Chapter - 8

Concrete Classes:

- You can make objects of that type.
- They are specific enough to get instantiated.

Abstract Class:

- That cannot be instantiated.
- To stop anyone using “new” on that type.
- Abstract type can still be used as reference type.

```
abstract public class Canine extends Animal
{
        public void roam(){ }
}

public class Makecanine(){
        public void go(){
                Canine c;
                c = new Dog();      // This is ok, as you use it as a reference type.
                c = new Canine();  //Compiler won't let you do this.
                c.roam();
        }
}
```

Why have an abstract class?

- To use it as a polymorphic argument or return type. Or to make a polymorphic array.
- It has no purpose until it’s extended. The instances of its subclass do the work at runtime.

Abstract class ⇒ must be extended

Abstract methods ⇒ must be Overridden

Abstract methods:

- No body (because the code inside makes no sense)
- Just end the declaration with a semicolon.
- You can't have an abstract method in a non-abstract class.

What's the use of Abstract methods?

⇒ You have defined a part of protocol for Subclasses.

Which is good for…
→ Polymorphism!!!
All the subclasses will have this method, which benefits Polymorphism
As you don't need to know the subtype and can simply call the superclass functions.

Since the abstract methods solely exist for polymorphism, the first concrete class in the inheritance tree must implement all the abstract methods.

Class Object:

- Every class extends class Object (without any need to mention it).
- Mother of all classes, Superclass of Everything.
- Any class that doesn't explicitly extends another class, implicitly extends the class object.

Is Class object Abstract?

→ No.

Purpose of Object class?

1. Polymorphic type for methods that need to work on any class.
2. To provide real method code that all objects in Java need at runtime. (For eg: Some methods related to threads that all the objects inherit)

When you put a Dog object into an ArrayList<Dog>, it goes in and comes out as a reference to type Dog.

When you put a Dog object into an ArrayList<Object>, it goes in and comes out as a reference to type Object.

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog =  new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0); //Compilation error
```

The compiler always checks for the reference type and not the object type.

You can call a method on an object only if the class of the reference variable has that method.

Casting an object back to its real type.

```
Object o = al.get(index);
Dog d = (Dog) o;   //casting the object back to a Dog.
d.roam();
```

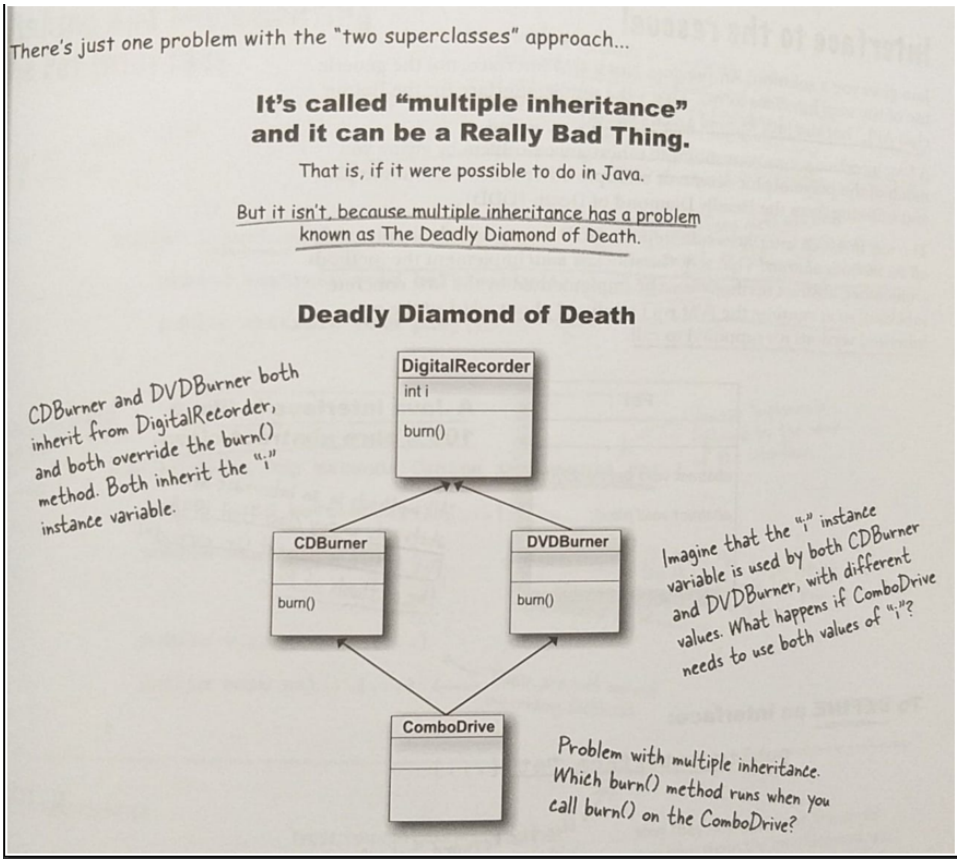
If you are not sure it's a Dog, then use instanceof operator.

```
if(o instanceof Dog){
        Dog d = (Dog) o;
```

```
}
```

Multiple Inheritance:

It has a problem known as Deadly Diamond of Death



Interfaces:

- Solves the multiple interface problem.
- It is a 100% pure abstract class.
- Make all the methods abstract, and the subclass decides how to implement it. So JVM isn't confused about which methods to call.

Interface Implementation:

```
public interface Pet{       // use the keyword interface instead of class
        public abstract void beFriendly();
```

```
        public abstract void play();
}

public class Dog extends Canine implements Pet {
        public void beFriendly() { }  // you need to implement those here.
        public void play() { }
}
```

When you implement an interface you can still extend a class.

Class as Polymorphic type: The objects of a class that is a subclass of the polymorphic type can use the polymorphic type.

Interface as Polymorphic type: Anyone in the Inheritance tree just that they should implement it.

Multiple Interfaces:

public class Dog extends Animal implements Pet, Saveable, Paintable { . . .}

When to make what?

Class:

That doesn't extend anything, and doesn't pass the IS-A test from anything.

Subclass:

When you need to make a more specific version of a class and need to override or add new behaviours.

Abstract class:

When you want to define a template for a group of subclasses and you have at least some implementation code that subclasses can use.

Interface:

A role that other classes can play, regardless of where those classes are in the inheritance tree.