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

---

Differences from C++
– Methods (member functions) are the only function type
– Object is the topmost ancestor for all classes (All classes inherit from it)
– All methods use the run-time, not compile-time, types (i.e. all Java methods are like C++ virtual functions)
– The types of all objects are known at run-time
– All objects are allocated on the heap (so, always safe to return objects from methods).
• No difference between “s is a String” and “s is pointer to String”
– Single inheritance only

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

Constructor lets you assign values to objects
* Consistency : enforces objects to have certain properties
* Side Effect : lets you run extra code

This Variable
-Allows passing pointer to current object to methods
- resolves naming conflicts(this.x = x)

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

---

toString() method
- we write this method but never call it
- Converted to a string
- Or called in print function

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**

———————————————————————

Garbage Collection
Automatic process, tracks count of references to an object, tries to free after heap space is below a threshold finalize()->marks object to delete, gc() run

———————————————————————

Packages
-resolves naming conflicts
-structured grouping
-restricts access to outside, allows inside access

———————————————————————

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 uncessary 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