19-20 (Mid)

① ~~Disti~~ Object Vs Class:

Aspect	class	Object
Definition	Blueprint or template	Instance of a class
Memory	Does not occupy memory	Occupies memory
Type	Logical entity	Physical entity
Usage	Used to define structure	Used to perform actions/data.
Example	Car	myCar - model, color, brand etc

```
class Car {
public:
    String color;
    Void drive() {
    cout << "Driving.." << endl;
    }
};
```

obj

```
Car myCar;
myCar.color = "Red";
myCar.drive();
```

Data Abstraction and data Encapsulation:

Feature	Data Abstraction	Data Encapsulation (binds the data and method into a single Unit)
Purpose	Hides implementation details	protects data from Unauthorized access
Focus on	What an object does	How data is secured and accessed
Implementation	Using Abstract classes	Using access specifiers in classes
Example	Tv Remote	ATM card
Goad	Simplicity	Security.

```
class Car {
public:
    void startEngine();
};
```
only show What todo, not how

```
class BankAccount {
private:
    int balance // hidden data
public:
    void deposit(int Amount);
    // Access through method.
```

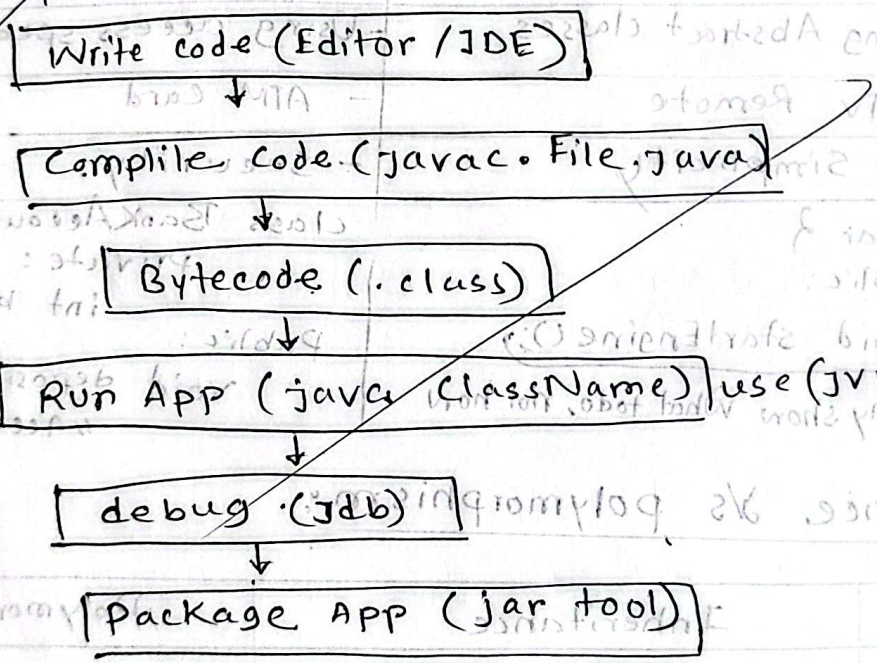
Inheritance Vs polymorphism:

Feature	Inheritance	Polymorphism
Meaning	One class inherits another	One function behaves differently
purpose	Code reusability	Flexibility in behavior
Type	Relationship between classes	Same interface, different behavior
Example	dog inherits from Animal	speak() behaves differently for Dog
Keyword	: (colon in c++)	Virtual, Override (for runtime).
	Can be single, hybrid, mutiple	Can be compile time & run time

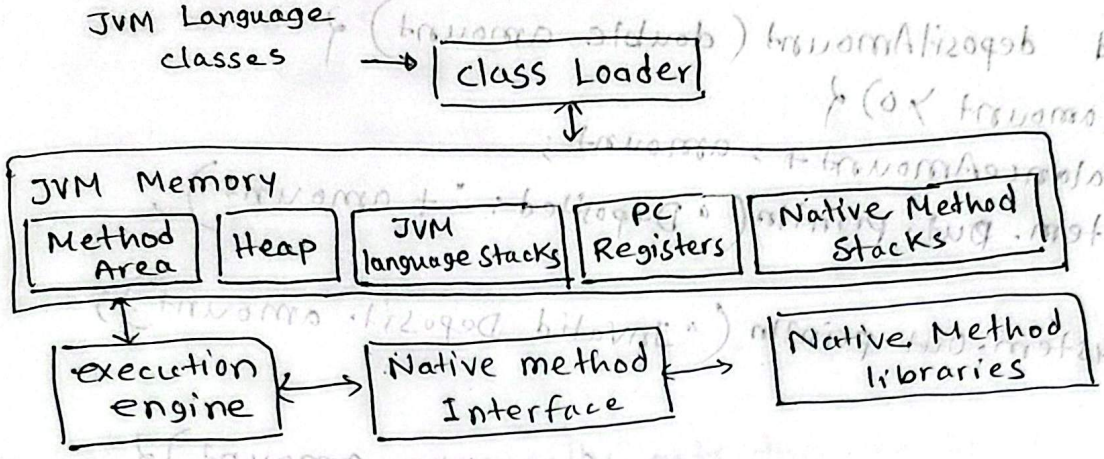
• Dynamic Binding vs Message passing:

Feature	Dynamic Binding	Message passing
Decided at	Runtime	Compile-time or Runtime
purpose	choose correct function during execution	Object Communication
Related to	Polymorphism	Encapsulation and Interaction
Example	Virtual function overriding	Object.method();
used for	flexibility in method resolution	Sending commands to objects.

(2) • Editor — for writing code
• Java compiler - compile .java file to .class file.
• Java Virtual Machine (JVM) - runs the compiled bytecode.
• Debugger - helps find and fix bugs.
• jar Tool - packages app into .jar file for deployment.



(3) JVM: Java Virtual Machine is a engine that provides runtime environment to drive the java code or application. It converts java bytecode into machines language. JVM is a part of java Runtime Environment (JRE).



(2)

Source code creation (IDE)
↓
Compilation (Javac)
↓
packaging (JAR)
↓
Execution (JVM)
↓
Debugging & testing (IDE)
↓
Deployment. (Application server)

Requirement Analysis
↓
Design Phase
↓
Development
↓
Testing
↓
Version Control
↓
Continuous Deployment
↓
Code Analysis
↓
Security Scaming → Documentation

Maintenance.
↑
FeedbackLoop
↑
monitoring
↑
Deployment
↑
performance profilling
↑
Bug Tracking
↑
Documentation

(4)

```
public class BankAccount {
    private String nameOfDepositor;
    private String accountNumber;
    private String accountType;
    private double balanceAmount;

public void assignInitialValues(String nm, String num, String
                                type, double blance){
    nameOfDepositor = nm;
    accountNumber = num;
    accountType = type;
    balanceAmount = blance;
}
```

```
public void depositAmount (double amount) {
    if (amount > 0) {
        balanceAmount += amount;
        System.out.println("Deposited: " + amount);
    } else {
        System.out.println("Invalid Deposit amount");
    }
}

public void withdrawAmount (double amount) {
    if (amount > 0 && amount <= balanceAmount) {
        balanceAmount -= amount;
        System.out.println("withdraw: " + amount);
    } else {
        System.out.println("Insufficient balance or invalid
                            amount");
    }
}

public void display () {
    System.out.println("Name: " + nameOfDepositor);
    System.out.println("Balance: " + balanceAmount);
}
```

(5) Inheritance is one in which a new class is created that inherit the features from the already existing class. It helps us create new class quickly and efficiently by allowing us to reuse and extend the functionality of existing classes. It provides code reusability, a subclass contains all the features of super class. So we should create the object of sub class.

[code reusablity, Extensiblity, Reduced Redundancy]

```
class Animal {
    void eat () {
        System.out.println ("Eating.....");
    }
}
class Dog extends Animal {
    void bark () {
        System.out.println ("Barking.....");
    }
}
```

Now, the dog class automically gets the eat() method from Animal. So you don't need to write it again.

(20-21) Mid

(2) Writing a java application basic steps are —

- Install java: Ensure we have Java Development Kit (JDK) installed on our computer.
- Write code: Use a text editor to write code.
- Compile: use the java compiler to compile our source code to bytecode.
- Run: Use the java virtual Machine (JVM) to execute our bytecode.
- Debug and test: Debug and test our application to identify errors.
- Build and package: If needed package our application to jar file.
- Documentation: Document our code and provide comments to make it understand.
- Version control: Use version control system like gi

(3) (a). Answer: ERROR

Explanation: (break) can only be used inside a loop (for, while, do-while) or switch. Here it is used inside if only, so compilation error occurs.

(b). Answer: something else

Explanation: characters are added as ASCII values, not concatenated.

'J' + 'a' + 'v' + 'a' = 74 + 97 + 118 + 97 = 386.

(c). Answer: Nothing.

Explanation: $_ is a valid variable name in Java. So, the program compiles and runs but gives no output.

(d). num1 == num2; num3 != num4.

Explanation: Java caches Integer values between -128 to 127.

num1 == num2 → both are 100 → true.

num3 == num4 → both are 500 (outside rang) → false.

(5)

```
pubic class Student {

    private String name;
    private String dept;
    private String session;

public void assignDetails(String n, String d, String s){
        name = n;
        dept = d;
        session = s;
    }
public void displayDetails(){
        System.out.println("Name : " + name);
        System.out.println("Dept : " + dept);
        System.out.println("Session :" + session);
    }
}
```

(2021 – 22) mid term-1 :

(2) i) 1 3 5 7 9 [Loop runs from i: 0 to i<10; if i is even (i%2==0), it skips printing (continue). so only odd numbers are printed]

ii) Output : 0 ;

- Loop starts with i=0 ; First iteration : immediately breaks . then prints the current value of i , which is still 0 ;

(5) ~~x~~. if format: if (condition) {
// code to execute if condition is true.
}

ex: int marks = 85 ;
if (marks > 80) {
System.out.println ("Excellent !");
}

~~x~~. if-else: if (condition) {
// code if true
} else {
// code if false
}

Ex: int num = 5 ;
if (num%2 == 0) {
System.out.println ("Even");
} else {
System.out.println ("Odd");
}

x. If-else-if: if (condition1) {
// code.
} else if (condition2) {
// code
} else {
// default code
}

Ex: int marks = 75 ;
if (marks >= 90) {
System.out.println ("Grade A");
} else if (marks >= 80) {
System.out.println ("Grade B");
} else if (marks >= 70) {
System.out.println ("Grade C");
} else {
System.out.println ("Grade F");
}

Switch:

```
switch (expression) {
    case value1 : //code
                  break;
    case value2 : //code
                  break;
    ...
    default :
                  //default code
}
```

Ex:

```
int day = 3;
switch (day) {
    case 1 :
        System.out.println ("Sunday");
        break;
    case 2 :
        System.out.println ("Monday");
        break;
    default :
        System.out.println ("Invalid day");
}
```

2018-19 : mid term-II

(1)

```
class Staff {
        int code;
        String name;
}
class Teacher extend staff {
        String subject;
        String publication;
}
class Typist extend staff {
        int speed;
}
class officer extend staff {
        int gread;
}
class regular extend typist { }
class casual extend typist {
        bool daily;
        int wage;
}
```

(2) describe the various forms of implementing interfaces with example:

(*) class Implements Interface Directly:

```
interface MyInterface {
    void myMethod();
}
class Myclass implements MyInterface {
    public void myMethod() {}
}
```

(*) Anonymous class Implements Interface:

```
Interface myInterface {
    void myMethod();
}
class main {
    public static void main (String[] args) {
    MyInterface obj = new myInterface() {
        public void myMethod() {}
    }
}
```

(*) Lambda Expression Implements Functional Interface

```
Interface myInterface {
    void myMethod();
}
class Main {
    public static void main (String[] args) {
        MyInterface obj = () -> {
        };
    }
}
```

(*) multiple interface:

```
Interface A {
    void A();
}
interface B {
    void B();
}
Class C implements A,B {
    publid void A() {
        S.O.P("A");
    }
    public void B() {
        S.O.P("B");
    }
}
```

2021-22 : mid-2

(1) Java's Wrapper classes (like Integer, Double, Boolean) are used to wrap primitive data types (int, double, boolean) into objects.

Main ~~pup~~ purpose :

- Object features:

primitive are not objects. So, they can not be used in collections like ArrayList, HashMap - Wrapper classes allow primitives to be treated as object.

- Utility methods.

Wrapper classes pororide many useful methods.

```
int a = 10;
String s = Integer.toString(a);
```

- Null values:

You can assign null to wrapper objects but not to primitives. This is useful for optional values.

- Auto boxing / Unboxing:

Java can automatically convert between primitives and wrapper Objects.

```
Integer x = 5; // autoboxing : int to Integer
int y = x; // Unboxing: Integer to int.
```

- Type Conversion and passing:

Wrapper classes can convert Strings to numbers.

```
int num = Integer.parseInt("123");
```

(4) Abstract class:

An abstract class in java is a class that can not be instantiated (you can not create object of it). It can have abstract methods (without body) and Concrete methods (with body). It is used to provide a base for subclass.

Ex:
```
Abstract class Animal {
    abstract void sound(); // abstract method
    void sleep() { // concrete method
        System.out.println("Sleeping.....");
    }
}
class dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

Static member:

A static member (variable or method) belongs to the class, not to any specific object. It can be accessed without creating an object of the class.

Ex:
```
class MathUtil {
    static int count = 0;
    static void show() {
        System.out.println("Static method");
    }
}
public class Main {
    public static void main(String[] args) {
        MathUtil.show();
        System.out.println(MathUtil.count);
    }
}
```

(5) In java, a package is a way to organize related classes and interfaces into a single namespace. A package can contain sub-packages, which further organize the code into a hierarchical structure.

To create a package in java, we include a package statement at the begging of our java source file to use a package use the import statement.

Ex: creating package:

```
package mypackage;
Public class Animal {
    public void sound(){
    System.out.println("Animal sound");
}
```

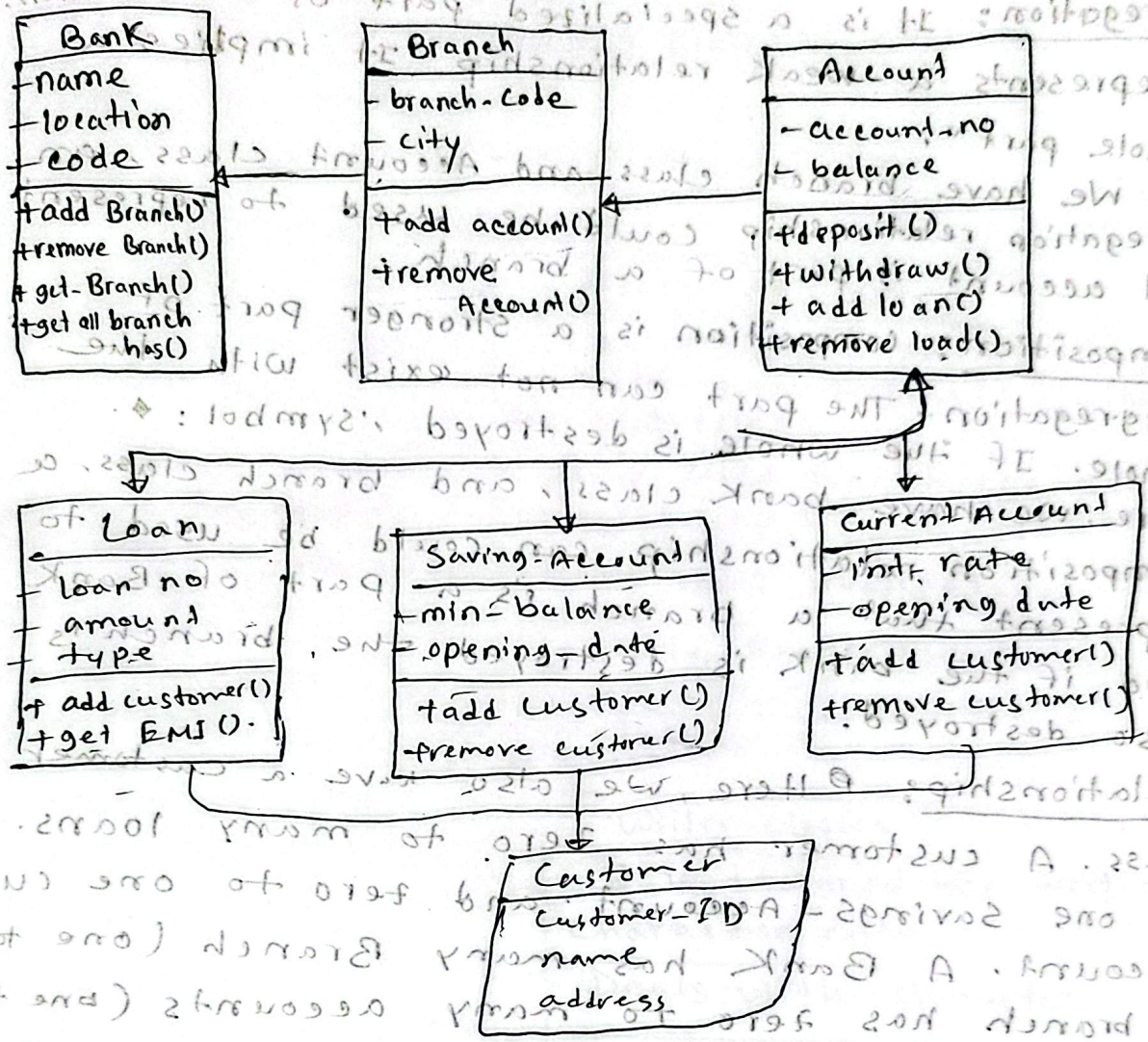
using the package:

```
import mypackage.Animal;
public class Main {
    public static void main(String[].args){
    Animal a = new Animal();
    a.sound();
}
```

2018 – 19 : mid-1.

(2) Java is often known a "platform-neutral" or "platform independent language" due to its ability to run on various hardware and operating system without modification. It can run on many different types of computer and doesn't care which one it's on. When we compile a java program it doesn't compile directly to the machine code for a specific platform. It compiles to an intermediate form called bytecode. The bytecode is platform independent. we can write our java code once and run it on any platform that has a JVM. Without modification java platform independence is also related with its security features. This Contributes to the ability to run java applications safely across different platforms.

(2) Banking Application:



• Inheritance: It represents a relationship. Here in Banking sylte account is a class and we create new classes loan, savings account, current_account that inherits from account. It means that loan, Saving-account and Current-account are types of account and they inherit the properties and method of the Account class.

• Association: It is a general term that represents a connection between classes. Here, bank, branch and account class might be associated with each other through association to represent the relationship among them.

- **Aggregation:** It is a specialized part of association. It represents a "weak relationship". It implies a "whole part".
Here we have branch class and Account class, an aggregation relationship could be used to represent that account is part of a branch.
- **Composition:** Composition is a stronger part of aggregation. The part can not exist with the whole. If the whole is destroyed, symbol: ◆.
here, we have bank class, and branch class. a composition relationship can could be used to represent that a branch is a part of Bank. and if the bank is destroyed the branch is also destroyed!
- **Relationship:** Here, we also have a customer class. A customer has zero to many loans. zero to one savings-Account; and zero to one current account. A Bank has many Branch (one to many) A branch has zero to many accounts (one to many)

2019-20: mid II

(2) * Arithmatic exception:

```
int result = 5/0;
```

* Array Index out of bounds exception.

```
int[] arr = {1,2,3};
int value = arr[5];
```

* Number format exception:

```
String str = "abc";
int number = Integer.parseInt(str);
```

* File not found exception:

```
File file = new File("nonexistent.txt");
```

* class NOT Found:
Class.forName("Non existent class");

* Interrupted excepted:
Handling thread interruption.

(3) * Input stream Vs Reader classes:

Input stream	Reader classes
- It is designed for reading row binary data.	- It is designed for reading character data, making it suitable for text files.
- It is typically used for byte-oriented input.	- It is used for character-oriented input.
- Low level abstraction	- High level Abstraction
Ex: File input Stream	Ex: File Reader

* Output stream Vs Writer class:

Output Stream	Writer classes
- It is designed for writing row binary data.	- It is designed for writing character data.
- deals with bytes	- Deals with character.
- Ex: File output stream	Ex: File Writer.

(4) condition 1: the class must implement the Comparable<T> inter
- The interface defines the natural ordering of the objects.
- The class must override the compareTo(T o) method.
- This method returns:
  - A negative value if this < 0.
  - zero if this == 0
  - A positive value if this > 0.

Ex:
```
class Student implements Comparable<Student>{
    String name;
    int roll;
public Student(String name, int roll){
    this.name = name
    this.roll = roll;
}
```

```
public in compareTo (student s) {
    return this.roll - s.roll; //sorting by roll in ascending order
} }
```

- **Condition 2:** The `compareTo()` method must be consistent with equals():
  - if two objects are "equal" then compareTo() should return 0;

(5) Threads allows a program to operate more efficiently by doing multiple things at the same time.
Threads can be used to perform complicated tasks in the background without interputing the main program.
Thread can be implement by two ways -
  - By inheritance
  - By interface.

- **Thread by inheritance:**

```
package threaddemo;
public class Threaddemo {
    public static void main (String[] args) {
        A objA = new A();
        B objB = new B();
        Obj A.start();
        Obj B.start();
    }
}
Class A extends Thread {
    public void run() {
for (int i=0; i<10; i++) {
System.out.println ("Hi");
} }
}
```

- Handling multiple clients requests.
- Asyneronous operations.
- Resource management.
- Improved performance.

2020-21 : Midii

(1) class Account {
protected String customerName;
protected int accountNumber;
protected String accountType;
protected double balance;

public Account (String customerName, int accountNumber,
String accountType, double balance) {

this.customerName = customerName;
this.accountNumber = accountNumber;
this.accountType = accountType;
this.balance = balance;
}

public void depAmount (double amount) {
balance += amount;
}

public void showBalance () {
System.out.println ("current Balance " + balance
}
}

* Derived class : SavingsAccount;

class SavingAccount (String customerName, int accountNum
double balance) {
super (customerName, accountNumber, "Savings", balance
}

public void computeInterest () {
double interest = balance * 0.05;
balance += interest;
System.out.println ("Interest added : " + interes
}

```
public void withdraw(double amount){
    if (amount <= balance) {
        balance -= amount;
    } else {
        System.out.println("Insufficient balance");
    }
}
```

(A) Derived class: CurrentAccount

```
class currentAccount extends Account {
public currentAccount(String customerName, int accountNumber, double balance) {
    super(customerName, accountNumber, "current", balance);
}
public void withdraw(double amount) {
        double minBalance = 1000;
    if (balance - amount < minBalance) {
    System.out.println("Cannot withdraw. Minimum balance (BDT 1000) must be maintained.");
    } else {
        balance -= amount;
    }
}
public void checkMinBalance() {
        if (balance < 1000) {
    System.out.println("Waring: Balance below minimum. A penalty will be imposed");
    }
}
```

(2) • Abstract class: An abstract class is a class that can not be instantiated (can not create object from it).

Ex:
```
abstract class Animal {
    abstract void makeSound();
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

• final class: A final class is a class that cannot be extended / inherited.

Ex:
```
final class Car {
    void drive() {
        System.out.println("Car is running.");
    }
}
```

• Static member: A static member belongs to the class, not to objects. Shared by all objects.

Ex:
```
static void changeCollege(String name) {
    college = name;
}
```
Uses: Student1.changeCollege("BUET");

• Java package: A package is a group of related classes. It helps to organize code and avoid name conflicts.

• built in package Ex: import java.util.Scanner;

• User define package example:
```
package myPack;
public class Student { }
```

- **Interface:** An interface is a collection of abstract methods. It is used to achieve abstraction and multiple inheritance in Java.

Ex:
```
interface Animal {
    void sound();
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

2018-19 : MID III

2021-22 : Final

II (a) Method override vs Method overloading:

Overloading	Overriding
- Method overloading is a compile time polymorphism	- Method overriding is a run time polymorphism
- It is occured within the class	- It is performed in two classes with inheritance
- Method Signature is different	- Method Signature + return type is same
- May or may not need inheritance	- Must need inheritance
- A user can easily overload it	- It is not possible for a user to override it
- Checked at compile line	- Checked at run time
- It is also known as the early binding	It is also known as the late binding

Method overloading: Method overloading is a compile time polymorphism. A method called "print" that can accept an integer, a double or a string.