OOP state-behaviour-identity

public static void main(String[] args)

* Since its static , may access only static classes/ methods, or need to create references with new operator

---

Strings : are real class in Java ,not an array of char,
Shortcut “double quote” = new String(“double quote”);

* Always use equals to compare strings

Default values for arrays:

* Numerical types : 0
* Object types : null

ex/ Circle [] circles = new Circle[numCircles];
circles[i].setRadius(Math.random() * 10); NullPEx

Also

for(Circle c: circles) {

c = new Circle(Math.random() * 10); // Fails to store c in

array } //array is still null objects

Objects and References
Object references are initially null.
Primitives cannot be cast to an object(ex/string)
But can be wrapped(Integer)
New operator is used to explicitly create the object

Overloading
-methods/constructors with same name
-either # of params or type of params must differ

OOP Best Practices
-instance variables must be kept private
-DRY : Don’t Repeat Yourself
-Code Reuse

- Limit Ripple Effect(changes to one class requires changes to other classes)
- SOLID

Encapsulation
Lets you change internal structure/data structures of a class without changing external access/representation.
-Doesn’t always require getter/setters

Inheritance
-Allows child to inherit characteristics of parent
-Allows access to non-private fields of parent class
*NECESSARY FOR CODE REUSE
Allows hierarchical class/code design, such that shared behaviour is inherited by classes that require it.
*super(with args) are used for non-default initializtn

OOP Design
If you want to process objects of different types , either make them inherit from same class, or implement same interface
(You cannot use the face that all objects inherit from Object class to call function implemented by different types of objects… (wont compile)

Abstract Classes
-cannot be directly instantiated
-requires a subclass
-methods marked abstract need to be implemented by a subclass to be instantiated.
+)
*Enforces behaviour, all subclasses will have certain methods
* Allows handling of collections of different types

Interfaces

- more flexible as a class can implement multiple interfaces but can inherit single class
- but downside is they cannot have mutable vars

@Override
-catches errors at compile time(ex/ against a typo)

- expresses design intent(shows that method stems from parent class)

Visibility Modifiers
public : all visible
private : only accessible from within the class
protected : access within class/subclasses, package, (used when child requires access to parents internals)
default : within class and package (rarely used)
final : variable -> const (cannot be changed)
: class -> cannot be subclassed
: method : cannot be overwritten by subclass
Synchronized : puts a lock , allows only one thread
Volatile : other threads can see changes(guarante)

- ———————————————

ENUMS
*classes with fixed number of instances(12 months)
*Easy comparison(if m==Month.DECEMBER)
*Automatic toString conversion
*Enums can have methods (is Weekend(){…})

***CLASSPATH, list of directories for classes
GENERICS
*Allows usage of List/Maps/Sets
ex/ public static <T> T lastElement(List<T> elems)
<T> tells java that this is a type, not a reference
T can be used as return type or argument type
—Generics can only be used on Objects not on primitives (autoboxing)

StringBuilder
-Strings are immutable, upon concatenation original string is copied , new String is created.O(N^2)

- StringBuilder is directly modifiable, and operation is O(N), has reverse and insert operations

---

Collections
—Lists: ArrayList : head(N), tail(amor 1), search 1,
LinkedList : head(1), tail(1), search (N),
ArrayUtils:toList()const,sort(arr,comparisonfunction)
—Map<K,V> -> HashMap
map.put(key,value) 1, map.get(key) 1, map.keys(o(N))
—Set<V> -> HashSet , add, contains

Inner Classes : used only for helping outside class
—Nested Classes :
static class B
— Member Class :
class B
— Local Class : only used in defined method
method (){ class B}
— Anonymous class

SOLID
1- Single Responsibility Principle:
Each class one responsibility, one reason to change
Book -> info | InventoryView { Book , searchBook)

2- Open Closed Principle:
Open for Extension / Closed for Modification
Discount Manager{ processBookDiscount(BookDiscount)
Interface -> getBookDiscount
CookBookDiscount implements BookDiscount
3- Liskov Substitution Principle:
A subclass can be used instead of a superclass
List <E> ~ LinkedList<E>

4- Interface Segregation Principle:
Classes shouldn’t be forced to implement unnecessary interfaces,
BookInterface -> getInfo, listenSample, search SecondHand
HardCopyInterface -> searchSecondHand
AudioBookInterface listenSample

5- Dependency Inversion Principle:
Avoids tightly coupled code,
Shelf -> add Book | to Book implements Product
-> addProduct (Where you pass a book/dvd..)

Exceptions
1) Prevent it
2) a) partial fix and normal op
b) partial fix and rethrow
c) Handle and throw different exception
3) Don’t Catch (declare throws clause)
*If no one catches till main, program will terminate with stack trace
Throwable 1)Error 2)Exception a)Runtime Exception b)…
For checked exceptions (either try-catch
Or explicitly declare it may throw exception=
*Unchecked(error/runtime exceptions) dont require throws clause

---

Threads
-extends java.lang.Thread class
-implement java.land.Runnable interface
Need to implement run() method

* 5 states , new,ready,running,blocked,finished

Thread Pools , removes overhead of creating threads (Executor object, executors class)
Race Condition
Both threads access shared resource,
Lock Interface (lock unlock newCondition)
Condition(await,signal,signalAll)
Synchronized()
Any object can be a monitor, once a thread locks it
Static -> class lock | instance object lock
Blocking Queue wrapper(put,take) Array,Linked,

Semaphore
Restrict number of threads accessing to it, wait/release

FINAL
Field Method Constructor Class Array

Lambda
– The expected type must be an interface that has exactly one (abstract) method

- Drop Interface and Method Names(Comparator)
- Drop Parameter Type Declarations(String)
- Use Expression Instead of Block
- Omit Parens When One Parameter

Find any variable or parameter that expects an interface that has one method
@FunctionalInterface
button.addActionListener(event -> handleButtonClick());

As variables (makes real type more obvious)
AutoCloseable c = () -> doSmth();
Replace this use of an anonymous inner class
doSomething((args) -> value);
Method References
if the function you want to describe already has a name, you don’t have to write a lambda for it, but can instead just use the method name.

signature of the method you refer to must match of the method in functional(SAM) interface.

Type of Method References can only be found out by context(goes for all lambdas)

SomeClass::staticMethod
someObject::instanceMethod as expects
SomeClass::instanceMethod +1
SomeClass::new Employee::new
Lambdas are lexically scoped
– They do not introduce a new level of scoping
The “this” variable refers to the outer class, not to the anonymous inner class that
the lambda is turned into
– Lambdas cannot introduce “new” variables with same name as variables in method that creates the lambda
Effectively final local variables
– Lambdas can refer to, but not modify, local variables from the surrounding method
– These variables need not be explicitly declared final as in Java 7
– This rule (cannot modify the local variables but they do not need to be declared
final) applies also to anonymous inner classes in Java 8
java.util.function defines many simple functional (SAM)
– Predicate<T> — T in, boolean out
– Function<T,R> — T in, R out
– Consumer<T> — T in, nothing (void) out
– Supplier<T> — Nothing in, T out
– BinaryOperator<T> — Two T’s in, T out
Predicate<T>
Lets you search collections for entry or entries that match a condition

public interface **Predicate<T>** { boolean test(T t);}
Generic interface **Function<T,R>** {
R apply(T t);
Lets you make a “function” that takes in a T and returns an R
– Use Function to generalize the transformation operation (salary, population, price)
public static <T> int **mapSum**(List<T> entries,Function<T, Integer> mapper)
public interface **BinaryOperator<T>** {
T apply(T t1, T t2);
– Lets you make a “function” that takes in two T’s and returns a T
Having all the values be same type makes it particularly useful for “reduce” operations that combine values from a collection.
public interface **Consumer<T>** {
void accept(T t);
Lets you make a “function” that takes in a T and does some side effect to it
Lets you do an operation (print each value, set a raise, etc.) on a collection of values
public interface **Supplier<T>** {
T get();
Lets you make a no-arg “function” that returns a T. It can do so by calling “new”, using an existing object, or anything else it wants. Lets you swap object-creation functions in and out Supplier<Employee> maker2 = () -> randomEmployee(); Employee e1 = maker1.get();
Higher order functions, functions that return functions
– You can also have a lambda that returns another

compose
– f1.compose(f2) means to first run f2, then pass the result to f1. Default method.

andThen
– f1.andThen(f2) means to first run f1, then pass the result to f2

identity
– Function.identity() creates a function whose apply method just returns the argument unchanged

transform
– Given a list and a function, returns new list by passing all the entries in the old list through the function
• Very similar to map method of Stream, which we will cover later
public static <T,R> List<R> transform(List<T> origValues, Function<T,R> transformer)

Chained Function Composition

Function<String,String> makeUpperCase = String::toUpperCase;
Function<String,String> makeExciting = word -> word + ": Wow!";
List<String> excitingUpperCaseWords = transform2(words, makeExciting, makeUpperCase);

• Difference between andThen of Consumer and of Function
– With andThen from Consumer, the argument is passed to the accept method of f1, then that same argument is passed to the accept method of f2
– With andThen from Function, the argument is passed to the apply method of f1, then the result of apply is passed to the apply method of f2

comparing
– Static method that takes function that returns a key and builds a Comparator from it
Arrays.sort(words, Comparator.comparing(String::length));
• reversed
– Default method that imposes the reverse ordering
Arrays.sort(words, Comparator.comparing(String::length) .reversed());
• thenComparing
– Default method that specifies how to break ties in the initial comparison
Arrays.sort(employees, Comparator.comparing(Employee::getLastName) .thenComparing(Employee::getFirstName));

Sorting with method that returns lambda
Arrays.sort(words, Comparator.comparing(String::length));

summary
Predicate
– Default methods: and, or, negate
– Static method: isEqual
• Function
– Default methods: andThen, compose
– Static method: identity
• Consumer
– Default method: andThen
• Comparator
– Default methods: reversed, thenComparing
– Static method: comparing
• Custom higher-order functions
– Regular method that returns lambda or Function that returns lambda

Interfaces

Standard solution
– Put abstract getArea method in the interface, define it in the classes
– Make static method that takes a Shape[] and sums the areas
• Java 8 twist
– Put static method directly in Shape instead of in a utility class as would have been done in Java 7

Default (Concrete) Methods in Interfaces
– Java 7 and earlier prohibited concrete methods in interfaces. Java 8 now allows this
Conflict Resolution
    Classes win over interfaces
public class ChildClass extends ParentClass implements Int1
– Conflict resolved: the version of someMethod from ParentClass wins over the version from Int1
    Conflicting interfaces: you must redefine
public class SomeClass implements Int1, Int2

The conflict cannot be resolved automatically, and SomeClass must give a new definition of someMethod
can refer to one of the existing methods
Interface1.super.someMethod(…)

Streams
Wrappers around data sources such as arrays or lists
Not data structures
– Streams have no storage; they carry values from a source through a pipeline of operations.
• They also never modify the underlying data structure
Designed for lambdas
Do not support indexed access
• Can easily be output as Lists or arrays
• Lazy - Parallelizable
• Can be unbounded
words.stream().map(…).filter(…).other(…);

forEach(Consumer)
Calling a Lambda on Each Element of a Stream
– employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))
forEach is a "terminal operation", which means that it consumes the elements of the Stream
CANNOT Change values of surrounding local variables
CANNOT Break out of the loop early

• **map(Function)**
Produces a new Stream that is the result of applying a Function to each element of original Stream
– ids.map(EmployeeUtils::findEmployeeById)
Stream.of(nums).map(n -> n * n)
flatMap
– Each function application produces a Stream, then the Stream elements are combined into a single Stream. For example, if company is a List of departments, this produces a Stream of all combined employees
• company.stream().flatMap(dept -> dept.employeeList().stream())

• **filter(Predicate)**
Produces a new Stream that contain only the elements of the original Stream that pass a given test (Predicate)
– employees.filter(e -> e.getSalary() > 500000)

• **findFirst()**
Returns an Optional for the first entry in the Stream. Since Streams are often results of filtering, there might not be a first entry, so the Optional could be empty
– employees.filter(…).findFirst().orElse(defaultValue)
When you know for certain that there is at least one entry use get() if unsure .orElse(otherValue)
• collect(Collectors.toList()) & toArray(ResultType[]::new)
– List<Employee> empList = employees.collect(Collectors.toList());

Intermediate methods
– These are methods that produce other Streams. These methods don't get processed until there is some terminal method called.
• Terminal methods
– After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it.
• These methods can do a side-effect (forEach) or produce a value (findFirst)
• Short-circuit methods
– These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

– **limit(n)** returns a Stream of the first n elements.
– **skip(n)** returns a Stream starting with element n (i.e., it throws away the first n elements)
– limit is a short-circuit operation

sorted
– sorted with a Comparator works just like Arrays.sort, discussed earlier
– sorted with no arguments works only if the Stream elements implement Comparable
– Sorting Streams is more flexible than sorting arrays because you can do filter and mapping operations before and/or after
min and max
– It is faster to use min and max than to sort forward or backward, then take first element
– min and max take a Comparator as an argument
• distinct
– distinct uses equals as its comparison

Sorting
• Big ideas
– The advantage of someStream.sorted(…) over Arrays.sort(…) is that with Streams you can first do operations like map, filter, limit, skip, and distinct
– Doing limit or skip after sorting does not short-circuit in the same manner as in the previous section
• Because the system does not know which are the first or last elements until after sorting
min and max are O(n), sorted is O(n log n)

allMatch, anyMatch, and noneMatch take a Predicate and return a boolean
– They stop processing once an answer can be determined
• E.g., if the first element fails the Predicate, allMatch would immediately return false and skip checking other elements
– count simply returns the number of elements
• count is a terminal operation, so you cannot first count the elements, then do a further operation on the same Stream

reduce :
Repeated combining
Reduction operations on IntStream and DoubleStream
– min(), max(), sum(), average()
– You start with a seed (identity) value, combine this value with the first entry of the Stream, combine the result with the second entry of the Stream, and so forth
• reduce is particularly useful when combined with map or filter
• Works properly with parallel streams if operator is associative and has no side effects

Paralel Reduction
reduce is the same if
– No side effects on global data are performed
– The combining operation is associative (i.e., where reordering the operations does not matter).

Infinite
Stream.generate(SUupplier)
Stream.iterate(seed, valueTransformer)
Using methods in the **Collectors** class, you can output a Stream as many types
(toList()
(joining(delimiter))
(toSet())
(toCollection(CollectionType::new))
partioningBy(...)), strm.collect(groupingBy(...)

Serialization
ObjectOutputStream
– For serializing (flattening an object)
● ObjectInputStream
– For deserializing (reconstructing an object)
should implement the Serializable interface
Its class should also provide a default constructor
Serializability is inherited
Only the object's data are preserve
transient keyword prevents the data from being serialized
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(booleanData);
oos.close();

Version Control
adding a new field
InvalidClassException
Unique Identifier
serialVersionUID
Creating Your own
Protocol via **Externalizable** interface
implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes, methods must explicitly coordinate with the supertype to save its state
● These methods supersede customized implementations of writeObject and readObject methods

If the object supports Externalizable, the writeExternal method is called
– If the object does not support Externalizable and does implement Serializable, the object is saved using ObjectOutputStream.