are rare and usually fatal.
- All user declared exceptions are subclasses of Exception.
- Only exception class objects can be thrown.
- Exception can be further subclassed as,

RuntimeException such as ArrayIndexOutofBounds, SecurityException, or NullPointerException.
Other exceptions such as EOFException and MalformedURLException.

- Runtime exceptions usually occur because of code that isn't very robust, which you can avoid and fix.
- Other exceptions are those which indicates that something very strange and out of control has happened. EOFExceptions, for example, happen when you're reading from a file and the file ends before you expect it to.
- In many cases, the Java compiler enforces exception management when you try to use methods that use exceptions; you'll need to deal with those exceptions in your own code or it simply won't compile.

  When overriding a method in a subclass,
- when overriding a method, the types listed in the overriding method throws clause can be a subset of the types listed in the overridden method throws clause.
- It can throw fewer or no exceptions.

Checked vs. Unchecked Exception :

- A checked exception is some subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses.
- Making an exception checked forces client programmers to deal with the possibility that the exception will be thrown. eg, IOException thrown by java.io.FileInputStream's read() method
- Unchecked exceptions are RuntimeException and any of its subclasses. Class Error and its subclasses also are unchecked.
- With an unchecked exception, however, the compiler doesn't force client programmers either to catch the exception or declare it in a throws clause. In fact, client programmers may not even know that the exception could be thrown. eg, StringIndexOutOfBoundsException thrown by String's charAt() method
- Checked exceptions must be caught at compile time. Runtime exceptions do not need to be. Errors often cannot be.

There are two choices to handle exceptions,

  - ♦ You can put a try block around the code that might throw the exception and provide a corresponding catch block that will apply to exception in question.
  - ♦ You might decide that if this exception occurs, your method can't proceed and should be abandoned. In such, case, you need not provide the try/catch construction, instead make sure that the method declaration includes a throws part that informs potential callers that the exception might arise.
- Using try/catch block : To catch an exception, you do two things:

You protect the code that contains the method that might throw an exception inside a try block.

You test for and deal with an exception inside a catch block.

```
public static String readString() {
  int ch; String r = "";
  boolean done = false;
  while (!done) {
    try {
       ch = System.in.read();
       if (ch 0 || (char)ch =='\n') done = true;
       else r = r + (char) ch;
    }
    catch(IOException e) { done = true; }
  }
  return r;
}
```

- A try block can be followed either by a catch block or a finally block or both
- A catch block must always be associated with a try block
- Only one catch block, that is the first applicable one, will be executed.
- A finally can never stand on its own ie, without try block
- A try block need not always be followed by a catch block
- try, catch, and finally are the keywords to deal with exceptions that may or may not occur.

```
try{
 System.out.println("Before an Exception");
mightThrow();
System.out.println("After throwing Exception");
}
catch (IOException e) {
System.out.println("Exception thrown");
}
finally {
System.out.println("finally");
}
```

output :

- If no exception thrown by mightThrow() the result will be,

```
Before an Exception
After throwing Exception
```

finally

- If IOException(or a subclass of IOException) is thrown the result will be

Before an Exception
Exception thrown
finally

- If mightTrhow() throws exception other than IOException ie, uncaught exception

Before an Exception
finally
>

Remember:finally method gets executed:

- if the try block completed normally (no exception)
- if one of the catch blocks handled an exception
- if an exception occurred that none of the catch block is handled

- finally() will be executed under any circumstances
- Specific exception should come first and the common exception should come at the end otherwise you will get a compiler error

Using throws clause : Instead of handling errors at the point where they are likely to occur, the error can be passed up the call stack by use of the throws clause when declaring a method. The throws keyword is used to identify the list of possible exceptions that a method might throw.

eg,

```
    public class factorial {

  static long[] table=new long[21];

  static {table[0]=1;}

  static int last=0;

  public static long method(int x) throws IllegalArgumentException {

    if (x>=table.length) throw new IllegalArgumentException("overflow; xis too large");

    if(x    while(last      table[last+1]=table[last] * (last+1);

      last++;

    }

    return table[x];

  }

  public static void main(String arg[]) {

//    method(25);//This method will throw "overflow" exception

//    method(-5);//This method will throw "must be non-negative" exception

  }
}

}
```

Exception Heirarchy :

```
Object

   |

   |

Throwable

   |

   |
```

Exception−−>ClassNotFoundException, ClassNotSupportedException, IllegalAccessException, InstantiationException, IterruptedException, NoSuchMethodException, RuntimeException, AWTException, IOException

RuntimeException−−>EmptyStackException, NoSuchElementException, ArithmeticException, ArrayStoreException, ClassCastException, IllegalArgumentException, IllegalMonitorStateException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, SecurityException.

IllegalArgumentException−−>IllegalThreadStateException, NumberFormatException

IndexOutOfBoundsException−−>ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

IOException−−>EOFException, FileNotFoundException, InterruptedIOException, UTFDataFormatException, MalformedURLException, ProtocolException, SockException, UnknownHostException, UnknownServiceException.

Remember :

- Unchecked exceptions
  - can just show up anywhere, and they don't have to follow any rules for overriding.
- Checked exceptions
  - An overriding method can't throw more or broader checked exceptions.
  - A method which throws a checked exception or calls a method which throws a cheked exception, must either declare it or provide a try/catch for that exception (or for one of that exception's superclasses).
- There are no rules even for Checked exceptions when overloading.