No:---------- Date:- OOP -----

*What is OOP?

* OOP is a Programming Paradigm, which uses objects and its interactions for building a software

* Why we use OOP ?

- OOP is the successor of procedural (structural) programming
- ⊙ makes complex software faster to develop and easier to maintain
- ⊙ enables the reuse of code by applying simple accepted rules

what is the variable ?

- ⊙ reversed place or a container to store data in it until it needed
- ⊙ takes name to facilitate handling
- ⊙ we declare its type before using it

Final vs Constant

Constant : ⊙ just like a variable
⊙ we have to declare its value (default) before using it
⊙ its value remains constant

Final : ⊙ just like a Constant
⊙ we don't have to declare its value before starting the program
⊙ when it takes a value it remains Constant

Data type

set (range) of values that have similar characteristics
(Name - size - default value)

* Operators

No:---------- Date:----------

arithmetic : used to perform math operators
(+ - % / ++ --)

assisment : allow assigning values to variables
(=, +=, -=, /=, %=, &=)

Comparison : Compare between variables (<, >, ==)

String Concatination : add two strings

Logical : &&, ||, !, ^ take bool and return bool

bitwise (binary) : operates on binary representation of numeric types

Type Conversion : size of, type of

square operator : to access element in array

* loop : repeated execution of a fragment of source code

* method : Part of the program to solve a problem which takes parameters and return value

* Recursion : programming technique where method call itself
  - direct : when method call itself in its body
  - indirect or (mutually) : A → B → C → A
  - bottom : when the solution found in the base cases directly without need of recursion

* Condition : (Temporary operator)

No:---------- Date:----------

Class: ◉ definition or specification or container of a given type of objects
◉ Pattern which describe state and behaviour of copies which created from pattern

object: a copy created from the definition of a given class (pattern) where method and var are stored

* object characteristics: difference ?

[1] State: define and describe object in general (attribute) or in a specific moment (Data member)

[2] behaviour: the specific action which can be done by the object (method) discribe change

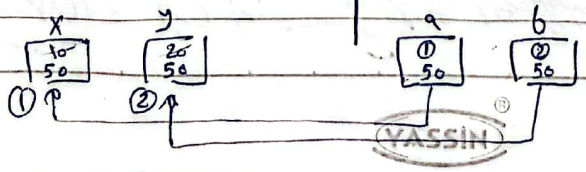
* Class Elements
- declaration: declare the name of the class
- body: encluded in curley brackets (Container)
- Fields: variables declared inside the class
- properities: describe the characteristics of the class which kept in fields
- methods: perform particular action

→ Parameter types ... ?

* Call by values vs reference
→ Actual Parameter: Parameters passed to a function
→ formal Parameters: Parameters recieved by a function
→ Call by reference
  ↳ actual & formal Parameters refers to same memory location

```
int x=10, y=20;
add(&x, &y);
```

```
void add(int* a, int* b)
{ *a = 50; *y = 50; }
```

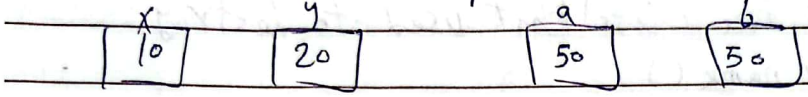


No:---------- Date:----------

Call by value

- Copy the actual parameter value to the formal parameters
- value of different parameters stored in different locations in memory

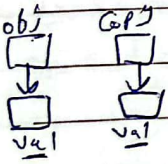
```
int x=10, y=10;          | void add (int a, int b)
add(x,y);                | { a=50; b=50; }
```



Shallow Copy vs deep Copy --?

Shallow:
- Copy obj and its reference in memory and store them
- any change in obj or copy reflects to obj
- faster than deep Copy (Compile time)
- automatically implemented by Compiler

Deep Copy:
- Copy obj and its value and stove them in different memory location
- implemented manually
- slower than shallCopy
- Change not reflect on original obj




```
class BoX{                  | BoX B1, B2;
int* X;                     | B1->X = 10;
BoX(BoX& obj)               | B2 = B1; // assignment
{ *X = *(obj.X); }          | BoX B3 = new BoX(B1);
};                          | // Copy constructor
```

Copy Constructor: intilize object from exsisting object unique Costructor to create new

No:---------- Date:----------

Constructor: a special method of the class that has its name

- Called automatically when creating new object
- Perform intialization of its data
- no Parameters → Parameterless Constructor

Decostructor

- special method of the class that used to destroy the object ~name();
- no overloading & Parameterless &
- work automatically if it isn't private

Compile time vs Run time

time where source code converted to executable code (syntax error)	time where executable code start running (logic & runtime error)

Access modifier

	Same class	child Package	child X Package	child Package X	child X Package X
Public	↘	↘	↘	↘	↘
private	~~X~~	X	X	X	X
protected	↘	↘	↘	↘	X
default	↘	↘	↘	X	X
(subclass)					

Static keyword

→ create members to be accessed without need to create object
→ members can be overloaded not override
→ useful for sharing functions ?
because all members share allocated memory space between all instances during execution that stored in heap

YASSIN

No:---------- Date:----------

Friend

- used to access private and protected members of other classes (not a member of class to be instantiated)

```
class Box{                    class F
private:                      {public:
int pri;                      void display(Box& b)
protected:                    { cout << b.pri << " " << b.pro;
int pro;                      }
friend class F;               }
}
```

Namespace

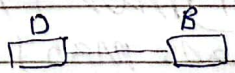
- declarative region or scope or container for the identifier (var, func)
- in nested namespace inside for first

Inheritance

- deriving properities and characteristics from class to another (parent-child relationship)
- superclass where properities inherited
- subclass who inherits from base class
- why inheritance ? to avoid duplicating code & reuseability
- create new class based on existing class

Types of Inheritance

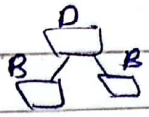
① single
- derived class inherits from single base class



② Hierarchal
- more than drived class inherits single base class

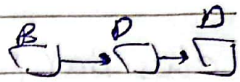
YASSIN

No:---------- Date:----------



③ multiple
↳ derived class inherits more than one base class

④ multilevel
↳ derived class inherits a derived class



⑤ Hybrid (virtual)
↳ Combine more than one type of inheritance
↳ multiple must be used

- Is-A relationship
  ↳ refers to a relationship between subclass and superclass or interface

- extend keyword : relation ship between two classes
- implement keyword : relationship between class & interface

- multiple doesn't supported to reduce complexity, Confusion ambiguity غموض

- static & final make class or properity :
  Can't be instantiated

- A base = new B();
  ↳ base class inherits methods only not Properities

---

* Polymerphism

→ means many forms
→ single entity (object) shows multiple behaviour

- Types

→ static : (Compile time)
→ Dynamic (runtime)

No:---------- Date:----------

* why Polymerphism?

- Perform many operations (various) using methods with same name
- implementation to abstract class or interface
- add dynamic behaviour to the code

* Polymerphism vs Inheritance --?

take advantage of relationship to make code more dynamic	presents parent-child relationship between two classes • reuseability • avoid redundancy of code

* Compile time vs Run time

→ Compile time where the source code converts to executable code (binary code) // syntax error ;

→ Run time where the executable code starts running
// logical error divided by zero

* Binding ---?

→ linking between method call & method defination

→ static (early Binding)

- happens and resolved at compiled time by compiler
- called early as it takes place before program runs
- Compiler check method defination & referance
- variable to call not v.v
- compiler check type of object not is pointing to it

87, 88, 82, 16, 26, 60, 63, 67, 69

No:---------- Date:----------

ابتدائي

- Premitive or non Primitive object
  - int x = 10; قيمة ابتدائية
- final, static, private methods are examples overloading

* Dymaic Binding
  - resolved based on type of object at runtime (object)
  - Called late as it occurs during runtime
  - Type of object, can't be determined by Compiler
  - override is example Animal a = new Dog();
  - Can't implemented by data members

*why static & final methods are static binding ?
→ because the Compiler knows that these methods cannot be overriden and accessed by reference variable.

→ Compiler differentiate between methods having same name by signatures
→ object Pass by value or reference based on lang used
→ objects Passed by reference
→ object Passed by value in function intialized and lost in its scope
→ two classes Can have same name if in different namespaces

* overloading
methods have the same name with different parameters or return type

* overriding (virtual method)
custom implementation of an inherited member

No:---------- Date:----------

* Pure virtual method
like a virtual method, but it has no definition in the base class.

* Class (Concrete)	abstract class	Interface
Container of a given type of objects	a class can't be instantiated	• a class to achieve complete abstraction
class inherits more than one	class inherits only one	• class implements more than one • Can extend interfaces
doesn't have pure virtual method	Contains code or data	• Contains only methods with no definition
always has a body	• Can have constructor assign access modifiers to members	• Can define class inside it
	• use multi level inheritance	• define static or final fields in it only
		• Can't have constructor

Encapculation
- hiding data and its implementation inside class
- Implements desired level of abstraction
- Constructor executed when object created by new keyword
- Parameter's constructor not equivalent to default Constructor
- Setter & getter Permit different access level
  - provides debugging point for when a Property changes at runtime

YASSIN

No:---------- Date:----------

* Association
→ relationship where objects have their own life and no owner person —— flat

* Aggregation
→ relationship where objects have their own life, but their is ownership
→ child can't belong to another parent
→ if parent not exisit child can exsist Book → lib

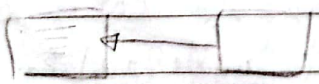
* Composition
→ whole/part relationship
→ object composed to another ones
→ object is part of whole hand → body

→ in multilevel first class represents highest level of abstraction

→ Code duplication is insidious because one has to maintain all duplicates

→ static Constructor when object created and only one time used to intialize all members with static value

→ CRC Cards ? brainstorming tool used in design oop
Contains (object, responsibility, Collaborations)



→ Abstractions makes relevent information and encapsulation implents level of desired abstraction

No:---------- Date:----------

→ why unit testing harder than functional programming?

objects may maintain internal state which hard to test

→ function of a user diagram?

link the actor to the role played in all use cases

→ open/closed principles open for extension and closed for modification

→ aggregation described as a collection of objects

→ we need to return object when it passed by reference because it automatically reflected in the calling function

→ parameter is the variable declared in function, but argument the value of the variable

→ what is finalize method ---?

it used in java to clean up code when object collect grabage

not recomended as it leads to resource leak and inefficient memory. use (AutoClosable with try & Catch)

Public class Book implements AutoClosable {

@override

protected void finalize() throws Throwable { super.finalize(); }

@override

public void close() throws EXCEPTION { }

No:---------- Date:----------

SoLD Principles

- each class module in your codebase should have specific purpose
- methods & properities should be related to this purpose
- makes code understanble, maintainble, efficient, productive

[1] Single Responsibility Principle [SRP]

- a class should have only one responsibility
- easier to test & maintain

```
class User {
string email ;
string password ;
User ( email, Password ) }
```

```
class EmailSender {
emailsender ( string body ) ;
}
```

[2] open / closed P [OCP]

- class should open to extension but closed to modification

```
abstract class Payment {
void PaymentProcess ( ) ;
}
```

```
class credit implement Payment { }
class debit implement Payment { }
```

No:---------- Date:----------

(3) Liskov substitution p [LSP]

- subclass could replace the parent class without affect on the correctness of the program
- subclass implement all parent methods and acts like a parent

```
class Parent {
void go() { };
}
```

```
class child extends Parent
{ void go() { }   // child
  void goParent(Parent p)
    { p.go(); }   // Parent
}
```

(4) Interface Sergregation [ISP]

- client shouldn't be forced to implement methods he doen't use

```
abstract class printer{}
abstract class scanner{}

class multi implements printer, scanner { }

class user_print extends multi {}
```

No:---------- Date:----------

[5] Dependancy Inversion [DIP]

- high level module doesn't depend on low-level module

abstract class Payment {
void pay() {};
}

class credit Implement Payment
class debit Implement Payment

class Paymentprocess {
Payment _p;
void payway (this._p) { _p.pay(); }

Test Driven Development

- why? → to ensure app performs correctly & fix BUGS
- Types
  - Unit: Function or class
  - widget: test single widget
  - Integration: for all app or a part of it

	unit	widget	integration
confidence	low	low	high
dependancies			
maintainance			
execution →	high	high	low

- structure of a good test
  1. setup → create instance of object
  2. side effects → collect result you want to test
  3. expectation → compare result assigned with expected valve

YASSIN

~~8, 11, 13, 17, 18, 22, 23, 24, 29, 31, 32, 33, 34, 37, 38~~
~~39, 41, 42, 44, 47~~

* What's manipulators ?
provides a way to control the apperance of input or output in flexible manner
- insertion → to format data output
// Cout << setprecision(3) << 3.1425 << endl;
- extraction → to format data input
// cin >> x;

* What's inline function ?
technique used by Compiler to insert complete body of function wherever function is used in source code
// void printer () ⇒ print (" ");

* what is operator overloading ?
- used to give a specific meaning when applied to object of your class

1. Arithmetic operator (+, -, *, /)
2. Comparison operator (==, !=, <, >)
3. stream operator (input/output)

```
class Book {
private:
string title;
string author;
```

No:---------- Date:----------

public:
Book operator+ (Book& b2) { (1)
Book b1;
b1.title = b2.title
b1.author = b2.author
return b1;
}

bool operator== (Book& b2) { (2)
Book b1;
retun (b1.author == b2.author);
}

friend ostream operator<< (ostream& os, Book& b)
{
os << "title:" << b.title << "author: " << b.author << endl;
return os;
}

* what's Exception handling ?

an event occurs during execution of program
(try, catch, throw) keywords

* What's tokens ?
- identifiers, constants, strings, keywords operators
- Complier recognize can't breakdown to elements

No:---------- Date:----------

* keyword to describe overloading ?
  - operator

* Define sub, base, super class ... ?

- super → parent class where classes inherits from it
- base → it refers to root class
- sub → child class which inherits from parent class

* virtual vs pure virtual fun ... ?

- virtual → method can be overriden in child class and can be defined in parent

- pure virtual → method can be overriden in child but can't be defined in parent

```
// void printer ();      virtual
// void printer () = 0;  pure virtual
```

* operators can't be overloaded ... ?

① Scope Resolution (::)
② Member selection (.)
③ member selection through pointer (.*)

structure vs class ... ?

access	structure	class
contains	• public • data • used for data	• private • data & methods • OOP principles

No:---------- Date:----------

* what's "this" ___ ?

keyword refers to current object of a class

* Types of Constructor ___ ?

- Default → with no parameter
- parameteric → create new instance, passing arguments
- Copy → create a copy of exisiting object

* New (vs) override ___ ?

- new → create implementation of base class function
- override → rewrite of base class function in drived

* what's sealed modifier ___ ?

- keyword used in Java, C# to prevent drived classes from override the method inherited

```
class Book{
public virtual void mymethod(){}
}
class Libary : Book{
public sealed override void mymethod(){ }
}
```

YASSIN