OOPS concepts

1. Objects and classes
2. Encapsulation - Holding data and its operations together and restricting access
3. Abstraction - Showing only the calling details without exposing the implementation details
4. Inheritance
5. Polymorphism

Class JavaBasics{

int a;

public JavaBasics(int a){

a=a;

}

}

This will always print 0 as a will always be considered as local variable without the **this** keyword

JavaBasics@1 (Hexadecimal representation of the hashcode())

**Var-args**

print() and format() function uses var args

methodA(int…varargs)

Initialization blocks are executed before the constructor

Nested classes

1. There are 2 types of nested classes static and inner(non-static)
2. Nested classes can access both static and no-static members of the outer class
3. Inner classes methods can be called directly or by calling outer class methods

finalize() method is called just before garbage collection (Object class method)

System.gc() --> Runtime.getRuntime().gc()

Constructors have no return type so writing void will be considered as method

Method overloading with different return type results in compiler error

Covariant return type are applicable for method overriding

class A{

public Object method1(){return null;}

}

class B extends A{

public Integer method1(){return null;}

}

This is example for covariant return type

class A{

public int method1(){return 10;}

}

class B extends A{

public float method1(){return 1,5;}

}

This will not compile (incompatible return type)

Downcasting leads to ClassCastException which is RunTimeException

Vehicle v=new Bus() (Upcasting)

Bus b=(Bus)v (Downcasting)

Final variables are initialized inside constructors, if there are multiple constructors then final variable needs to be initialized in each of them or call the constructor which

initializes the final variable

Private methods can be overridden inside inner class

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Keywords** | **Class** | **Interface** | **Constructor** | **Method (Class)** | **Method (Interface)** | **Variable (Instance)** | **Variable (local)** | **Variable (interface)** |
| public | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes |
| private | No | No | Yes | Yes | No | Yes | No | No |
| protected | No | No | Yes | Yes | No | Yes | No | No |
| Default (Package private) | Yes | Yes | Yes | Yes | No | Yes | No | No |
| final | Yes | No | No | Yes | No | Yes | Yes | Yes |
| abstract | Yes | Yes | No | Yes | Yes | No | No | No |
| static | No (Nested classes can be static) | No | No | Yes | Yes | Yes | No | Yes |
| default | No | No | No | No | Yes | No | No | No |

When a class extends from another class and implement interface, extend should come before implements

Interface don't have constructors but abstract class have

class A{

static int a;

int b;

static class B{

//Nested static class

//Can access only a

}

class C{

//Inner class

//Can access both a and b

}

}

A.B obj=new A.B() //instantiating static class object

A objA=new A(); //instantiating inner class object

A.C objC=objA.new C();

A.C obj=newA().newC() //Another way of instantiating inner object

Method hiding - When subclass has static method with same signature as base class, reference variable type determines which static method will be called

Method overriding - When subclass has same method signature as base class, object created determines which method will be called

Interface static methods cannot be called from the reference of its implementations

interface A{

static void show(){}

class B implements A{

static void show() {}

}

A.show()

B.show()

A a=new B();

a.show()

A class cannot have static and non-static method with same name

You cannot override instance method with static method or vice versa

First static blocks, instance blocks then parent constructor and class constructor are called

Only static fields or methods can be called from static (methods/blocks)

**Creating Immutable classes**

**Immutable object do not state the change after creation and are hence thread safe**

1. Final class
2. Final variables
3. No setter methods
4. Parameterized constructor
5. If the class has mutable object then in parameterized constructor the clone of the mutable object should be used and if the mutable object is returned in getter method

then copy of mutable of object should be returned

StringBuffer and StringBuilder are final classes but not immutable and they inherit from the same base class

They have the same methods except that the methods are synchronized in StringBuffer

Negative numbers are stored as 2's complement of actual value (Flip all bits and add 1)

Binary rep of -6 is 11111111111111111111111111111010

6 | -6

00000110 --> 6

11111010 --> -6

--------------

11111110 --> The 2's complement of this value is taken and converted to equivalent decimal

00000001

+1

00000010 --> -2

**Bitwise operations are applicable for integers (long, short, byte, int, char) and return integer**

**If the operands are boolean then they return boolean value (logical operators)**

In case of right shift operation

When an input number is negative, where the leftmost bit is 1, then the empty spaces will be filled with 1

When an input number is positive, where the leftmost bit is 0, then the empty spaces will be filled with 0

-6>>2

11111010 --> -6

11111110

00000001

+1

00000010 --> -2

-6>>>2

00111110 --> 62

>> (right shift),<< (left shift), >>>(unsigned right shift)

Operator precedence

|  |  |
| --- | --- |
| **Operators** | **Precedence** |
| postfix | *expr*++ *expr*-- |
| unary | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**Shutdown Hooks**

Shutdown hooks are executed when JVM is shutting down normally

JVM can be shut down by calling Ctrl C, kill

There can be more than 1 shutdown hooks, however order of execution may not be guaranteed

Shutdown hooks may not be executed sometimes - calling kill -9 / Runtime.halt()

Shutdown hooks cannot be added/removed within shutdown hook throws exception

During the execution of shutdown hooks it can only be terminated by Runtime.halt()

**UncaughtExceptionHandler**

Used to handle exceptions without catching them

1. Each thread can define its own exception handler, JVM first checks for this handler
2. ThreadGroup class has implementation of UncaughtExceptionHandler, so if no specific thread handler exists
   1. It will call the parent Thread Group uncaughtException()
   2. If no parent exists and DefaultExceptionHandler exists for that thread then it is called
   3. If no DefaultExceptionHandler exists then the thread will terminate passing the exception to main thread
3. Steps 1-2 are now repeated for main thread

System.arraycopy(source array, source position, destination array, destination position, no of elements)

If the size of the elements copied is greater than the size of the destination array then ArrayIndexOutOfBoundsException is thrown

For the primitive array it’s a deep copy and for object array it is a shallow copy

**Shallow copy** means a new copy is created but the internal objects are shared between the copy and actual object

**Deep copy** means new copy will have its own internal objects and actual object its own

BufferedReader is faster than Scanner bcoz Scanner parses the input using regex

BufferedReader is synchronized

**Streams and Java8**

Predicate<T> - returns true/false for operation on T

Behaviour parameter is achieved through

Anonymous classes (too much verbose)

Lambda expressions

return is an statement not an expression ()->return "Happy"; is not valid

A functional interface can have multiple default methods but exactly one abstract method

The abstract method of the functional interface is functional descriptor

Lambda expression can be assigned to variable or passed as parameter to method

**Behaviour parameterization uses strategy pattern**

Passing behaviour as parameters to methods and use them internally to provide different behaviour

Examples

sort() in List<> takes different implementations of Comparator

Thread class take different implementations of Runnable

Behaviour parameterization can be used to address continuous change in requirements

Passing code as method argument can be achieved by anonymous inner classes but they are not concise as lambda expressions

Lambda is an anonymous function

(Parameter)->expression

(Parameter)->{statement;}

Lambda expressions can be used where functional interfaces are defined

Functional interface-Interface declaring only one abstract method

Examples- Comparator, Runnable, Callable

**Functional interface can have default methods but only 1 abstract method**

Lambda expression provide implementation of abstract method in functional interface

Functional descriptor- signature of abstract method in functional interface

Lambda expression can be assigned to variable or passed to a method expecting functional interface

Lambda signature

|  |  |  |
| --- | --- | --- |
| Predicate<T> | Boolean test(T) | T->boolean |
| Function<T,R> | R apply(T) | T->R |
| Consumer<T> | void accept(T) | T->void |
| Supplier<T> | T get() | ()->T |

java.util.function - package containing functional interfaces

Boxed variables take more memory and require additional lookups for finding the primitive data type

IntPredicate avoids boxing while Predicate<Integer> will perform boxing

Lambda expression cannot throw a checked exception

For a lambda expression to throw exception define custom functional interface that throws checked exception

Or have try catch block inside the lambda expression

Target Type - Variable or the method parameter the lambda expression points to

If ()->expression is same as ()-{expression;} then the latter is expression statement and the return type is void

Expression statements are

* Method Invocations
* Assignments
* Increment/Decrement expressions
* Class Instance Creation expressions

Lambda's can capture static, instance and local variables. Local variables should be final or effectively final

Example

Predicate<String> p=s->list.add(s);

Consumer<String> c=s->list.add(s); // valid since it is an expression statement

Consumer<String> c=s-> s.length();

Type for the parameters in the lambda expression can be omitted because the signature is available

Capturing lambdas- Variables that are not part of parameter list in lambda expression and are defined outer

Instance or static variables and local variables declared as final or act as final ie not reused after captured in lambda can be used in lambda expressions

Method references shorthand for lambda expressions, better readability

Constructor references

Comparator - reversed(), thenComparing() - compose methods

Predicate - negate(), and() , or()

Function - andThen() ,compose()

Function<Integer, Integer> f1= x->x+1;

Function<Integer, Integer> f1= x->x\*1;

Function<Integer, Integer> f2=f1.andThen(f2) - (x+1)\*1

Function<Integer, Integer> f2=f1.compose(f2) - (x\*1)+1

Streams

Collections are data structures, they deal with how data will be stored and accessed

Stream is sequence of data of particular type which deals with computation of data

Stream is a fixed data structure you cannot add or remove elements, at any point it will fixed no of elements

Streams is sequence of data of particular element type originating from data source which may be collection, I/O etc.

Streams can be consumed only once, if stream is consumed then a new stream needs to be generated from data source

Collections use external iterations

Intermediate operations-operations that connect the streams and return stream value eg. filter, map, limit

Terminal operations-operations that close the stream and return non stream values eg; collect

Computations are performed when terminal operation is called

|  |  |  |
| --- | --- | --- |
| Filtering using predicate | filter() | Intermediate |
| Filtering unique elements | distinct() | Intermediate |
| Truncating stream  Short circuit | limit() | Intermediate |
| Skipping elements | skip() | Intermediate |
| Apply function to each element of stream | map() | Intermediate |
| At least one match  Short circuit | anyMatch() | Terminal |
| Everything matches  Short circuit | allMatch() | Terminal |
| Nothing matches  Short circuit | noneMatch | Terminal |
| Find any  Short circuit | findAny() | Terminal |
| Find first element | findFirst() | Terminal |

Flat map combines multiple streams into a single stream

words.stream

.map(s->s.split(""))

.map(Arrays::stream)

|  |  |  |
| --- | --- | --- |
| Hello | World | Stream<String> (.stream) |
| |H|e|l|l|o| | |W|o|r|l|d| | Stream<String[]>(split) |
| |H|e|l|l|o| | |W|o|r|l|d| | Stream<Stream<String>>(Array::stream) |

**Reducing**

Reducing the stream into single value

|  |  |
| --- | --- |
| T | reduce(initial value, T BinaryOperator<A,B>) |
| Optional<T> | reduce(BinaryOperator<A,B>) |

reduce() - produces a new object or value

**A - value of previous operation**

**B- current value in stream**

IntStream , LongStream and DoubleStream are used to prevent the unboxing operations performed in case of Integer, Double or Long when doing reduce operation

mapToInt returns IntStream which has methods like min(), max(), sum() and avg()

They also define methods to generate sequence of numbers

range(1,100) - excludes 100

rangeClosed(1,100) - includes 100

**Creating infinite streams**

Stream.iterate and Stream.generate

Stream.iterate takes initial value and unary operator which is applied to new value generated by stream (Stateless)

Immutable because new values are generated

Stream.generate uses a Supplier to generate values (Stateful)

**boxed()** - Used to convert primitive stream elements to Objects

**Creating streams**

|  |  |
| --- | --- |
| **Collection** | **stream()** |
| **Arrays** | **Arrays.stream()** |
| **From numbers** | **IntStream.range()**  **IntStream.rangeClosed()**  **DoubleStream/LongStream** |
| **Infinite streams** | **Stream.iterate()**  **Stream.generate()** |
| **Simple stream** | **Stream.of()** |

**Optional**

|  |  |
| --- | --- |
| Creating empty optional | Optional.empty() |
| Create optional from value | Optional.of(value) //throws null pointer if value is null |
| Create optional containing null value | Optional.ofNullable(value) // If value is null returns empty optional |

Optional.map -> If optional contains value it will apply the function, if its empty nothing happens

Optionable is not serializable

|  |  |
| --- | --- |
| get() | Throws NoSuchElementException if no value is present |
| orElse(T) | Provide default value when empty |
| orElseGet(Supplier) | Used when default value is time consuming to create, only created when Optional is absent |
| orElseThrow | Allows to throw Exception of desired type when no value present |
| filter(Predicate<T>) | If not empty then execute the predicate |
| flatMap() | Converts Optional<Optional<t> to Optional<T> |
| ifPresent(Consumer<T>) | If value is present then consume |

Do not use primitive optional as Optional contain only 1 value it does not give any performance benefit and also they don’t contain the map, flatMap or filter methods

**Collectors**

|  |  |
| --- | --- |
| Collectors.counting() | Count of all elements in the stream |
| Collectors.maxBy(Comparator<T>) | Maximum element |
| Collectors.minBy(Comparator<T>) | Minimum element |
| Collectors.summingInt(Function<A,B>) | Sum of all elements |
| Collectors.averagingInt(Function<A,B>) | Average of all elements |
| Collectors.summarizingInt(Function<A,B>) | Returns IntSummaryStatistics which contains sum, average, max and min |
| Collectors.joining() | Calls toString method of element in the stream and concatenates them  Internally uses StringBuilder |

Collectors.reducing(starting value, Function<A,B>, BinaryOperator<A,B>)

Reorganize elements of stream into a collection

Collect vs reduce

Collect is parallel friendly, collect is a mutable operation, a collection is created and elements are added, different collections are created in different parts of the stream and are joined together

Collectors.counting()-reduce(0L,e->1L,Long::sum)

counting(), maxBy, minBy, averageInt, summingInt, summarizingInt are all special application of Collectors.reducing function

**Grouping**

Function used to group stream elements is called classification function

Collectors.groupingBy(Function<A,B>) where Function<A,B> is classification function

Collectors.groupingBy(Function<A,B>,Collector()) used to create multi level grouping

Collectors.collectingAndThen(Collector,Function<A,B>) -Apply function on value the collector returns

Collectors.mapping(Function<A,B>) apply transformation function on elements before accumulating

**Partitioning**

Special type of grouping in which classification function is a predicate

Can have only 2 keys TRUE or FALSE

Partitioning same as filtering expect you can have list of both TRUE and FALSE stream elements in partitioning

Partitioning also supports multilevel grouping similar to groupingBy

BinaryOperator<T> for a binary operator the parameter type and return type would be T

**Collectors Basics**

Consist of

1. Supplier<A> supplier() - return an empty instance of Accumulator
2. Accumulator returns the function BiConsumer<A,T> that does the reduction operation, output of the first accumulation is input to next accumulation
3. Combiner - Returns BinaryOperator<T> that combines the result of accumulators in parallel stream
4. Function<A,R> Finisher return function that is performed at the end of accumulation process

When defining own collection Characteristics(Hints) need to be set otherwise NPE

Reduce operation has supplier, accumulator and combiner so the returned Type will be the accumulated Type

Other two methods return the same type as the stream

So collector allows Accumulator of Type A, input of Type I and return type of Type R, finisher is missing in reduce operation.

The accumulator A in reduce() is a BiFunction<I,A,A> which returns type A whereas in a collector the accumulator is BiConsumer<A,I>

peek() can be used for debugging, it behaves as if the stream is consumed but allows us to view the output after each stream operation

Streams.parallel() on a sequential stream does not turn it into parallel but indicates that operation needs to be performed in parallel( boolean flag is set)

Parallel() and sequential() can be called on a stream any number of times but the last call whether parallel/sequential

decides execution of pipeline

Parallel stream uses ForkJoinPool

No of threads = No of processors

Boxing and unboxing operations increase cost of streams

Also ordered operations like findFirst() are difficult to parallelize

Spliterator - Allow iterating elements in parallel

**Default methods**

From Java8 onwards static methods are allowed on interfaces and default methods which methods with implementation

sort(),stream() method in List<T> interface

**Adding new methods to interface is binary compatible means existing classes continue to run unless they are recompiled**

Types of compatibility

Binary

Source // program continue to run even after recompile

Behavioural // running the program after change with same input results in same behaviour

Rules for resolution during multiple inheritance

1. If the class overrides the methods then class method will be called
2. If more specific interface exists (interface extending from base interface) then its methods will be called
3. If still ambiguity exists class needs to override the method and call the desired interface method using

X.super.m() where X is the interface name and m is the default method