



Some slides contain lots of animation.

Interfaces



- ** Interfaces & a little more on Abstract Classes
- ** Multiple Inheritance



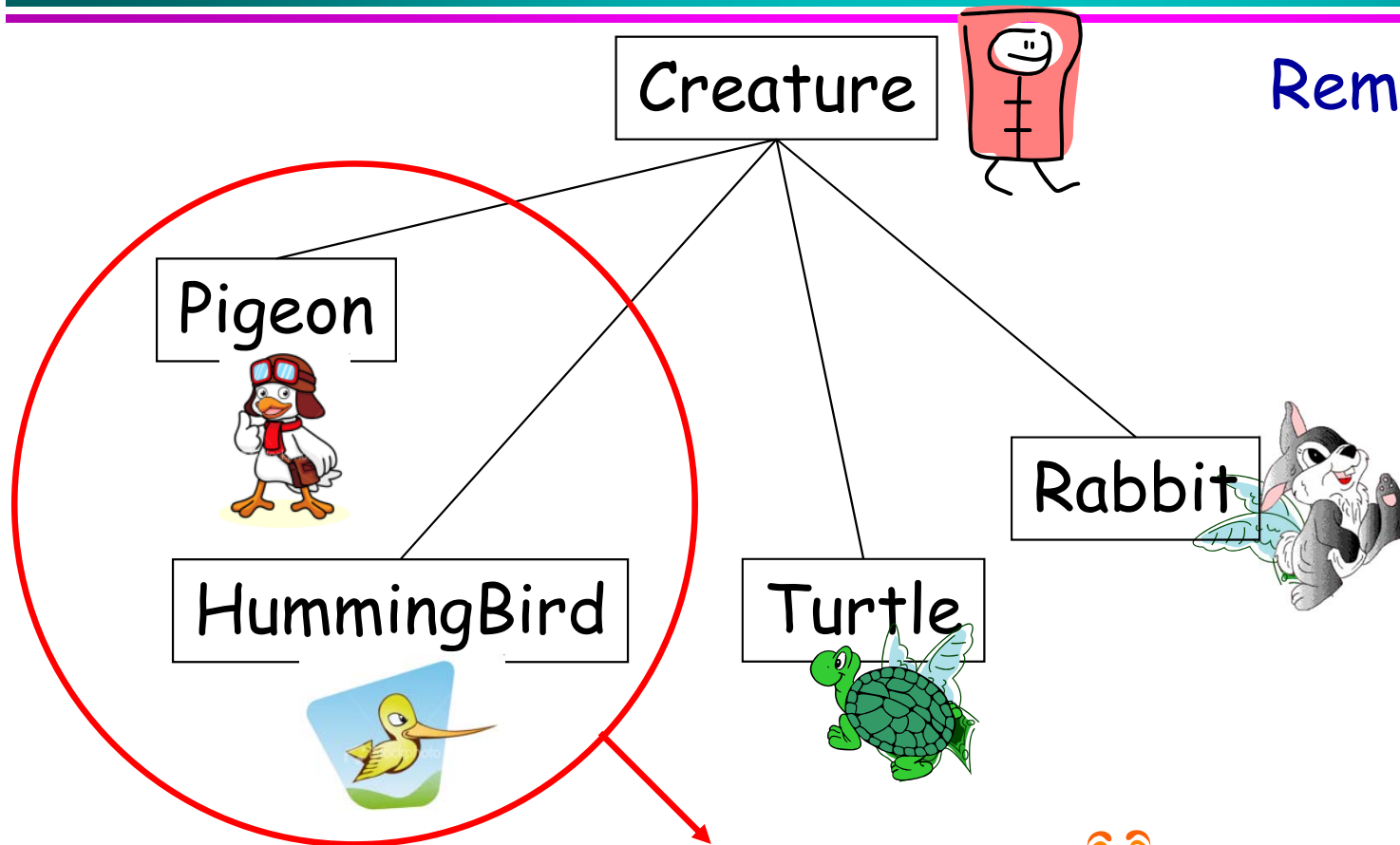
Chapter 6 (section 6.1) – “Big Java” book

Chapter 8 – “Head First Java” book

Chapter 15 – “Introduction to Java Programming” book

Chapter 4 – “Java in a Nutshell” book

More creatures (*before abstract*)



Remember?

Creature

```
String name;  
String tailType;  
Color color;  
int speed;
```

```
run();  
swim();  
fly();
```

but if we put in a generic **fly** method in **Creature** so would Rabbits and Turtles!!

birds fly...



The classes that need to *fly* implement the **fly()** method.

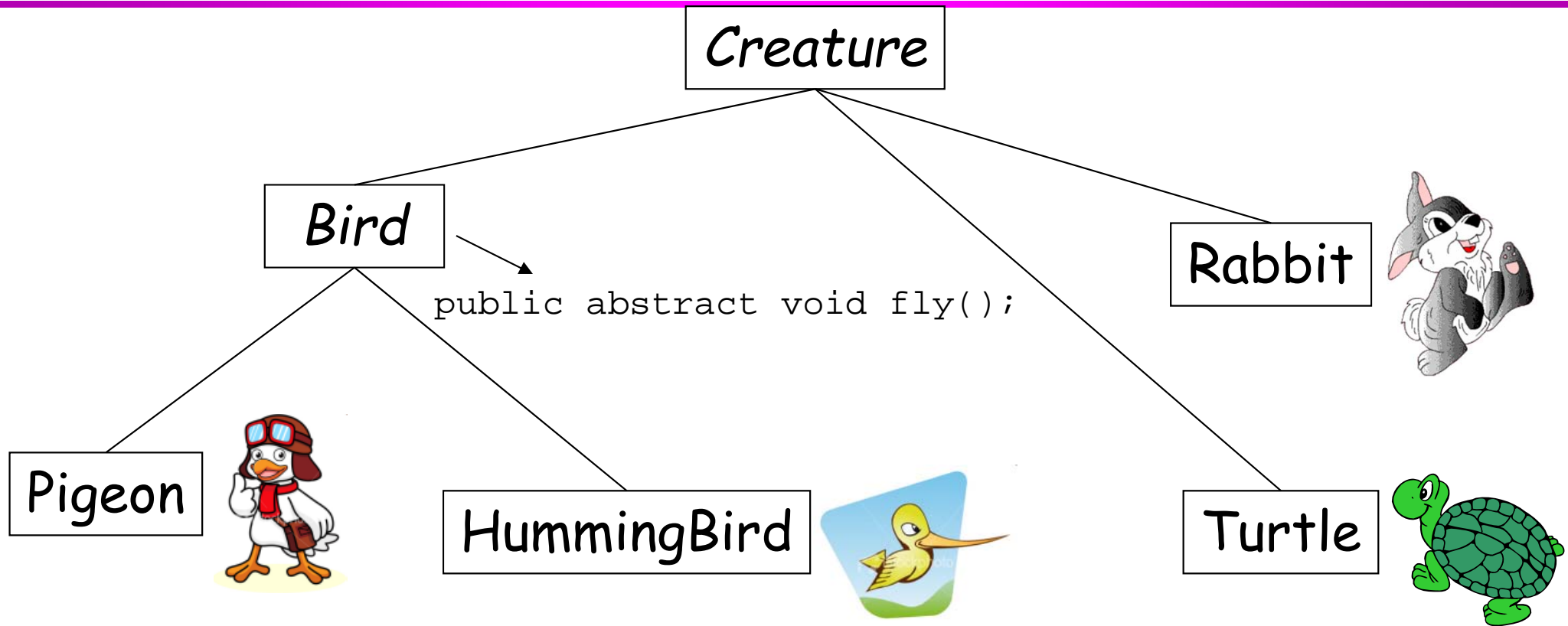


All subclasses of the then abstract class **Creature** must provide a **fly()** method. Even those that do not *fly*!



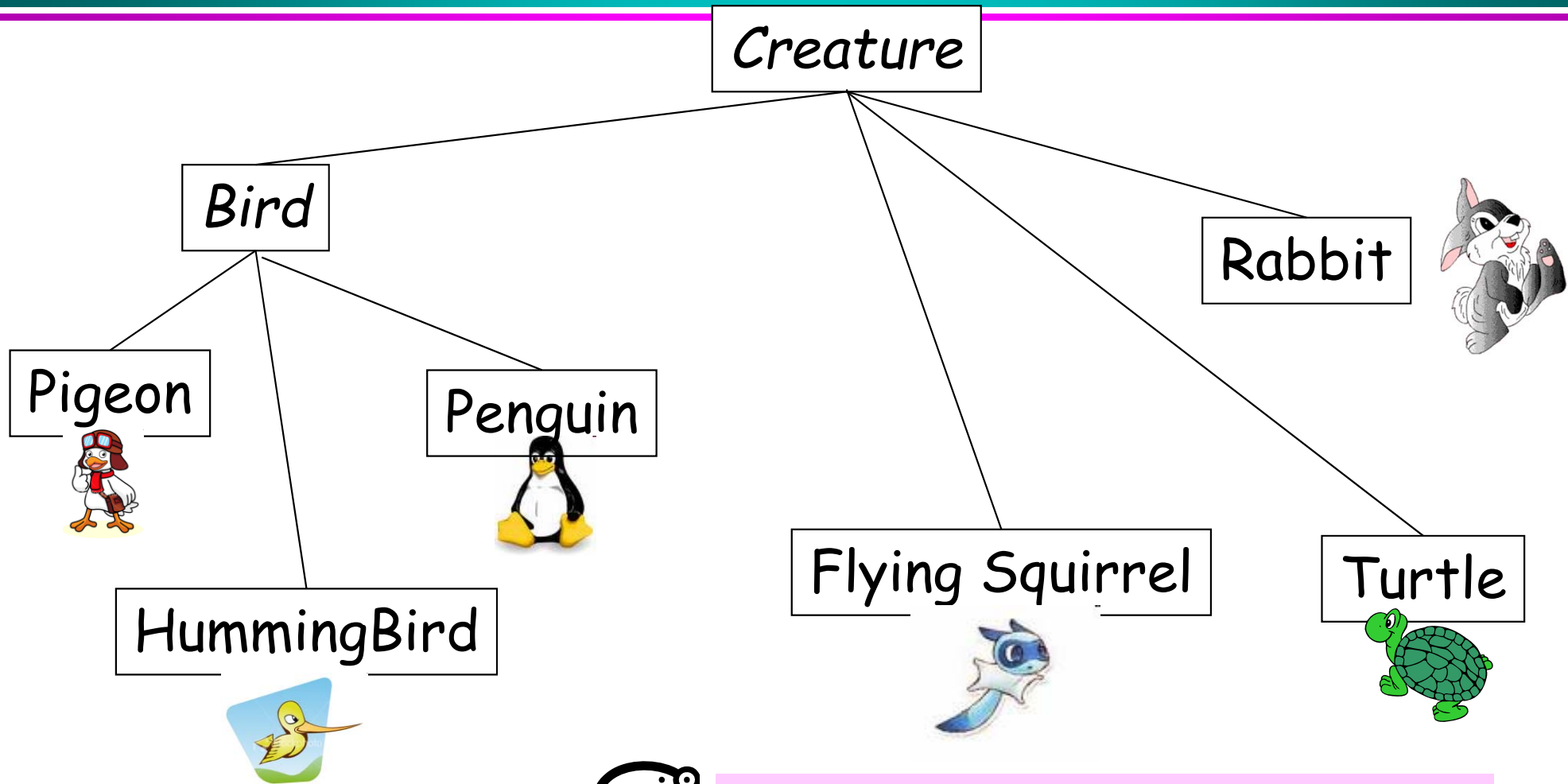
Instead of providing generic methods in **Creature**, we could make the **fly()** method abstract.

Creating multiple abstract parents



Now Birds can *fly*, but Rabbits and Turtles remain grounded!

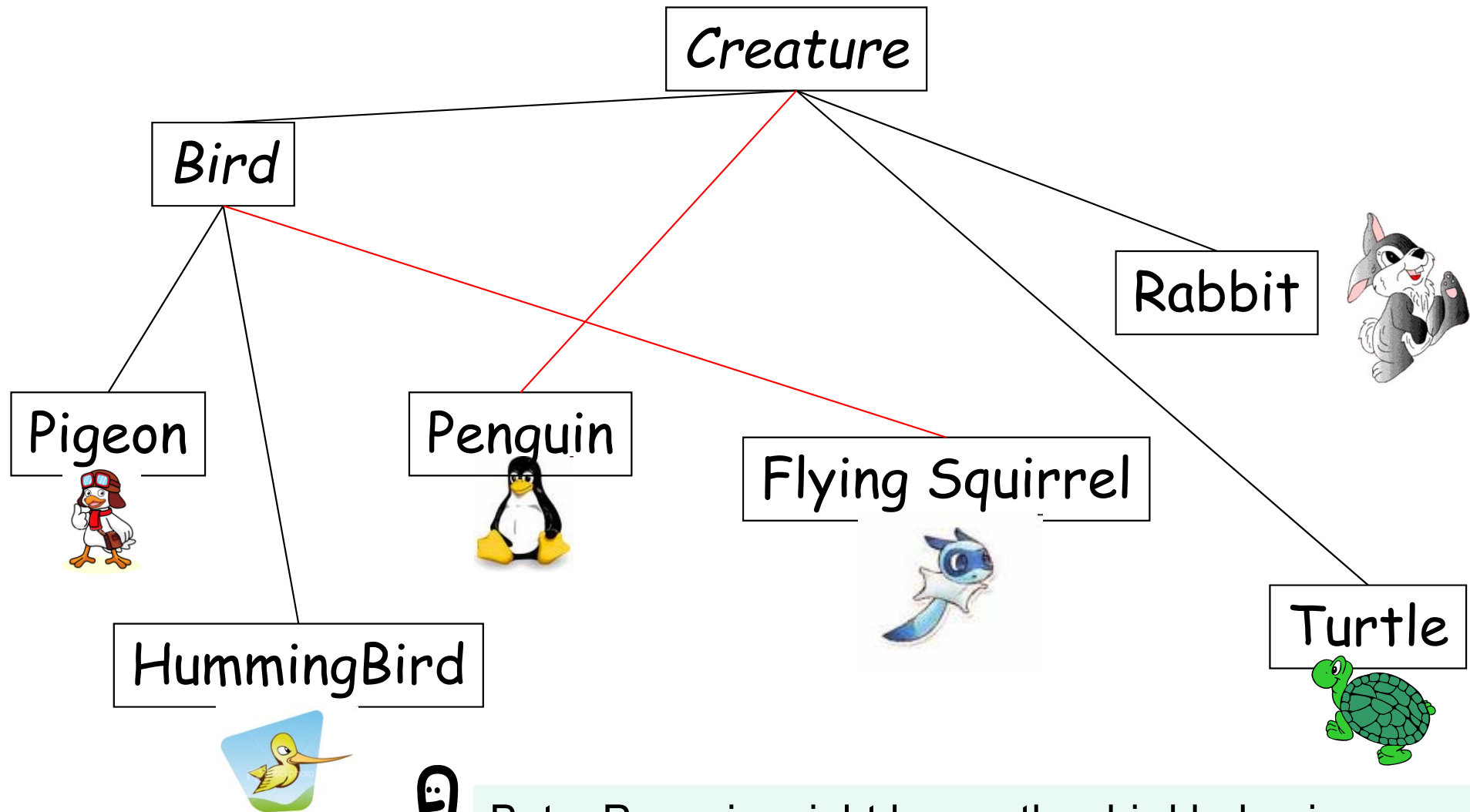
Let us add more Creatures (1/3)



Do Penguins fly?
And how about flying mammals?

Let us add more Creatures (2/3)

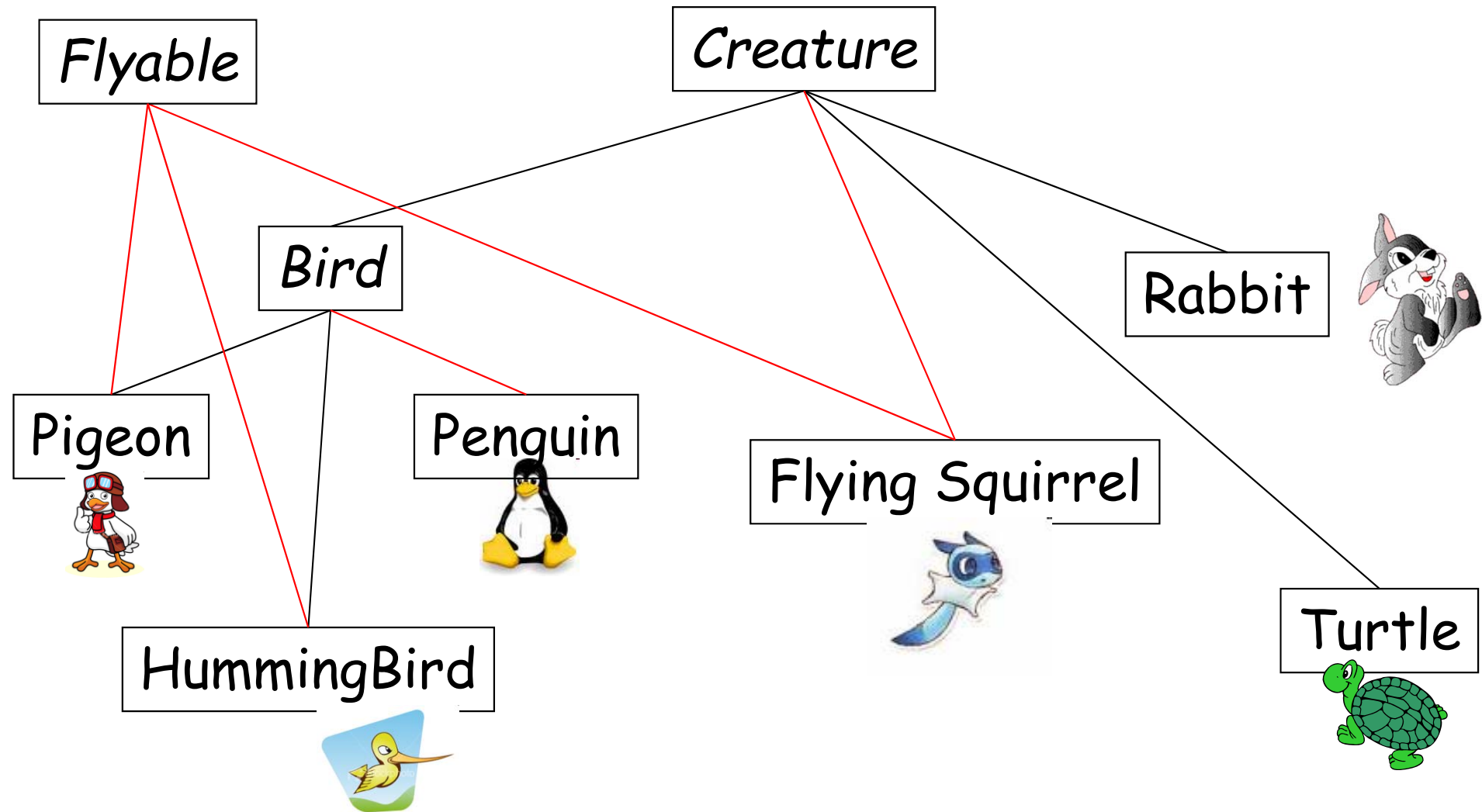
- How can we make a *penguin not fly* and a *flying squirrel fly*?



But a Penguin might have other bird behaviours ...
And a Flying Squirrel is not a bird!

Let us add more Creatures (3/3)

- Why not have **two parents**? Then only creatures that *fly* have the parent *Flyable* ...





“Deadly Diamond Problem” ...

DigitalRecorder

int i;

play()

... or why we cannot have
“multiple inheritance” in Java

CDRecorder

play()

Both inherit from
DigitalRecorder
and both override
method **play()**.

DVDRecorder

play()



ComboDrive



What is the value of the inherited
instance variable **i**? Which version
of the **play()** method is inherited?

- Java’s “multiple inheritance” is *at interface level* only!
 - If you’ve already got **class A** and **class B**, and you want to extend **A** and **B** together to generate **class C**, forget about it. You can’t do that!
- *Only interfaces can do multiple inheritance ... at design level.*

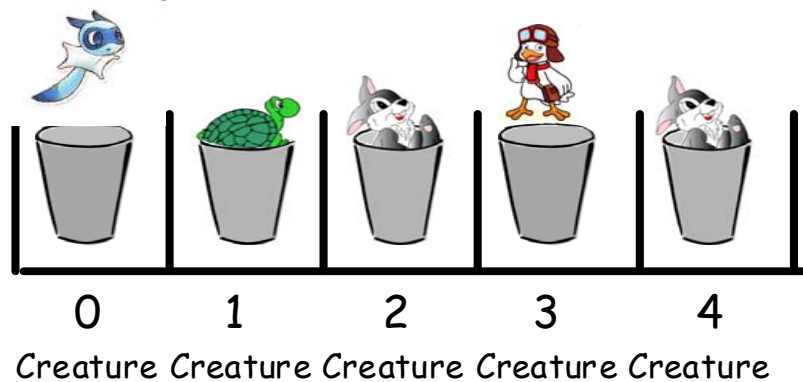
Interfaces

- An **interface** is like a **100% abstract** class.

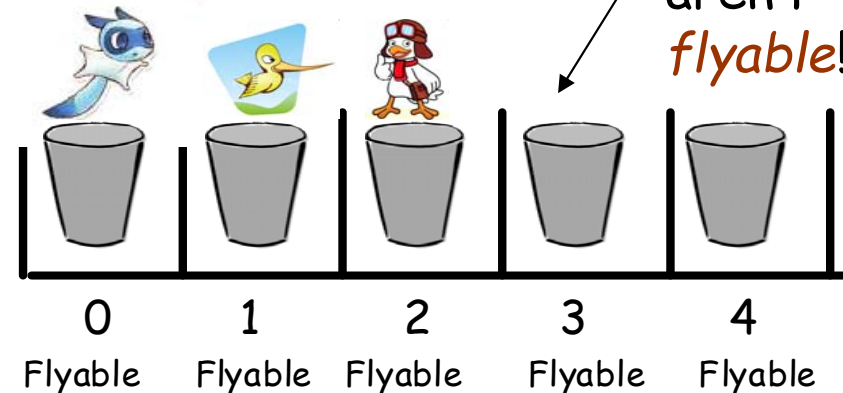


An interface allows polymorphic capabilities without the **problems** of multiple inheritance.

Note: you can't put *turtle* in here - they aren't *flyable*!!



or



- Since an **interface** has **NO** implemented methods, multiple inheritance is not a problem, as no class inherits a "finished" method.



From Java SE8, **interfaces** can also have **default** and **static** methods. More about this, later in these slides ...

interface Flyable

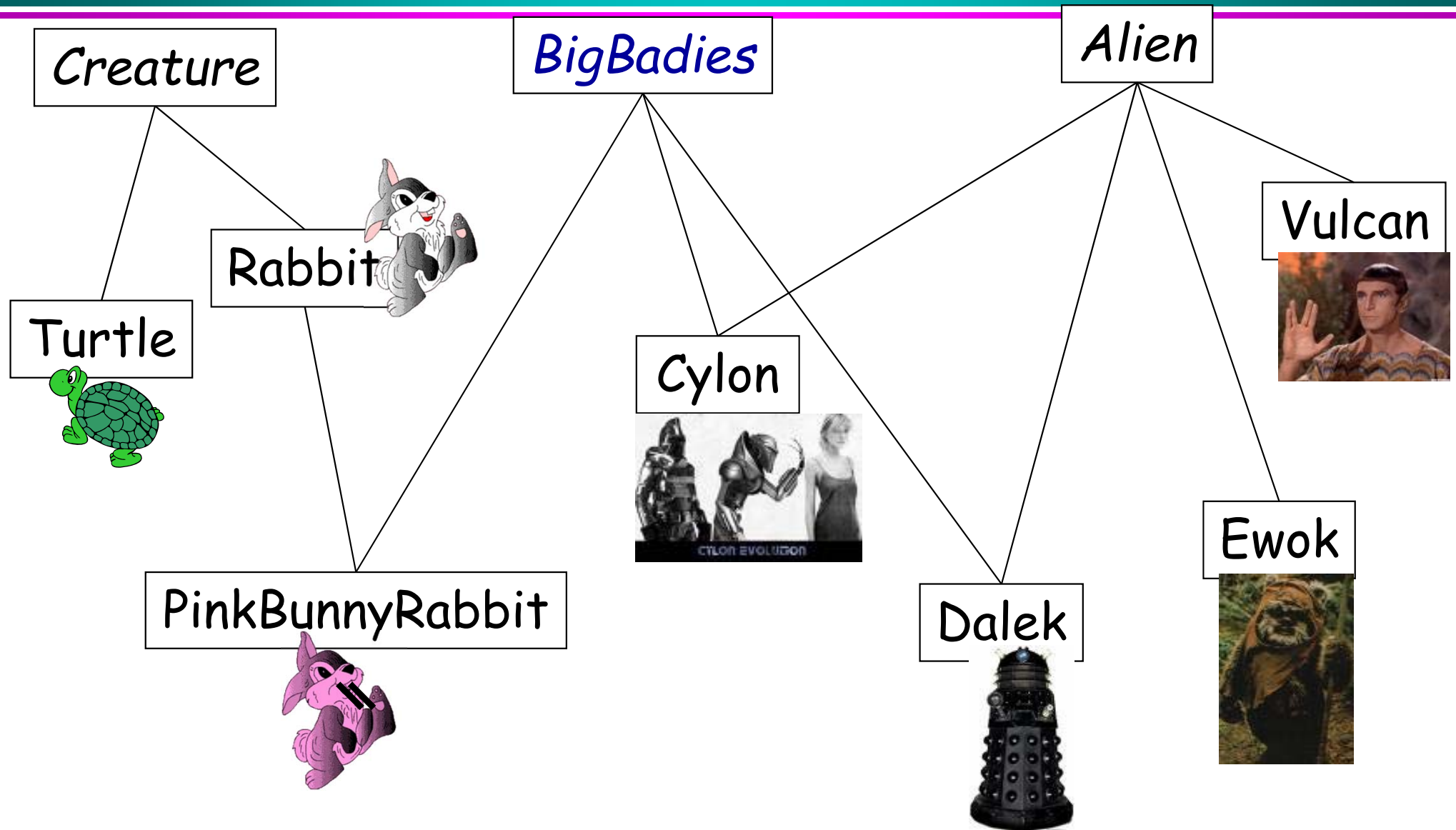
```
public interface Flyable {  
    public abstract void fly(); // OR public void fly();  
    // Even if you don't declare the method abstract or  
    // public, it is!!!  
}
```

```
public class FlyingSquirrel extends Creature  
    implements Flyable {  
    public void fly() {  
        // some code  
    }  
    public void run(int duration) {  
        // some code  
    }  
}
```

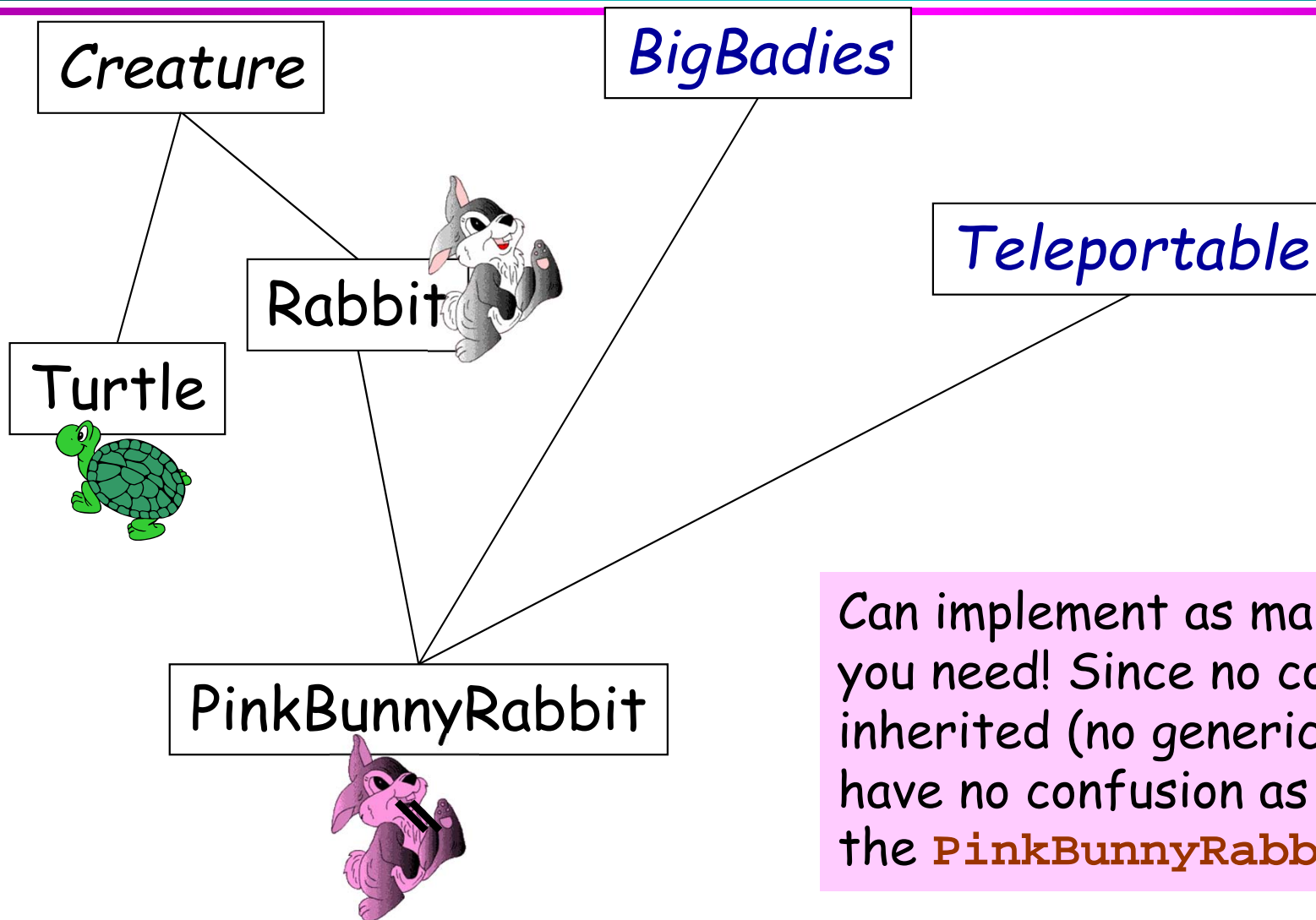
must provide implementation, as
you "said" you are a Creature

must provide implementation,
as you "said" you are Flyable

Interfaces can be used across *different inheritance trees*



Multiple interfaces



Can implement as many interfaces as you need! Since no code is actually inherited (no generic methods), you have no confusion as to which method the **PinkBunnyRabbit** has inherited!

extends and *implements*

- A class can only **extend** 1 class:
 - meaning 1 class can only have 1 parent;
 - a **PinkBunnyRabbit** can only have one direct parent – **Rabbit**.
- A class can **implement** as many interfaces as it likes!
 - A **PinkBunnyRabbit** can *be* (via interfaces) a **BigBadies** and **Teleportable**.

Interfaces

- At *design time*, we can write code that needn't worry about the *implementation* of any class that implements **Flyable** (or **BigBadies** or **Teleportable**)
 - We can treat the implementation as a black box, and rest safe in the knowledge that it must provide **fly()**.
- Interfaces are then like *certificates*, which say “I provide these services”.
 - You *can't* make an instance of an interface so e.g.,

```
Flyable friend = new Flyable(); // ERROR!
```

Practice Exercise 1

- Which of the following is a *correct interface*?

```
interface A {  
    void print() {  
        // some code  
    }  
}
```

```
abstract interface B {  
    print();  
}
```

```
abstract interface C extends I1, I2 {  
    abstract void print() {  
        // some code  
    };  
}
```

```
interface D {  
    void print();  
}
```

Notes on Abstract Classes & Interfaces

- Neither **abstract classes** and **interfaces** can have an instance made of them.
- If you don't provide *any* method implementation, then use an **interface** instead of an **abstract class**.
- A class can **implement** many interfaces, but **extends** only one superclass.
- Interfaces are how Java provides (a kind of) *multiple inheritance*.
- If even one method in a class is declared to be abstract, then the whole class must be declared abstract.
- Both abstract classes and interfaces *can* contain constants, which will be inherited by classes that *extend* or *implement* them, respectively.


Example (1/2): Abstract Class *versus* Interface

```
public interface Countable {
    int x = 20;
    int y = 30;          // declaring interface constants
    void counting();     // declaring an interface method
}

public class Example implements Countable {
    int x = 1;
    int y = 2;
    int sum = 0;
    public void counting() { // implements interface method
        sum = x + y;
        System.out.println("Sum is " + sum);
    }
}

public class Example3 extends Example implements Countable {
    int sub = 0;
    public void counting() {
        // implements interface abstract method
        sub = Countable.y - super.x;
        System.out.println("Sub is " + sub);
    }
}
```

*Unnecessary to provide an implementation for **counting()** at this level.*



Example (2/2): Abstract Class *versus* Interface

```
public class ResultOfCount {  
    public static void main(String args[]) {  
        Example x = new Example();  
        x.counting();  
        Example1 y = new Example1();  
        y.counting();  
    }  
}
```

- Analysis of program:
 - Output of the program:
Sum is 3
Sub is 29
 - The **counting()** method is implemented (overridden) by two classes that implement the **Countable** interface.
 - An **interface may have many methods**. If a class implements an interface, but only implements some of its methods, then this class becomes an abstract class; it cannot be instantiated.



... and things for you to try out!

Example: Abstract Classes & Interfaces Implementation

```
interface InterfaceExample {  
    void method1();  
    void method2();  
}
```

Example1 implements **method1()**, but not **method2()** so it cannot be instantiated.

```
class Example1 implements InterfaceExample {  
    public void method1() {  
        // implement 1st method  
    }  
}
```

```
class Example2 extends Example1 {  
    public void method2() {  
        // implement the 2nd method  
    }  
}
```

Example2 implements **method2()** (and inherited **method1()** from **Example1**), so it can be instantiated.

Extending an Interface

- Like classes, *interfaces can be extended* as well.

```
interface Father {  
    int age = 30;  
    void wash();  
}  
interface Mother {  
    long bank_account = 100000;  
    void cook();  
}  
interface Child extends Father, Mother {  
    void cry(boolean tear);  
}
```

Child inherits from **Father** and **Mother** and has the following:

```
int age = 30; (!!!)  
long bank_account = 100000;  
void wash();  
void cook();  
void cry(boolean tear);
```



This example tells us *how to pack several interfaces together*.

Name Conflicts

- What happens if **Father** interface and **Mother** interface contain *same named methods* and *variables* (constants)?
 - *Same named methods*:
 - If they have different parameters, then **Child** interface has both (this is same as *overloading*).
 - If they differ by only return type, then *error*.
 - If the two methods are identical, only keep one.
 - *Same named constants*: we keep both constants. To refer to them, *use parent interface name* as prefix.
 - **Example**:
 - If both **Father** and **Mother** contain an **age** variable, then **Child** interface contains both.
 - To refer to them, we use: **Father.age** or **Mother.age**.

Java Interfaces: before/after Java SE8

- Before Java SE8, interfaces could have:
 - constant fields (e.g. `public static final int x = 10;`);
 - `abstract` methods (e.g. `public abstract void doStuff();`)
- From Java SE8, interfaces can also have:
 - `default` methods → Allow developers to **add new functionality to interfaces**, without impacting any existing classes that are already implementing the interface.
 - **Can be overridden** in the class that implements the interface.
 - Provide **backward compatibility** for existing interfaces.
 - `static` methods → Allow developers to **define utility methods** in the interface.
 - Are similar to `default` methods, but **cannot be overridden** in the class that implements the interface.



For interfaces with same `default` method signatures, invoke `super` on relevant interface.

Example: Interface with default and static methods

```
public interface Interviewer {
    public abstract void conductInterview(String name);
    default void submitInterviewStatus() {
        System.out.println("Accept");
    }
    static void bookIntRoom(String day, int duration) {
        System.out.println("Interview on: " + day);
        System.out.println("Book room for: " + duration + " hour(s)");
    }
}

public class Manager implements Interviewer {
    public void conductInterview(String name) {
        System.out.println("Interview for " + name);
    }
}

public class Project {
    public static void main(String[] args) {
        Manager mgr = new Manager();
        mgr.conductInterview("Jane Smith");
        Interviewer.bookIntRoom("Monday", 1);
        mgr.submitInterviewStatus();
    }
}
```

Practice Exercise 2

- What is the **output** of the program below? Explain.

```
public interface TestInterface1 {  
    default void show() {  
        System.out.println("Default TestInterface1");  
    }  
}  
  
public interface TestInterface2 {  
    default void show() {  
        System.out.println("Default TestInterface2");  
    }  
}  
  
public class TestClass implements TestInterface1,  
                                   TestInterface2 {  
  
    public void show() {  
        TestInterface1.super.show();  
        TestInterface2.super.show();  
    }  
  
    public static void main(String[] args) {  
        TestClass d = new TestClass();  
        d.show();  
    }  
}
```




... and things for you to try out!

Practice Exercise 3

- What will happen if each of the statements is inserted **where** indicated in the code?

```
public interface MyConstants {  
    int r = 4;  
    int s = 6;  
    // INSERT CODE HERE  
}
```

1. `final double circumference = 2*Math.PI*r;`
2. `int total = total + r + s;`
3. `int AREA = r*s;`
4. `public static MAIN = 15;`
5. `protected int CODE = 31337;`