UBC OKANAGAN

COSC 121
Computer Programming II

Abstract Classes and Interfaces

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Previously:

- Abstract classes
- User-defined interfaces

Today:

- More on user-defined interfaces
  - `default`, `static`, and `private` methods
- Java standard interfaces
  - `Comparable` interface
  - `Cloneable` interface
    - Shallow vs. deep cloning.
- Interfaces or abstract classes?

More on Interfaces

Interface Structure

Interfaces have **zero or more** of the following items:

```
public interface somename {
   constants
   abstract methods
   default methods          //introduced in Java 8
   static methods           //introduced in Java 8
   private methods          //introduced in Java 9
   private static methods   //introduced in Java 9
}
```

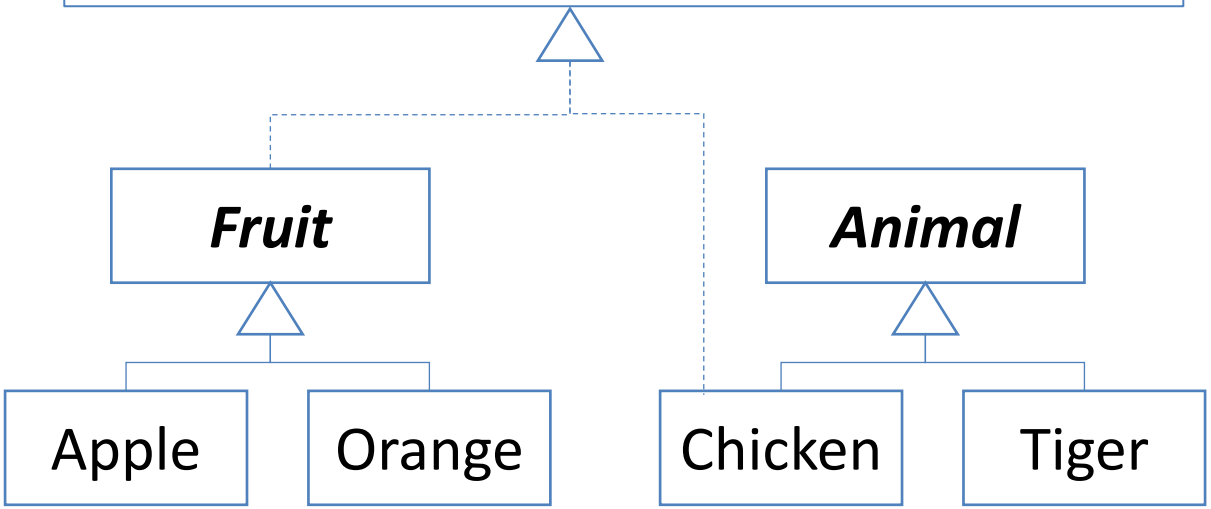
`default` methods

`default` methods provide *default implementation* in an interface.

Example:

`Fruit` and `Chicken` classes don't need to implement `print()`. However, they can also provide their own implementation of print if required.

```
public interface Edible {
  public default void print() {
     System.out.println("edible!");
  }
}
```



Why need them? For backward compatibility. ***i.e.*** *support* adding new methods to current interfaces without breaking the old code.

`static` methods

`static` methods are implemented methods that are called using the interface name.

Example:

```
public interface Foo {
    public static void printMe() {
        System.out.println("I am foo!");
    }
}
```

Static method can only be called by interface name as follows:

```
Foo.printMe();
```

`private` Methods

`private` methods are accessible only within that interface.

Why?

- default & static methods are public by default. If you want part of the code to be accessible only in the interface, use private.

Example:

```
interface F{
  private void helper(int n){ System.out.println(n); } // hidden
  default void one(){ helper(1); }                     // public by default
  default void two(){ helper(2); }                     // public by default
}

public class C implements F{
 /** cannot use or override helper() **/
}
```

Java Standard Interfaces

Java Standard Interfaces

Java **provides** standard interfaces to serve different purposes.

For example:

- **`Comparable`**
  - This interface allows comparing or sorting objects.
- **`Cloneable`**
  - This interface allows cloning objects (creating exact copies)

The Comparable Interface

Comparable

```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

When you want to compare two objects, e.g. two Employees: is e1>e2?, the two objects must be comparable.

Any class that implements the **`Comparable`** interface must have a method **`compareTo`**.

- The `compareTo` method allows the class to define an appropriate measure of comparison.

The `Comparable` interface is a ***generic*** interface.
The generic type **T** is replaced by a concrete type when implementing this interface.

Comparable: Example

Example: Employee class

```
public class Employee implements Comparable<Employee> {
    int salary;
    //other class members and constructors
    public int compareTo(Employee otherEmployee) {
        if (salary > otherEmployee.salary)
            return 1;
        if (salary < otherEmployee.salary)
            return -1;
        return 0;
    }
}
```

Two usages:

(1) Which is “larger”?

```
Employee e1 = new Employee();
Employee e2 = new Employee();
if(e1.compareTo(e2) > 0)
    System.out.println("E1 is richer!");
```

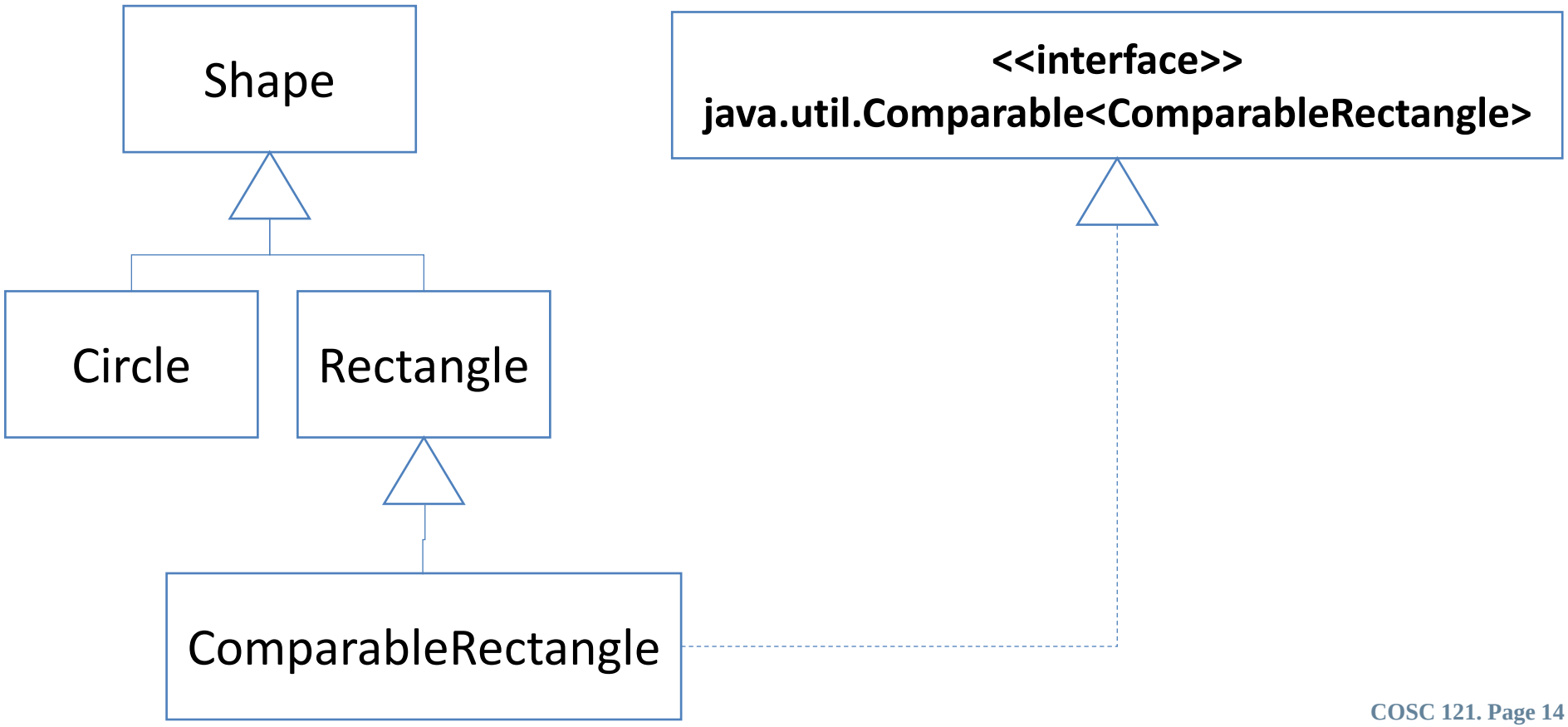
(2) Sorting (ascending)

```
Employee[] staff = new Employee[4];
//...
Arrays.sort(staff);
```

Comparable: Exercise

In our Shape example, Create a subclass called ComparableRectangle that implements the Comparable interface. The **criteria for comparison is the** rectangle **area**.

Then, create an array of comparable rectangles, sort its elements, and then display their information on the screen.



The `Cloneable` Interface

Cloneable

In order to create a **clone** of an object (i.e., **a field-for-field copy** of the object), a class must implement the Cloneable interface.

- By convention, classes that implement this interface should override `Object.clone` (which is protected) with a **public** method

```
public class Robot implements Cloneable{
    public int x, y; //primitive type
    public Battery b; //reference type
    public Robot(int x, int y) {
        this.x = x;
        this.y = y;
        b = new Battery();
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
public class RobotTest{
    public static void main(String[] args) throws CloneNotSupportedException{
        Robot robot1 = new Robot(1, 2);
        Robot robot2 = (Robot) robot1.clone(); //create a clone of r1 (a new object)
        System.out.println(robot1 == robot2);
        //displays false as r1 and r2 are referring to different objects
    }
}
```

A few more notes about `Cloneable`

The **`Object's clone`** method returns the type '**`Object`**' which must be casted to the desired type

- **`Robot r1 = (Robot) r2.clone();`**

The **`cloneable`** interface is a ***marker interface*** (i.e. empty).

```
public interface Cloneable {}
```

We implement **`Cloneable`** to indicate to the **`clone()`** method that it is ok for it to make a field-for-field copy of instances of that class.

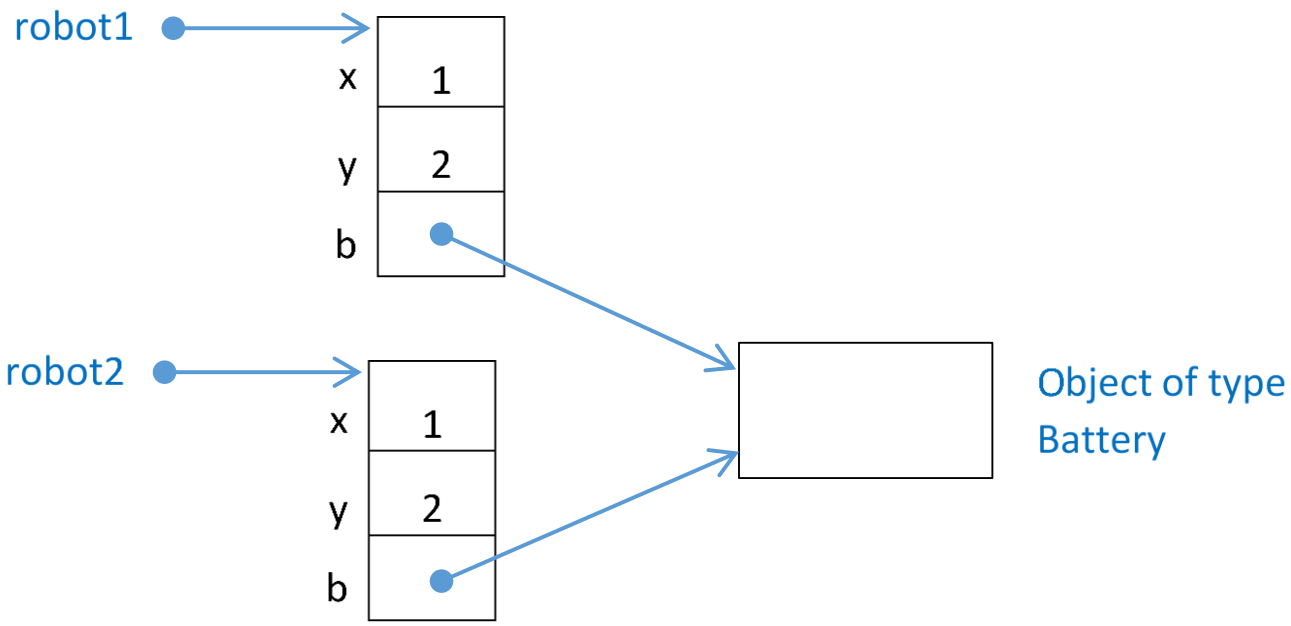
If you try to invoke the clone method on an instance that does not implement **`Cloneable`**, **`CloneNotSupportedException`** will be thrown.

- Handling exceptions (e.g., **`CloneNotSupportedException`**) is covered in Chapter 12.

`Cloneable`: Shallow vs. Deep Copy

The `Object`'s **clone** method performs a '**Shallow Copy**'.

- **Assume `Robot robot2 = (Robot) robot1.clone();`**
- **For a primitive field**, its value is copied.
  - e.g., the value of **x and y** (**int** type) are copied from robot1 to robot2.
- **For an object field**, the reference of the field is copied.
  - This means robot1.b and robot2.b will refer to the same Battery object.



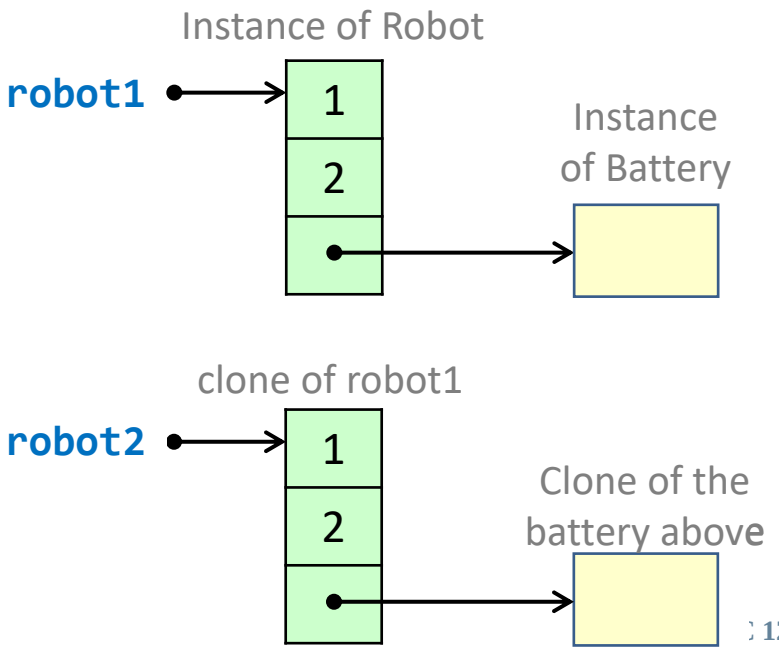
Cloneable: Shallow vs. Deep Copy

In order to perform **Deep Copy**, you have to clone each sub-object within the containing class's clone method.

```
Robot robot2 = (Robot) robot1.clone();
```

```
//in Robot class
public Object clone() throws CloneNotSupportedException{
    //1- create a clone of the Robot object
    Robot r = (Robot)super.clone();
    //2-create a clone for each sub-object & included it in Robot r
    r.battery = (Battery)battery.clone();
    //3- return the result
    return r;
}
```

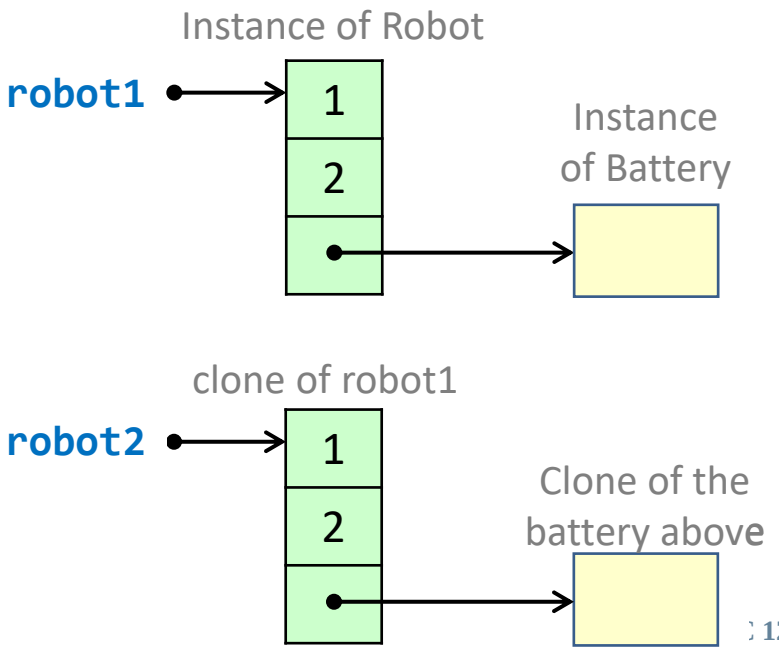
Remember that you also need to implement clone() within Battery class.



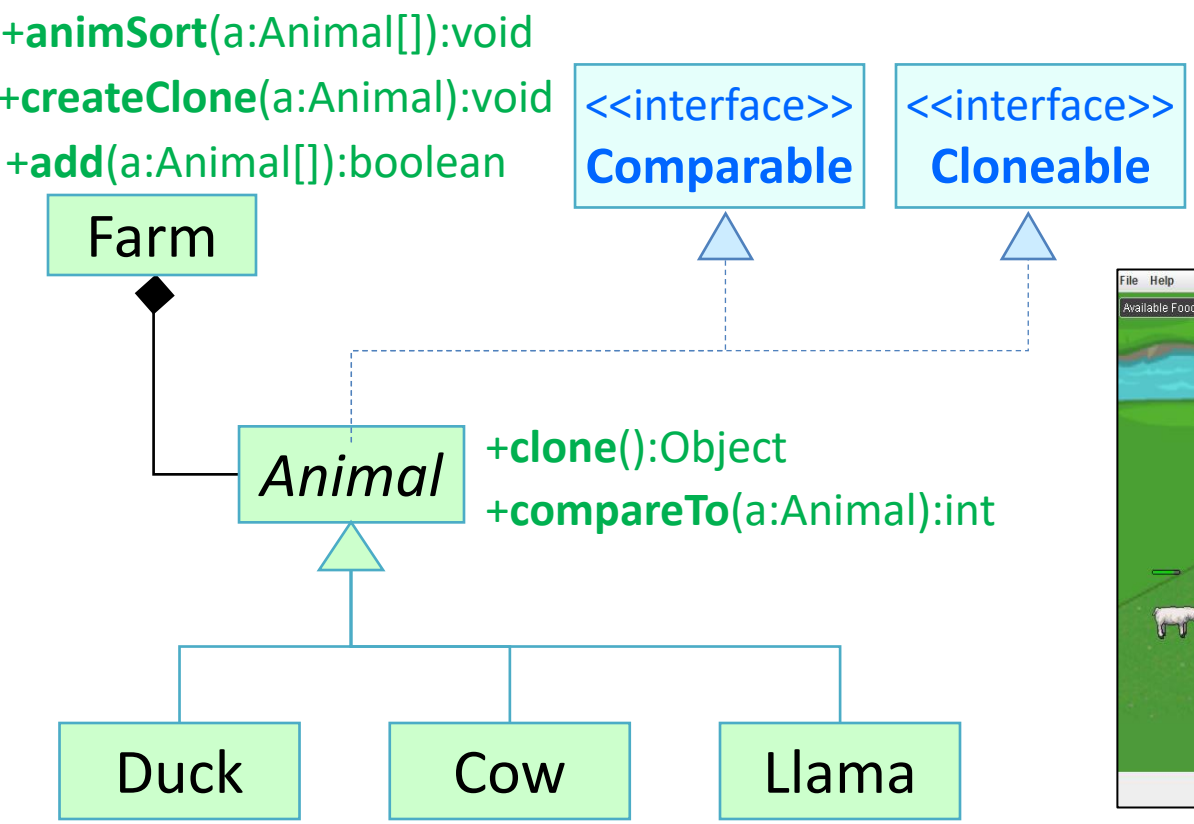
Cloneable: Shallow vs. Deep Copy

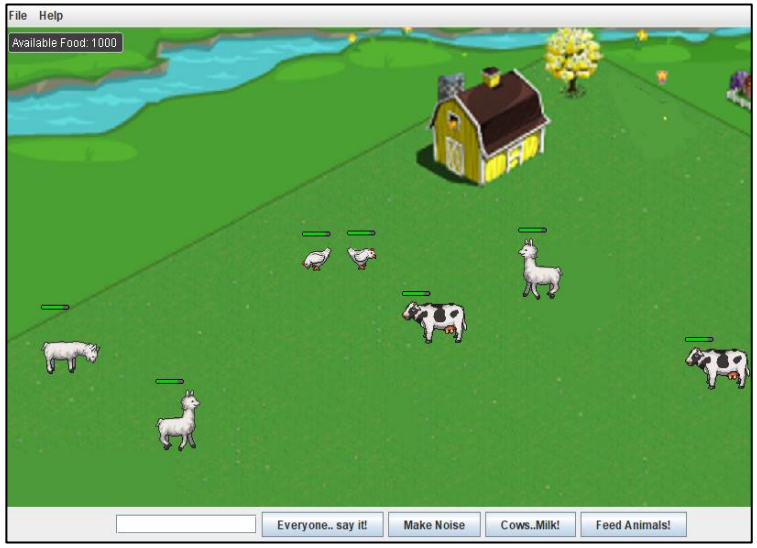
This is more detailed code, but serves the same purpose.

```
//in Robot class
public Object clone() throws CloneNotSupportedException{
    //1- create a clone of the Robot object
    Robot r = (Robot)super.clone();
    //2a- create a clone for each sub-object (i.e. reference attribute)
    Battery b = (Battery)battery.clone();
    //2b-include the cloned sub-object in the Robot object
    r.battery = b;
    //3- return the result
    return r;
}
```



The Project


+animSort(a:Animal[]):void
+createClone(a:Animal):void
+add(a:Animal[]):boolean
Farm
<<interface>>
Comparable
<<interface>>
Cloneable
Animal
+clone():Object
+compareTo(a:Animal):int
Duck
Cow
Llama


File Help
Available Food: 1000
Everyone.. say it!
Make Noise
Cows..Milk!
Feed Animals!

Interfaces or Abstract Classes?

Previously

An **abstract class** is usually used to define **common features** to a set of its subclasses, and it may contain **abstract methods.**

An **interface** is a **class-like construct** that contains

- ***constants***
- **methods: *abstract***, ***default***, or ***static***

Abstract classes use with related subclasses

- e.g., an abstract class Animal is parent of two classes, Cat and Dog

Interfaces include common behavior between related and unrelated classes

- e.g. an interface Edible can be implemented by two classes, Apple and Candy.

Interfaces vs. Abstract Classes: SIMILARITIES

Both place requirements on objects of other classes.

Both can have abstract methods

- Although neither need to have abstract methods.

For Both:

- You cannot create objects of either type.
- You can create reference variables of either type.

Both can inherit.

- abstract class from another class.
- interface from interfaces.

Interfaces vs. Abstract Classes: DIFFERENCES

Interface:

- ***Not a class***: An interface is not a class.
- ***Components***: only contains constants, abstract/default/static methods.
- ***Implementation***: An interface can be implemented by any number of ***unrelated classes***. A class may implement **any number of interfaces**.
- ***Inheritance***: an interface can extend any number of interfaces.

Abstract class:

- ***Root***: All classes share a single root, the Object class, but there is no single root for interfaces.
- ***Components***: can have **constructors**, abstract/concrete methods, instance variables.
- ***Inheritance***: A class can only inherit from one other class.

Interface or abstract class?

When to use abstract classes?

- A strong **is-a** relationship that clearly describes a **parent-child** relationship should be modeled using **classes**.
  - For example, a staff member **is-a** person.
- Also, use abstract class if you want to use super constructors and inherit reference variables and concrete methods.

When to use interfaces?

- A weak is-a relationship, also known as an **is-kind-of** relationship, indicates that an **object possesses a certain property**.
  - For example, all strings are comparable, so String class implements the Comparable interface.
- If **multiple inheritance** is desired.
  - i.e. for multiple inheritance, design one parent as a superclass, and others as interface.