Chapter 8 Important Notes : Serious polymorphism

- There is note before starting when declaring an object the compiler is check about the reference variable type and in the runtime the JVM check about the object type
- There is some classes just should not be instantiated like the animal example we saw in the last chapter it is logical to define an object from cat type or dog type but what about defining an object from animal class? There is no logic here cause there is missing information about what is the type of this animal there is some characteristics missing so how we can deal with the animal class to prevent the people from making an object from animal class cause we need ti make objects but from the less abstraction subclasses of class animal we do not want from the animal class itself objects and the solution is to define the animal class itself as ABSTRACT class and if we make this the complier will stop any code will try to define or create an object from animal class but you will still can use the animal class as a reference type but for its subclasses objects not for creating an object from its class
- So in the beginning of designing your classes you first will need to decide which classes will be concrete and which ones will be abstract ones and the difference is that the concrete classes you can create objects from it and the abstract classes is the opposite thing
- So you can always assign a subclass object to a super class refence even if the superclass is abstract
- The abstract class has virtually no use , no value , no purpose in life unless it is extended your subclasses objects which making the work at runtime
- Besides the classes the methods also can be abstracted and if you have an abstract class it is must means the class must be extended and the abstract method means this method it must be overridden and the abstract method will have no body you just mark it it will be overridden for all the subclasses will inherit this class and why it will nit have body in the super class that's cause of the code here will not make any sense cause there is more details missing to can define and write the code of this method define its behavior
- If you have declare an abstract method you must mark the class as an abstract class cause you can not have an abstract method inside concrete class cause the abstract method need subclasses to override it and if you define its class as a concrete one that will give you an error cause concrete

class means you can define an object from it and this object can freely use any method in this class cause the concrete classes means all its methods is complete from the body and implementation and there is no any missing methods and if you make the method abstract and the class itself concrete the error comes cause there is method in the concrete still have missing details which will be wrong thing.

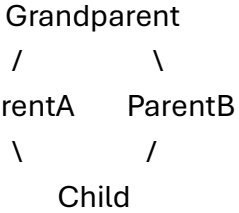
- You can mix the methods by putting abstract methods and non abstract ones inside the abstract class
- The point of an abstract method is that even though you haven't put in any actual method code, you've still defined part of the protocol for a group of subtypes (subclasses). It is like role all the subclasses must follow it to can be useful in something like polymorphism like we say in the previous chapter , So with an abstract method, you're saying, “All subtypes of this type have THIS method” for the benefit of polymorphism
- You can not make a new instance of an abstract type but you can make an array object declared o hold that type;
- Every class in java extends class object so the class object is the mother of all classes it is the superclass of everything and the object class is not abstract class cause It has methods implementation code that all classes can inherit it without needing to override them
- Some of the object class methods you can override them and some you can not cause they are defines as final methods
- Class object has 2 main benefits very important ones the first thing is that it make the polymorphism arguments very easy cause if you pass object from class object that will make your array list having different types of the classes objects and the second thing is that it defines the most needed methods any class will need it so it makes this methods ready to prevent the classes redefine them from the start
- The compiler decides whether you can call this method or no based on the reference type not based on the actual object type and that by going to the reference type class and check if this class having a method called in this name or no if yes continue if no give an error
- What about if we make thing like that one ;
  Snowboard S = new Snowboard () ;

  Object o =s ;
  Wha this code means?

Think of a variable as a remote control for an object. A **Snowboard** reference is like a remote with all the buttons for a Snowboard — it can call every method in the Snowboard class, plus all the methods it inherits from Object. An **Object** reference, on the other hand, is like a remote with fewer buttons. Even if it's controlling a real Snowboard object, it can only call the methods defined in the Object class. The type of the reference determines how many "buttons" (methods) you can use.
And you can imagine any object like it hasing inside it an object class object inside it as a center of its object so when you make a cat object the core of this object you will find it as an object class object inside it

- So now you've seen how much Java cares about the methods in the class of the reference variable. You can call a method on an object only if the class of the reference variable has that method
- Just remember that the compiler checks the class of the reference variable, not the class of the actual object at the other end of the reference (JVM which is checks about it).
- We will talk now about the multiple inheritance and that when we have class extends from 2 super classes and this is really bad thing (first thing the multiple inheritance not allowed in java) but what if it is existed it lead to deadly diamond of death and that cause imagine you have the same method was overridden in the both superclasses so when you extends the both which super class method you will use ? java is supposed to be simple, with consistent rules that don't blow up under some scenarios. So Java (unlike C++) protects you from having to think about the Deadly Diamond of Death. But that brings us back to the original problem! The problem is what we have 2 different things classes and we need class to inherit both of them? We know now the multiple inheritance not allowed cause it leads to the deadly diamond of death so the solution here is the INTERFACE
  This is an example for explain more and more the deadly diamond of death :

  →Imagine Java *did* allow multiple inheritance (like C++ does).
  You have a class that inherits from **two parents** that both inherit from the same grandparent, and both parents override the same method differently.



And this is the code but in c++ cause in java it will not be complied cause the multiple inheritance not allowed

```
class Grandparent {
  void sayHello() {
    System.out.println("Hello from Grandparent");
  }
}

class ParentA extends Grandparent {
  @Override
  void sayHello() {
    System.out.println("Hello from ParentA");
  }
}

class ParentB extends Grandparent {
  @Override
  void sayHello() {
    System.out.println("Hello from ParentB");
  }
}

// ❌ This would NOT compile in Java (multiple inheritance not allowed)
class Child extends ParentA, ParentB {
}

public class TestDiamond {
  public static void main(String[] args) {
    Child c = new Child();
    c.sayHello(); // ❓ Which one? ParentA's or ParentB's?
  }
}
```

The Problem

- Both **ParentA** and **ParentB** have their own sayHello() method.
- The **Child** class inherits from *both*, so there's ambiguity:
    - Should it use **ParentA**'s version?
    - Or **ParentB**'s version?

- This is the **Deadly Diamond of Death** — confusion and potential bugs.

Why Java avoids this
Java says:
"No multiple inheritance for classes."
Instead, if you want to share behavior from multiple sources, you use **interfaces** — which don't carry conflicting method code, only method *signatures*.

- So here comes the interface solution which solving the multiple inheritance problem in java that cause the deadly diamond of death (DDD) and the interface rule is very simple that is to make all the methods abstract and java interface is like a 100 % pure abstract class and that cause all the methods in the interface must be abstract besides the abstract class that can have both methods types abstract and not abstract
- When we define the interface we just use the keyword "interface" rather than class and when we use it we make the class implement it rather than extend it and in the interface the one class can implement multiple interfaces rather than happened in the multiple classes inheritance
- Interface methods are implicitly public and abstract that is their default definition
- Classes from different inheritance trees can implement the same interface
- There is very important benefit of the interfaces that the inherited classes doesnot offer it :
  When you use a **class** as a polymorphic type (like Animal[] or a method that takes a Canine parameter), the objects you can pass **must** be from the **same inheritance tree**, and they must be subclasses of the type you specified.
  Example:
  -A Canine reference can hold a **Wolf** or a **Dog** ☑
  -But it can't hold a **Cat** or a **Hippo** ❌ because those are from different inheritance trees.

When you use an **interface** as a polymorphic type (like Pet[]), it's different:
-Objects can come from **any inheritance tree**, as long as they implement the interface.
-This is very powerful in the Java API, because it lets you link unrelated classes through a shared role or capability.

- in java the class can have only one parent and that parent class defines who you are but you can implement multiple interfaces and those interfaces define roles you can play
- there is a very important question is that how I can know whether to make this thing as a class or subclass , abstract or concrete , interface ?
  the answer in this steps:
    - make it a class if it does not extends anything that means it doesnot pass the IS A test for any other type
    - make it a subclass only when you need to make a more specific version of a class and need to override or add new behaviors
    - use an abstract class when you want to define a template for a group of subclasses and when you want to ensure that there is no one will can make object from this abstracted class
    - use an interface when you want to define a role that other classes can play
- All abstract methods must be implemented in the first concrete subclass in the inheritance tree.
- Every class in Java is either a direct or indirect subclass of class Object (java.lang. Object).
- A class that implements an interface must implement all the methods of the interface, except default and static methods (which we'll see in Chapter 12).
- To invoke the superclass version of a method from a subclass that’s overridden the method, use the super keyword. Example: super. runReport () ;