\*\* Types :

\* Signed Numeric Integer( non decimal )

\*\* You can assign integer literals in base decimal, binary, octal, and hexadecimal.

For octal literals, use the prefix 0 ;

For binary, use the prefix 0B or 0b ;

for hexadecimal, use the prefix 0X or 0x .

\*\* Underscores

- can’t be between or right after 0b 0x,

- can be placed after ‘0’ for octal form,

- You can’t start or end a literal value with an underscore.

- You can’t place an underscore prior to an L suffix,

- You can’t use an underscore in positions where a string of digits is expected for example system.out.println(Integer.parseInt(“12\_22”))

- it is legal to have multiple underscores \_\_ between two digits

int a = 2\_\_\_2

\*\* default type is int

\*\* default value for byte int short long field if not initialized is zero

\*\* sysout on primitive long field call Long.toString(x), not the case for int short byte

\* Character (unsigned integer)

\*\* Because your keyboard may not have keys to represent all these characters, you can use a value from \u0000 (or 0 )

to a maximum value of \uffff (or 65,535 ) inclusive

\*\* Internally, Java stores char data as an unsigned integer value (positive integer)

\*\* We can cast a negatif integer to char, char c = (char)-211

\*\* Directly assign negatif integer to char not working

\*\* default value for char field if not initialized is 0, but calling sysout on it will print space

\* Signed Numeric decimal

\*\* Underscores

- You can’t place an underscore prior to a D , d , F , or f suffix

- You can’t place an underscore adjacent to a decimal point.

\*\* Float 32 bit, double 64 bit

\*\* it is ok to write

double b = .2;

\*\* sysout on primitive float double field call Float|Double.toString(x)

\*\* defautl type for decimal is double

float a = 3\*2.2; // compile time error can’t convert from double to float

float b = 3\*2.2f // is ok

\*\* Math.round(0.5) → 1

\* when calling overloading method with primitive, the closer method is chosen, exemple :

float comput(float a, float b){print(44)}

float comput(double a, double b){print(22)}

comput (1,2) // print 44, a float is closer than a double to int.same thing with long byte short..

\* operands of mathematical operators are ALWAYS promoted to AT LEAST int. (i.e. for byte \* byte both bytes will be first promoted to int.)

and the return value will be AT LEAST int, reference type are unboxed before promotion

byte a = 2, b=3;

byte c = a\*b; // ko because a\*b return int

final byte d = 2, e=4;

c= d\*e // ok because Implicit Narrowing

// int\*float is promoted to float\*float, then

// float==double is promoted to double==double:

\* compound assignment operators += -= internally do an explicit cast, A compound assignment expression of the form E1 op= E2 is equivalent to E1 = (T)((E1) op (E2))

double d = 232222222;

byte c = 2;

c+=d ; // ok

\*\* Wrapper class :

\* All wrappers objects are immutable

\* Constructor

Byte byt = new Byte("Constructor [string] or [byte primitive] or [string and radix(base 16 2 10...)]");

Short shoort = new Short("constructor [string] or [short primitive] or [string and radix(base 16 2 10...)]");

Integer integer = new Integer("constructor [string] or [integer primitive] or [string and radix(base 16 2 10...)]");

Long longg = new Long("constructor [string] or [long primitive] or [string and radix(base 16 2 10...)]");

Float floatt = new Float("consturctor [string] or [float primitive]");

Double doublee = new Double("consturctor [string] or [double primitive]");

Character charr = new Character('constructor [char primitive]');

String stringg = new String("constructor [empty], [String], [StringBuilder], [StringBuffer], [byte[]...], [char []...]");

Boolean bool = new Boolean("constuctor [boolean] and [string]")

\* parseX methods : return primitive type

Byte Short Integer Long have method parseByte parseShort parseInt returning primitive type and

accepting [String] or [String,int radix] as an arguments

Float Double Boolean have only method parseFloat, parseDouble and parseBoolean with [String] argument

String and Character don't have parseX method

throws NumberFormatException

\* valueOf methods : return object type,

\* Byte, Short, Character, Integer have caching system, calling valueOf will return value from cache if exist

\* boolean valueOf return referance to Boolean.FALSE or Boolean.TRUE

\* Double, Float don't have caching system, Double.valueOf("2.2") == Double.valueOf("2.2") --> false

\* String valueOf(int a) valueOf(String b), valueOf(float f)...

\* throws NumberFormatException for string arg variant of the method

\* String class :

\*\* extends Object

\*\* Internally string is stored as private final char value[];

\*\* All Implemented Interfaces: Serializable, CharSequence, Comparable<String>

\*\* Because String is based on array of char, first value of string start at index 0

\*\* Methods :

- char charAt(int i) :

> throw unchecked exception StringIndexOutOfBoundsException that extends IndexOutOfBoundsException if i is out of range

- int indexOf(char a) :

- int indexOf(char a, int index) :

- int indexOf(String a):

- int indexOf(String a, int index):

> return -1 if char not found or index out of range

- String substring(int start) : return from start to end of string

- String substring(int start, int end) :

> throw unchecked exception StringIndexOutOfBoundsException that extends IndexOutOfBoundsException if out of range

> or end greater than start

- String replace(char a, char b) :

> replace "all" char a with b, return always new string even if a==b

> if a=b, return the same string

- String replace(CharSequence a, CharSequence b) :

> replace all CharSequence a with b

> always return new string even if if a=b

- String replaceAll replaceFirst : to replace using regex

- boolean startWith(String a)

- boolean startWith(String a, int index)

- boolean endsWith(String a)

- int compareTo(String a), int compareToIgnoreCase(String), String concat(String a), boolean contain(CharSequence a), contentEquals(CharSequence a)

contentEquals(StringBuffer sb), String copyValueOf(char[] data), String copyValueOf(char[] data, int offset, int count), boolean equals(String a),

boolean equalsIgnoreCase(String anotherString), getBytes(), boolean isEmpty(), String intern()

\*\* tips :

- For all compile time strings, if two or more contains the same value, than they have the same reference,

String he = new String(“hello”)

final String a = “llo”

String b = “llo”

“he”+ a = “hello’’ // true because compile time computation

“he”+”llo” = “hello” // true because compile time computation

“he”+ b = “hello’’ // false because run time computation

“hello”= he // false

“Hello” = he.intern() // true intern method is used to create an exact copy of heap string object in string constant pool.

\*\* StringBuilder class :

\*\* extends Object

\*\* Package java.lang

\*\* All Implemented Interfaces: Serializable, Appendable, CharSequence

\*\* Reason for exist dealing with larger strings or modifying the contents of a string often. Doing so will improve the performance of your code

\*\* In contrast to the class String , the class StringBuilder uses a non final char array to store its value.

\*\* 4 constructors exists : new StringBuilder()--> capacity 16, new StringBuilder(StringBuilder a), new StringBuilder(int a), new StringBuilder(String a)

\*\* Methods :

- append(...) with different overloading version, argument Object, int, float, double, long, boolean, String, StringBuffer, char[]...

- appent(CharSequence a, int start, int end) to append subset of CharSequence

- insert(int offset, ?) insert at a specified position, different overloading version with argument Object, int, String, CharSequence....

throw StringIndexOutOfBoundsException if offset is not valid

- StringBuilder delete(int start, int end) throw StringIndexOutOfBoundsException

- StringBuilder deleteCharAt(int index) throw StringIndexOutOfBoundsException

- StringBuilder reverse(), not possible to reverse substring of stringbuilder

- void ensureCapacity(int minimumCapacity)

- StringBuilder replace(int start, int end, String str)

- char charAt(int index), String substring(int start, int end), String substring(int start), CharSequence subSequence(int start, int end)

length(), setLength(int newLength), void trimToSize()---> reduce capacity if not used, setCharAt(int index, char ch), lastIndexOf(String str)

lastIndexOf(String str, int fromIndex), indexOf(String str), indexOf(String str, int fromIndex), int capacity(), getChars(...), length()

\*\* ArrayList class :

\*\* extends AbstractList extends AbstractCollection extends Object

\*\* Package java.util

\*\* All Implemented Interfaces: Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

\*\* Methods :

- boolean add(E e), void add(int index, E element)--> move if exist and add, boolean addAll(Collection<? extends E> c),

boolean addAll(int index, Collection<? extends E> c)

- clear()

- contains(Object o), ensureCapacity(int minCapacity), E get(int index), int indexOf(Object o), lastIndexOf(Object o), isEmpty()

E remove(int index), boolean remove(Object o), boolean removeAll(Collection<?> c), E set(int index, E element), size(), Object[] toArray()

<T> T[] toArray(T[] a), trimToSize()

\*\* Exceptions :

\*\* System.out.println(exception), a stack trace is not printed. Just the name of the exception class and the message is printed.

exception.printStackTrace(), a complete chain of the names of the methods called, along with the line numbers, is printed.

It contains the names of the methods in the chain of method calls that led to the place where the exception was created going back up

to the point where the thread, in which the exception was created, was started.

\*\* e.printStackTrace() don’t stop execution in catch block, any line after catch will execute

\*\* Only System.exit() in try or catch block can omit execution of finally, return; not...

\*\* If return statement exist in catch try and final, the one in final is used

\*\* catch block cannot come after a finally block

\*\* its ok to have try block without catch block, but in this case finaly block is mondatory, try block can't be alone

\*\* ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException extend IndexOutOfBoundsException

\*\* uncheckedException = Error + runetimeException

\*\* checkedException = Exception others than runtimeException

Throwable --> Error --> uncheckd exception Error subclass

Throwable --> Exception --> RuntimeException

Throwable --> checkedException(IOException, FileNotFoundException...)

\*\* A checked exception is a subclass of the class java.lang.Exception , but it’s not a subclass of java.lang.RuntimeException .

It’s interesting to note, how- ever, that the class java.lang.RuntimeException itself is a subclass of the class java.lang.Exception .

\*\* With Unchecked exception the compiler don’t requires you to add try catch or throws

\*\* A NullPointerException will be thrown if the expression given to the throw statement results in a null pointer.

RuntimeException re = null;

throw re; // throw nullpointer exception

\*\* In case of multiple catch block, subclass must be before superclass, else it is unreachable code so compile time error

\*\* Ternary Operator ? :

\*\* Not permitted for either the second or the third operand expression of the ? operator to be an invocation of a void method.

\*\* Both sides of : should return some value even if different type

Math.random()<.05? break : continue; // not valid

\*\* Lambda Expression :

\*\* Lambdas work only with functional interfaces—interfaces that define exactly one abstract method.

(int e) → { e.getPerformanceRating() >= 5; return; };

\*\* Each lambda expression has multiple optional and mandatory sections :

- Parameter name 'e' depend on if the abstract method define a parameter

- '()' is mandatory if abstract method has no parameter, multiple parameter or if parameter type is present

else if it define only one parameter '()' are not mandatory

- Parameter type int is optional

- {} are optional, if exist --> ; must exist after them if not in method call, in method call compile error if ; after {}

- If expression has ';' than {} must exist

- If {} exist and method return something, we must have return keyword

- If we use keyword return we must use {} even for void method

- We can have return; if abstract method is void

- (void) -> …. Is not valid

- parameters order is important

\*\* Predicat<T> generic interface offred by java that define one method test(T t) and return boolean No neeed to define your one method in this case

\*\* Switch case :

\*\* Boolean can’t be used in switch case only

int | Integer

byte | Byte

short | Short

char | Character

enum, String

\*\* Switch work fine when no steatements exists after case exemple :

switch(x){

case 2:

case 22:

case 80:

default :

}

\*\* case steatements can't have same value : here compile error because 1\_000\_000==1000000

    case 1\_000\_000 : System.out.println("A million 1");

case 1000000 : System.out.println("A million 2");

\*\* Import statement

\*\* “import static” they need to be in this order import → static

\*\* import static and wildcard apply for class

import static package.class.\*; // ok

import static package.class.fieldstatic; // ok

import static package.\* // kooo

\*\* directly Import static method is without ‘()’

\*\* Date and Time API :

\*\* A Temporal interface defines a representation of a date, time, or a combination of both, depending on the implementation, LocalDate LocatTime...

\*\* LocalDate : Immutable, private constructor

- To create a LocalDate :

LocalDate.of(2015, 12, 27);

LocalDate.parse("2025-08-09")

LocalDate.now(),

- toString() ---> uuuu-MM-dd

- Methods getXXX are used to return day month year… getDayOfMonth() getMonthValue() getYear()

- minusXX() minus day month year from a given LocalDate minusMonths(2)

- plusXX() add day month year to a given LocalDate plusDays(1) plusWeeks(7)

- withXX(), withDayOfMonth(int), withDayOfYear(1) to replace value in the given LocalDate

- Method atTime(...) is used to convert LocalDate to LocalDateTime

- DateTimeParseException is triggered if wrong value is passed to parse() or of(), DateTimeException for atTime() method

- DateTimeParseException is a runetime exception

- LocalDateTime greatDay = LocalDateTime.parse("2015-01-01");//2

- String greatDayStr = greatDay.format(DateTimeFormatter.ISO\_DATE\_TIME); // will throw exception because time part not exist

\*\* LocalTime : Immutable, private constructor

- to create LocalTime, LocalTime.of(h, m) LocalTime.of(h, m , s) LocalTime.of(h, m , s, nano) LocalTime.now(); LocalTime.parse("15:08:23")

- LocalTime doesn’t define a method to pass a.m. or p.m

- toString() ---> HH:mm:ss.SSSSSSSSS, print only non zero part it can be HH:mm for exemple if ss=0 and sssssssss=0

- getXX() to query LocalTime on its hour, minutes, seconds, and nanoseconds getHour() getMinute()... without (s) No getHours(), getMinutes(), getSeconds()

- isAfter() isBefore() to check whether a time is after or before the specified time

- minusHours() , minusMinutes() plusHours() , plusMinutes() with s

- The withHour() , withMinute() , withSecond() , and withNano() methods accept an int value and return a copy of LocalTime with the specified value altered

- atDate() is used to combine LocalDate with LocalTime and return LocalDateTime atDate accept only LocalDate as an argument

\*\* LocalDateTime : Immutable, private constructor

- parse("2050-06-05T14:00:00") now() LocalDateTime.of(2050, 6, 5, 14, 30, 0)

- The LocalDateTime class uses the letter T to separate date and time values in its printed value.

- toString() ---> uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS, print only non zero part it can be HH:mm for exemple if ss=0 and sssssssss=0

\*\* Period : Immutable, private constructor

- Period.of(1, 2, 7) 1 year, 2 months, and 7 days, Period.ofYears(2) 2 years,

Period.ofDays(-15) period of -15 days,

Period.between(LocalDate, LocalDate) first LocalDate is included in count, the second not

- You can add or subtract Period instances from the LocalDate and LocalDateTime

- Period.ofDays(35) A period of 35 days is not stored as 1 month and 5 days. Its individual elements, that is, days, months, and years,

are stored the way it is initialized.

- Periode.parse(“PnYnMnWnD”) p stand for parse, n is an number negative or positive

- Order is very importante nY -> nM -> nW-> nD, nW can’t preced nM or come after nD

- lowercase uppercase no matter Y y M m ….

- nW is converted to nD, exemple 5w-1d ----> 34D

- (-) sign before letter P mean all values are negatives

- Period can be used as a argument to methods plus() and minus() of class LocalDate and LocalDateTime

- In the class Period , both the getXXX() methods and minusXXX() plusXXX() methods use the plural form: getYears() , minusHours()

- When you subtract Period instances using the minusXXX() methods, the individual elements are subtracted. Subtracting P10D from P1M returns P1M-10D and not P20D

- The method multipliedBy(int) in the class Period is used to modify all elements of a Period instance. Period doesn’t define a “divideBy()”

\*\* DateTimeFormatter :

- used to format and parse date and time objects

- three way of instantiation DateTimeFormatter :

\* Using static method .ofLocalizedDate ofLocalizedTime ofLocalizedDateTime

\* Using the public and static fields of this class ISO\_DATE BASIC\_ISO\_DATE …

\* Using method ofPattern and supply a string pattern

- ofLocalizedDate , ofLocalizedTime , and ofLocalizedDateTime format date and time objects according to the locale

(language, region, or country) of the system on which your code executes. So the output might vary slightly across systems.

Args ofLocalizedXXX : FormatStyle.MEDIUM FormatStyle.FULL FormatStyle.LONG...

- FormatStyle is a enumeration

- format() method in DateTimeFormatter has argument of LocalXXX, in class LocalXXX format() has argument DateTimeFormatter

- format method return string, not a static method in LocalXXX andDateTimeFormater

- if you're confused between M, m, D, and d, remember that an upper- case letter represents a bigger duration period.

So M is for month and m is for minutes. Similarly, D represents day of year; d represents day of month.

- If you access Java’s source code, you’ll notice that the format and parse methods in date and time classes simply call

the format and parse methods on a DateTimeFormatter instance.

- parse() method is static in LocalXXX and instance method in DateTimeFormatter

- you cannot format LocalDate with a formatter that expects time component such as DateTimeFormatter.ISO\_DATE\_TIME,

but you can format LocalDateTime using date formater such as DateTimeFormatter.ISO\_DATE

\*\* java.time.temporal.TemporalAdjusters

- TemporalAdjuster(without s) is a functional interface which has many predefined implementations in the TemporalAdjusters class.

- TemporalAdjusters return Common and useful TemporalAdjusters.

- Two way of using TemporalAdjusters :

\* thisAdjuster.adjustInto(temporal) // temporal is the Temporal interface implemented by LocalX...

\* temporal.with(temporalAdjusters)

TemporalAdjusters.lastDayOfMonth()

TemporalAdjusters.firstDayOfNextMonth()

\*\* Narrowing :

double -> float -> long -> int -> char -> short -> byte

\*\* Implicit Narrowing -> assign larger type value to less larger type variable) primitive conversion(without explicit cast),

- We can do Implicit Narrowing if the following conditions are met :

\* The source variable or expression is a constant of type byte, char, short or int

\* The target variable is of type byte short or char

\* The source value is representable in the target variable,

int a = 10;

byte b = a; // ok

int c = 10030893;

b=c // nok, b value can’t fit in c

- Implicit narrowing can’t be applied to float, long, double types

char a = 30L; // compile time even if 30 fit in 'a'

- Implicit norrowing not work with method call,

Short f = new Short(9); // compile time erreur, even if 9 is a constant

Long a = new Long(2); // ok because its widening

\*\* Cast is used for explicit narrowing for from larger to smaller type

\*\* Widening :

byte -> short -> char -> int -> long -> float -> double

\*\* widening is implicit for all numeric primitive type

\*\* Widening and autoboxing not work implicitly we need to cast to the same type :

long a = 2;

Double c = a;// not work

Double d = (double)a; // ok

Double d = (Double)a; ko

\*\* Widening of Object is not allowed implicitly and with casting

Integer a = 3;

Double c = a; /ko

Double d = (Double)a; //ko

Double d = (Double)((Object)a); // ko

Double d = (double)((int)a); // ok

\*\* In method calling Widening of Object is not possible so go(Short) cannot call go(Integer) since they are not in the same inheritance hierarchy.

\*\* You CANNOT widen and then box. Eg. go(int) cannot call go(Long) since to call go(Long) the compiler need to

convert int to Integer then Integer to Long which is not possible.

\*\* Widening beats boxing eg. go(int) will call go(long) instead of go(Integer) if both are available

\*\* Widening beats var-args eg go(byte,byte) will call go(int,int) instead of go(byte...x) method.

\*\* Boxing beats var-args eg go(byte,byte) will call go(Byte,Byte) instead of go(byte...x) method.

\*\* widening depends on inheritance tree. Eg. go(dog) can call go(Animal)

Generic--->Animal--->Dog

go(new Dog); will call go(Animal a) if go(Generic a) and go(Animal a) exist both

\*\* You can box and then widen if same inheritance hierarchy. Eg. An int can boxed to Integer and then widen to Object

\*\* If primitive type is passed to equals(Object a) method, compiler will do autoboxing than object widdening

\*\* autoboxing and unboxing :

\*\* autoboxing and uboxing are implicit from type to the same type

\*\* Interface :

\*\* All abstract, default, and static methods in an interface are implicitly public , so you can omit the public modifier. In addition,

an interface can contain constant declarations. All constant values defined in an interface are implicitly public , static , and final .

Once again, you can omit these modifiers.

\*\* any method without body is abstract implicitly even if you omit it

\*\* Abstract class :

\*\* In abstract class, abstract keyword is mondatory else we need to specify body

\*\* Class containing abstract method must be abstract

\*\* Class can be abstract without abstract method

\*\* A static method isn’t associated with a particular object but rather is called on Type, if static method is overloaded in child class, Polymorphism don't applay

class a{ public static void m1() {System.out.println("hello");} }

class b extends a{ public static void m1() {System.out.println("world");} }

a dd = new b();

dd.m1(); // will print hello

\*\* overriding vs overloading

\*\* https://4.bp.blogspot.com/-D\_zk1PmEJ5Q/URpJA7hcWoI/AAAAAAAAHeg/0F7wb3aJkJE/s1600/overriding%2Bvs%2Boverloading.png

\*\* if return type is primitive it must be the same when overriding

\*\* static methods are never overridden, they may be hidden by a static method of a subclass with the same signature but we Can't reduce visibility of the static method

\*\* abstarct method implemented in subclass is overriding

\*\* for loop

\*\* The third part of the for loop does not allow every kind of statement. It allows only the following statements here:

Assignment, PreIncrementExpression, PreDecrementExpression, PostIncrementExpression, PostDecrementExpression, MethodInvocation, and ClassInstanceCreationExpression.

Thus, Math.random()<0.5 is not valid here, and so this will not compile.

- Not permitted to pass void method as argument to System.out.println

- We can instantiate Object class // new Object()

- It is allowed to have main method in abstract class, main method is static, main is case sensitive, main must be public

- int[] x = new int[0]; is a valid code even if size is zero

- String, StringBuilder, and StringBuffer and Boolean and Integer and Long and Short are all final also System, Number is not final

- while (false) { x=3; } → compile time error because x=3; is unreachble

- if(false) {x=3; } → no compile time error, this is an exception to the rule, the compiler

- if(DEBUG){ System.out.println("beginning task 1"); }

//Here, the DEBUG variable can be set to false in the code while generating the production //version of the class file, which will allow the compiler to optimize the code by removing the //whole if statement entirely from the class file.

- Break label work only if “label” is within the scope of “break label”

JILL: System.out.println(c);

if (c > 3) break JILL; else c++;

// Here a compile time error because JILL label is not in the scope of break JiLL

- The statement iA[i] = i = 30 ; will be processed as follows:

iA[i] = i = 30; => iA[0] = i = 30 ; => i = 30; iA[0] = i ; => iA[0] = 30 ;

- If a class or enum is public in a file, it must have the same name of the file.java, so we can’t add multiple public class or enum in the same file\*

- Local variable must be initialised, primitive and object both before any usage, null is a valid initialization for object

- A class (or an interface) can invoke a default method of an interface that is explicitly mentioned in the class's implements clause (or the interface's extends clause) by using the same syntax i.e.

<InterfaceName>.super.<methodName>. However, this technique cannot be used to invoke a default method provided by an interface that is not directly implemented (or extended) by the caller.

- super.super.Method not work, wa can only acces direct super method

- A class cannot have two methods with the same signature in its scope where one is static and one is instance.

- Method returning null pointer to object can be used to access static field of the same object look q5 exam 7

- Array is Object like others, method isArray() of Class return true if the object is an array

- You can't specify array length if you are initializing it at the same place.

new Object[1]{ new Object() }; // compile time error

- In an array access, the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated.

int[] a = { 1, 2, 3, 4 };

int[] b = { 2, 3, 1, 0 };

System.out.println( a [ (a = b)[3] ] ); // print 1

//First a[ is loaded as {1,2,3,4} than (a=b)[3] is evaluated to 0 so a[0] → 1

- In Java, fields are not overridden, they are hidden, polymorphism don’t apply to field as with static method, you can always access the two fields.

class A{int f =2;}

class B extends A{ int f = 3; }

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

sysout(a.f) // return 2

sysout(aa.f) // return 3

- First, static statements/blocks are called IN THE ORDER they are defined. Next, instance initializer statements/blocks are called IN THE ORDER they are defined.

Finally, the constructor is called.

- The package statement, if exists, must be the first statement in a java code file.

- Final attributes must be initialized either directly or in an instance block(if not static attribute) or static block(if static attribute) or in a constructor else compile error

- The getClass method always returns the Class object for the actual object on which the method is called irrespective of the type of the reference.

- For q 15 exam Test 5 : A class or interface type T will be initialized immediately before the first occurrence of any one of the following:

\* T is a class and an instance of T is created.

\* T is a class and a static method declared by T is invoked.

\* A static field declared by T is assigned.

\* A static field declared by T is used and the field is not a constant variable (final static ) (§4.12.4).

\* T is a top level class (§7.6), and an assert statement (§14.10) lexically nested within T (§8.1.3) is executed.

\* A reference to a static field (§8.3.1.1) causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface.

- The left operand of a binary operator is evaluated before its right operand.

- The Java programming language guarantees that every operand of an operator (except the conditional operators &&, ||, and ? :) appears to be fully evaluated before any part of the operation itself is performed.

- The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

- There is difference between order of evolution and associativity

- The operands of an operator are evaluated before the operator is evaluated. Consider the following expression:

((x=4) \* x)

- First, the left operand of \* is evaluated; it produces the value 4. Then the right operand of \* is evaluated. Since evaluation of the left operand set x to 4, evaluation of the right operand produces 4. Finally, the \* operator itself is evaluated, producing the value 16.

- There is not just one rule of precedence but multiple rules that are at play here. As per 15.7.1,

"The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.".

The first operator that the parser encounters is ||. Therefore, it must first evaluate the left side of it, which is a= true. Then whole of the right side of || becomes the second operand. Thus, the given expression will be grouped as (a = true) || ((b = true) && (c = true)). Since || is a short circuit operator and since the left side is true already, there is no need to evaluate the right side.

- by default java.long package is imported in evry class file

- class from default package can't be used in package class

- interface static method are not inherited by class or interface, static method from class are inherited

- Constructor must declare all the checked exceptions declared in the base constructor (or the super classes of the checked exceptions).

They may add other exceptions as well. This behavior is exactly opposite from that of methods.

The overriding method cannot throw any checked exception other than what the overridden method throws. It may throw subclasses of those exceptions as well.