Mahidol University
Faculty of Information
and Communication Technology

LECTURE 10
Interface and Collection

ITCS209 Object Oriented Programming

Dr. Siripen Pongpaichet/Dr. Tipajin Thaipisutikul
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

(Some materials in the lecture are done by Aj. Suppawong Tuarob)

Ref: Java Concepts Early Objects by Cay Horstmann

Semester 2/2023 21 Mar 2024

Class Learning Outcome

- Students can explain and use an **abstract class** concept correctly and appropriately
- Students can explain and use an **interface** concept correctly and appropriately
- Students can explain and use **Java Collection** correctly and appropriately

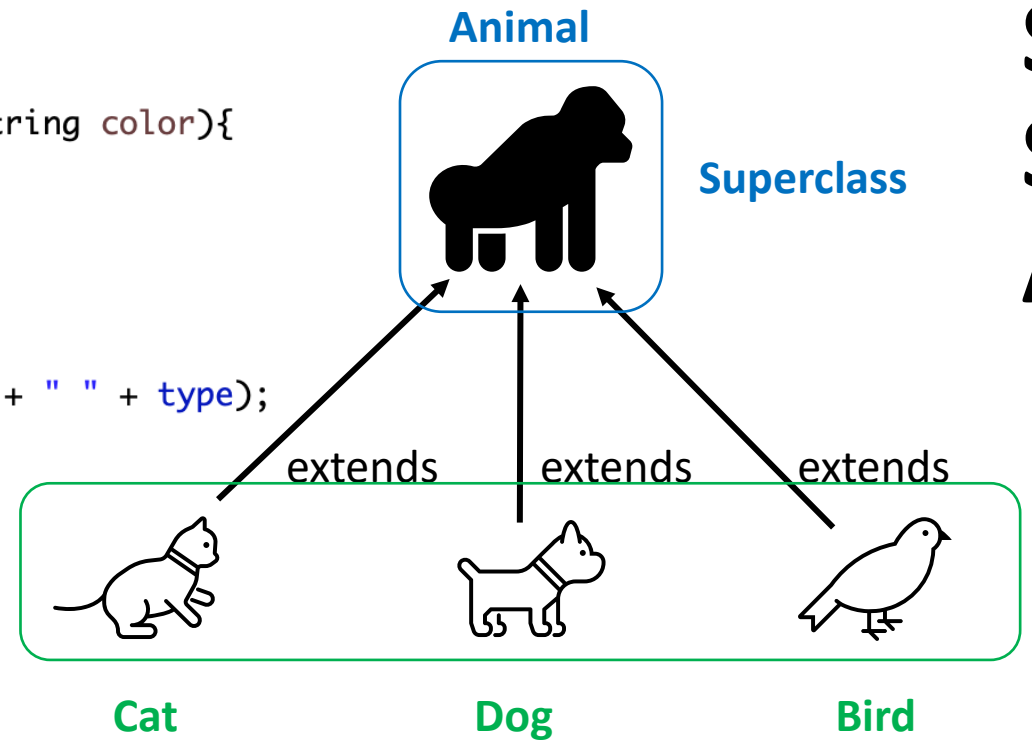
Agenda

- Recall – Inheritance
- Abstract Class
- Interface
- Java Collection
  - List
  - Set
  - Map

Recall: Superclass, Subclass & Abstract Class

Mahidol University
Faculty of Information and Communication Technology

```java
public class Animal {
    public String type;
    public String color;

    public Animal(String type, String color){
        this.type = type;
        this.color = color;
    }

    public void print() {
        System.out.println(color + " " + type);
    }
}
```



How about speak( ) method of Animal?

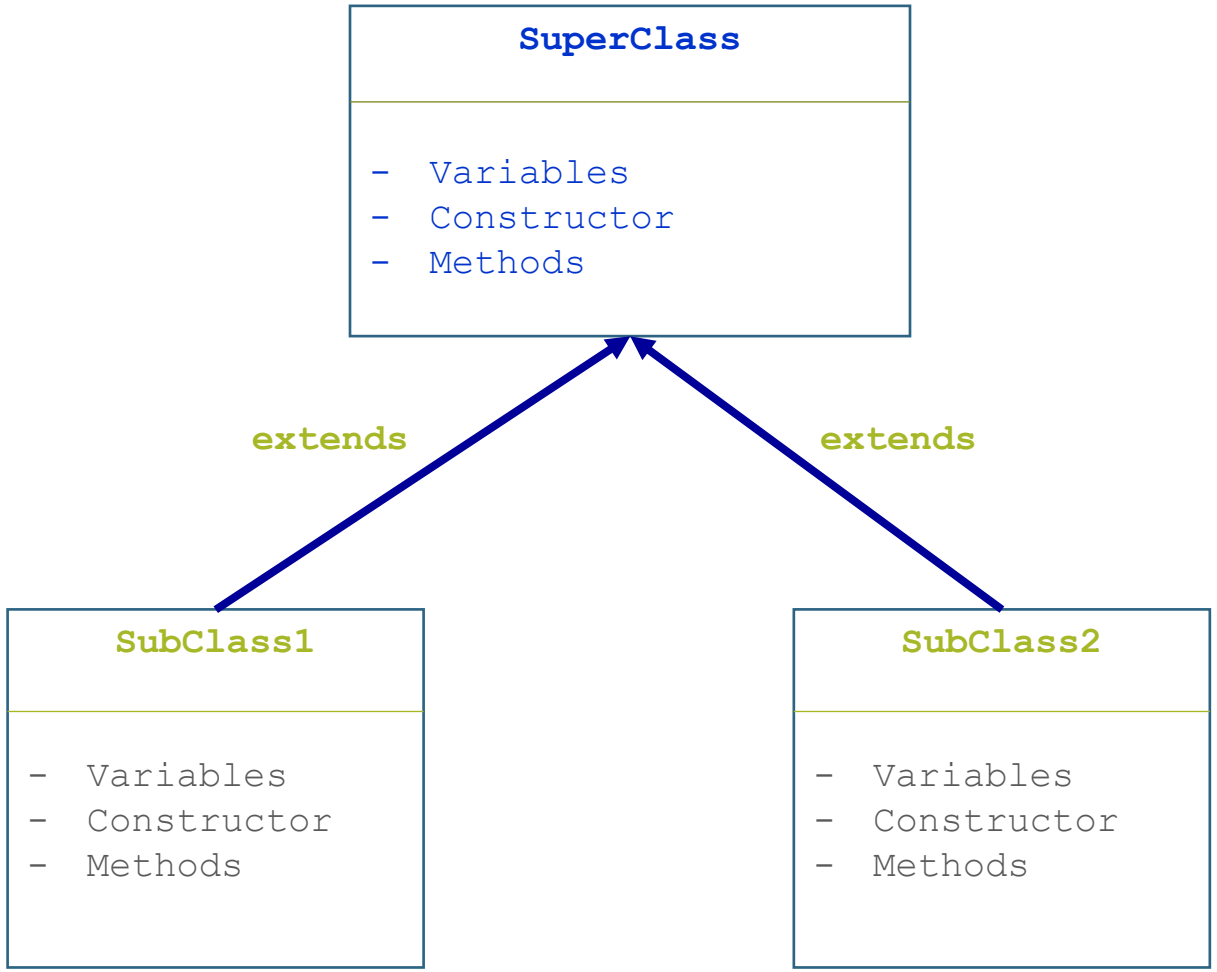
```java
public class Cat extends Animal{

    public Cat(String color) {
        super("Cat", color);
    }

    public void chase() {
        System.out.println("Chasing mouse >>>");
    }
}
```


```java
public class Dog extends Animal{

    public Dog(String color) {
        super("Dog", color);
    }

    public void catching() {
        System.out.println("Catching frisbee");
    }
}
```

```java
public class Bird extends Animal{

    public Bird(String color) {
        super("Bird", color);
    }

    public void fly() {
        System.out.println("Flying in the sky...");
    }
}
```

Abstract Class

Why Abstract Class



Do you remember **inheritance** concept?

Sometime you know what exactly to implement for the super class.

BUT

Sometime you don't !!!! and you want all sub class to have this method. What will you do…

Why Abstract Class (Cont)

For example

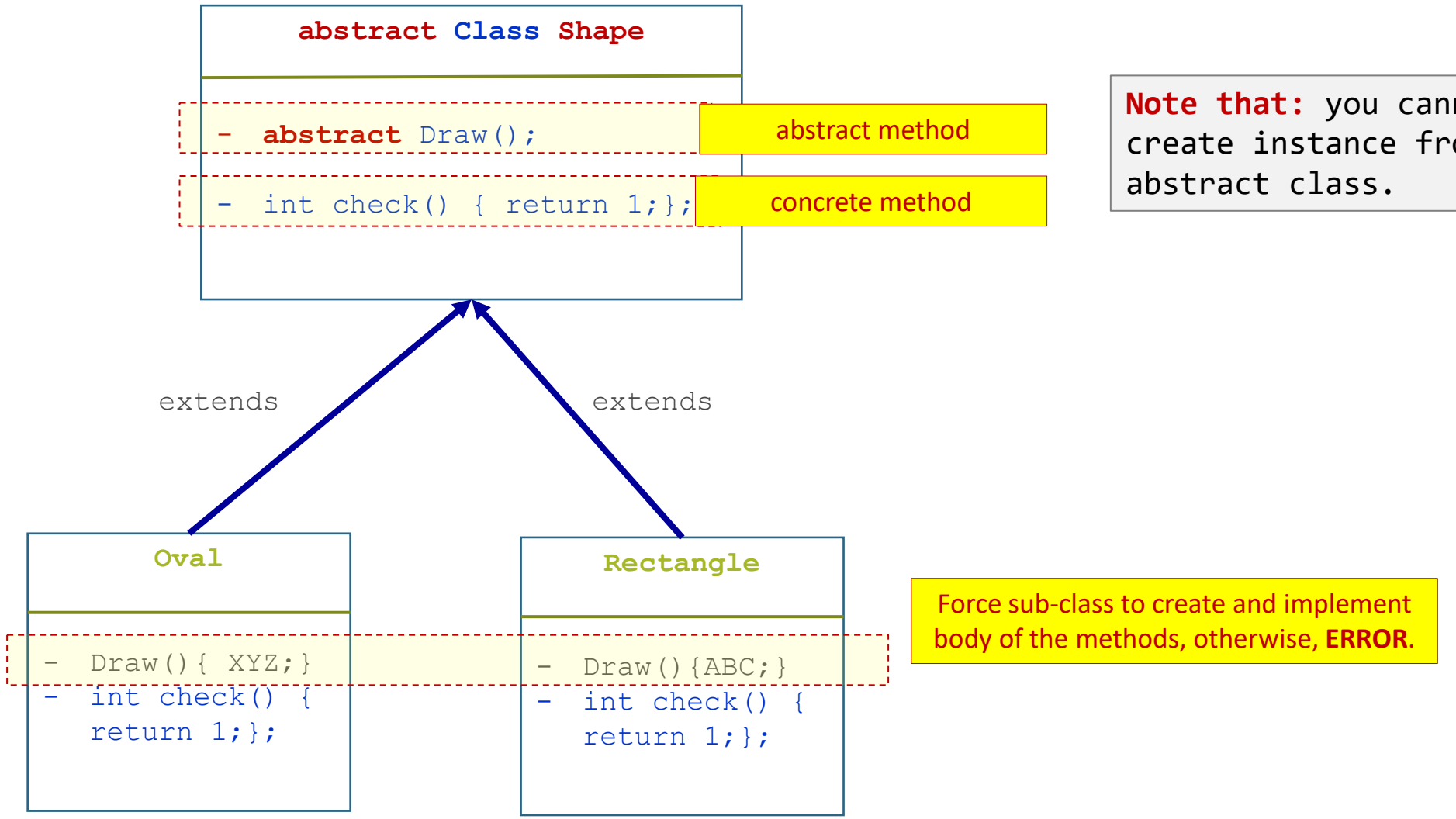
- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc.

- You know that Shape must have method "Draw()" and you don't know what should be implemented in the method (simply because you don't know what a `Shape` should look like in generic).

- **HOWEVER**, you want to force all sub class to implement this method (this method is so important :P).

- Thus, **Abstract Class** is used!!!

What is Abstract Class



Note that: you cannot create instance from abstract class.

Force sub-class to create and implement body of the methods, otherwise, **ERROR**.

What is abstract Class (Cont)

- Any class containing an abstract method is an abstract class.
- You must declare the class with the keyword abstract:

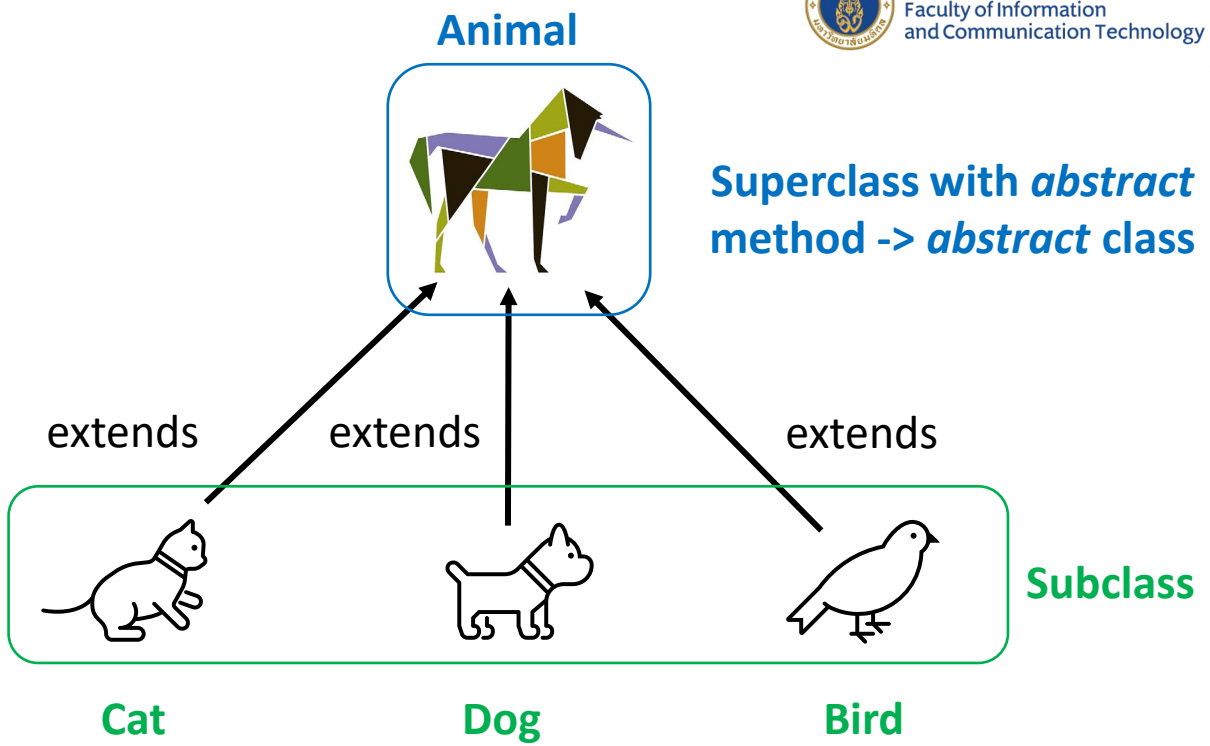
  abstract class MyClass {...}

- An abstract class is *incomplete*
  - It has “missing” method bodies
- You cannot instantiate (create a new instance of) an abstract class.

What is abstract Class (Cont)

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated
  - If the subclass does *not* define all the inherited abstract methods, it too must be abstract
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated
- In the abstract Class constructor will not be inherited, so you don’t need to create one (because you cannot create an object).

```java
public abstract class Animal {
    public String type;
    public String color;

    public Animal(String type, String color){
        this.type = type;
        this.color = color;
    }

    public void print() {
        System.out.println(color + " " + type);
    }

    public abstract void speak();
}
```



How about speak() method of Animal?

- The sound of general animal is UNKNOWN, so we cannot implement this speak() method.
- So, the speak() method become "abstract" as well as the Animal class become "abstract"
- Other subclasses that extend this Animal, MUST explicitly implement this speak() method.

```java
public void speak() {
    System.out.println("Meow!");
}
```

```java
public void speak() {
    System.out.println("Tweet!");
}
```

```java
public void speak() {
    System.out.println("Bark!");
}
```

```java
public class Dog extends Animal{

    public Do
        super
    }
```

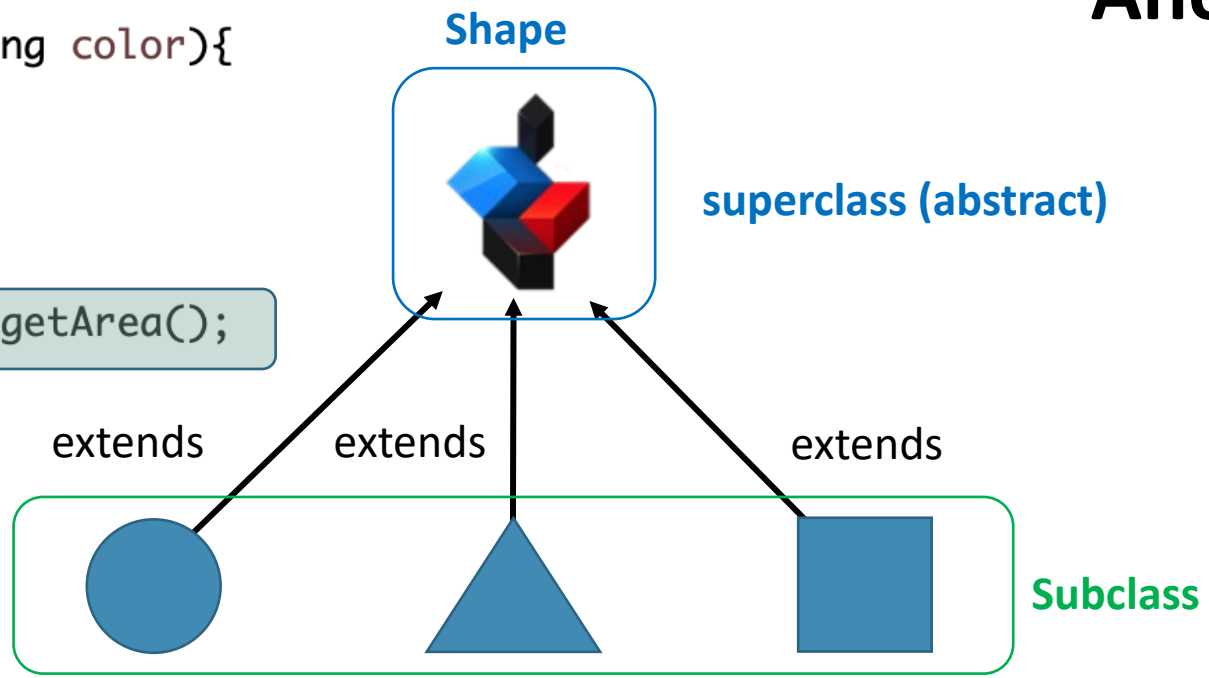
The type Dog must implement the inherited abstract method Animal.speak()

2 quick fixes available:

- Add unimplemented methods
- Make type 'Dog' abstract

Another Example of Abstract Class

```
public abstract class Shape {
    String type;
    String color;

    Shape(String type, String color){
        this.type = type;
        this.color = color;
    }

    public abstract double getArea();
}
```




```
public double getArea(){
    return 3.14 * r * r;
}
```

```
public double getArea(){
    return width * height;
}
```

```
public double getArea(){
    return 0.5 * base * height;
}
```

Abstract Method

- Appears in a **superclass**, but expects to be **overridden** in a subclass
- Notice that there is only a **method header**; **No method body**.
- See **abstract** key word, and **semicolon ;** at the end of method header.

```
public abstract double getArea();
```

- If a subclass fails to override the abstract method, an **error** will occur.
- Abstract methods are used to **ensure** that a subclass implements the method.

Abstract Class

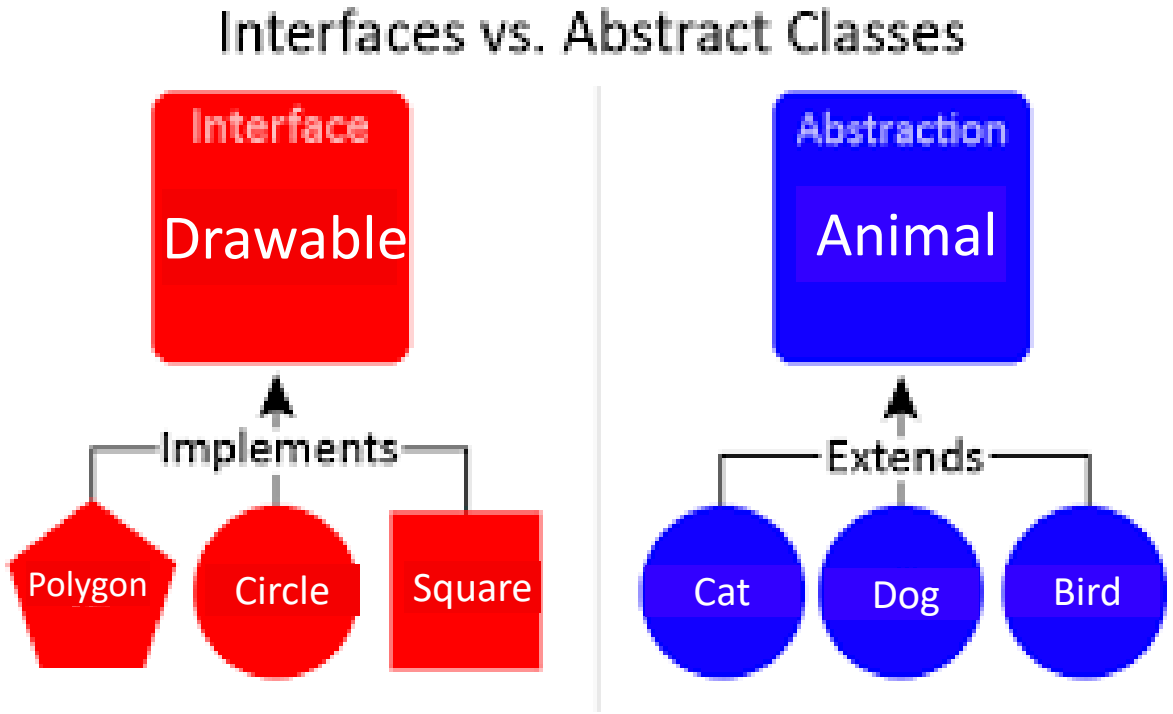
- A class contains an **abstract method**
- Declare the class with the keyword **abstract**

```
public abstract class Shape{
    . . .
    public void setColor(String c){
        this.color = c;
    }

    public String getColor(){
        return this.color;
    }

    public abstract double getArea();
}
```

- You **cannot instantiate** (create a new instance of) an abstract class

```
Shape var = new Shape();    // Error!
```

INTERFACE



To be able to declare and use interface types

To appreciate how interfaces can be used to decouple classes

Suppose, we have two classes

```java
public class BankAccount {
    private int id;
    private double balance;

    public BankAccount(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    // adding getter methods
    public int getId() {
        return id;
    }

    public double getBalance() {
        return balance;
    }
```

```java
public class Book {
    private String title;
    private double price;

    public Book(String title, double price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }
}
```

1. Using Interface for Algorithm Reuse

- Consider the average method from two different classes

```java
public static double average(BankAccount[] objects) {
  double sum = 0;
  for (BankAccount obj : objects) {
    sum = sum + obj.getBalance();
  }
  if (objects.length > 0) {
    return sum / objects.length;
  }
  else {
    return 0;
  }
}
```

```java
public static double average(Book[] objects) {
  double sum = 0;
  for (Book obj : objects) {
    sum = sum + obj.getPrice();
  }
  if (objects.length > 0) {
    return sum / objects.length;
  }
  else {
    return 0;
  }
}
```

- The algorithm for computing the average is the same in all cases, but the details of measurement differ. *>>> Can we provide a single method that does just this service?*

Calling `average(...)` methods (overload)

```
3  public class App {
4
5      public static void main(String[] args) {
6          BankAccount[] accounts = new BankAccount[3];
7          accounts[0] = new BankAccount(1, 100);
8          accounts[1] = new BankAccount(2, 200);
9          accounts[2] = new BankAccount(3, 300);
10         System.out.println("Average Balance: " + average(accounts));
11
12         Book[] books = new Book[3];
13         books[0] = new Book("Java Prog", 200);
14         books[1] = new Book("OOP Concept", 400);
15         books[2] = new Book("Python Wow", 600);
16         System.out.println("Average Price: " + average(books));
17
18     }
```

1.1 Interface Type

- The problem is those two classes use different way to get the value
  - BankAccount uses **getBalance(),** while Book uses **getPrice()**
- Suppose both BankAccount and Book class can agree on a method name to use in average algorithm, probably -> `getMeasure()`
- The problem still remains on the type of obj we cannot write
  `for(BankAccount or Book obj: objects)`
- So we need a **new type** that describes any class whose objects can be measured

```
public static double average(Book[] objects) {
  double sum = 0;
  for (Book obj : objects) {
    sum = sum + obj.getPrice();
  }
  if (objects.length > 0) {
    return sum / objects.length;
  }
  else {
    return 0;
  }
}
```

1.2 Declaring Interface Type

- We now declare Measurable interface type as follow

Syntax

```
public interface InterfaceName
{
   method headers
}
```

```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.

No implementation is provided.

```
// same as this statement
// public abstract double getMeasure();
```

- The interface declaration lists all methods that the interface type requires.
  - This example requires only one, but in general interface types require several methods.

Now, we can have a reusable average method

```
public static double average(Measurable[] objects) {
   double sum = 0;
   for (Measurable obj : objects) {
     sum = sum + obj.getMeasure();
   }
   if (objects.length > 0) {
     return sum / objects.length;
   }
   else {
     return 0;
   }
 }
```

- This method can be used for objects of any class that conforms to the **Measurable** type.
- Interface types make code more reusable!

This standmixer provide the “rotation” service to any attachment that conforms to a common interface.

1.3 Implementing an Interface Type

- A class implements an interface by adding “implements” clause as shown below

Syntax

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

```
public class BankAccount implements Measurable
{
   . . .
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

List all interface types that this class implements.

BankAccount instance variables

Other BankAccount methods

This method provides the implementation for the method declared in the interface.

```java
public class BankAccount2 implements Measurable {
    private int id;
    private double balance;

    public BankAccount2(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    // adding getter methods
    public int getId() {
        return id;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public double getMeasure() {
        return balance;
    }
}
```

```java
public interface Measurable {
    double getMeasure();
}
```

```java
// finding average measurement of any measurable objects
public static double average(Measurable[] objects) {
    double sum = 0;
    for(Measurable obj: objects) {
        sum = sum + obj.getMeasure();
    }
    if(objects.length > 0) {
        return sum / objects.length;
    }
    return 0;
}
```

```java
public interface Measurable {
    double getMeasure();
}
```

```java
public class BankAccount2 implements Measurable {
    private int id;
    private double balance;

    public BankAccount2(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    // adding getter methods
    public int getId() {
        return id;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public double getMeasure() {
        return balance;
    }
}
```

Exercise:
From the **`Measurable`** interface, how to make **`Book2`** class implements **`Measurable`** interface

You can use BankAccount2 as an example

```java
public class Book2 {
    private String title;
    private double price;

    public Book2(String title, double price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }
```

```java
public interface Measurable {
    double getMeasure();
}
```

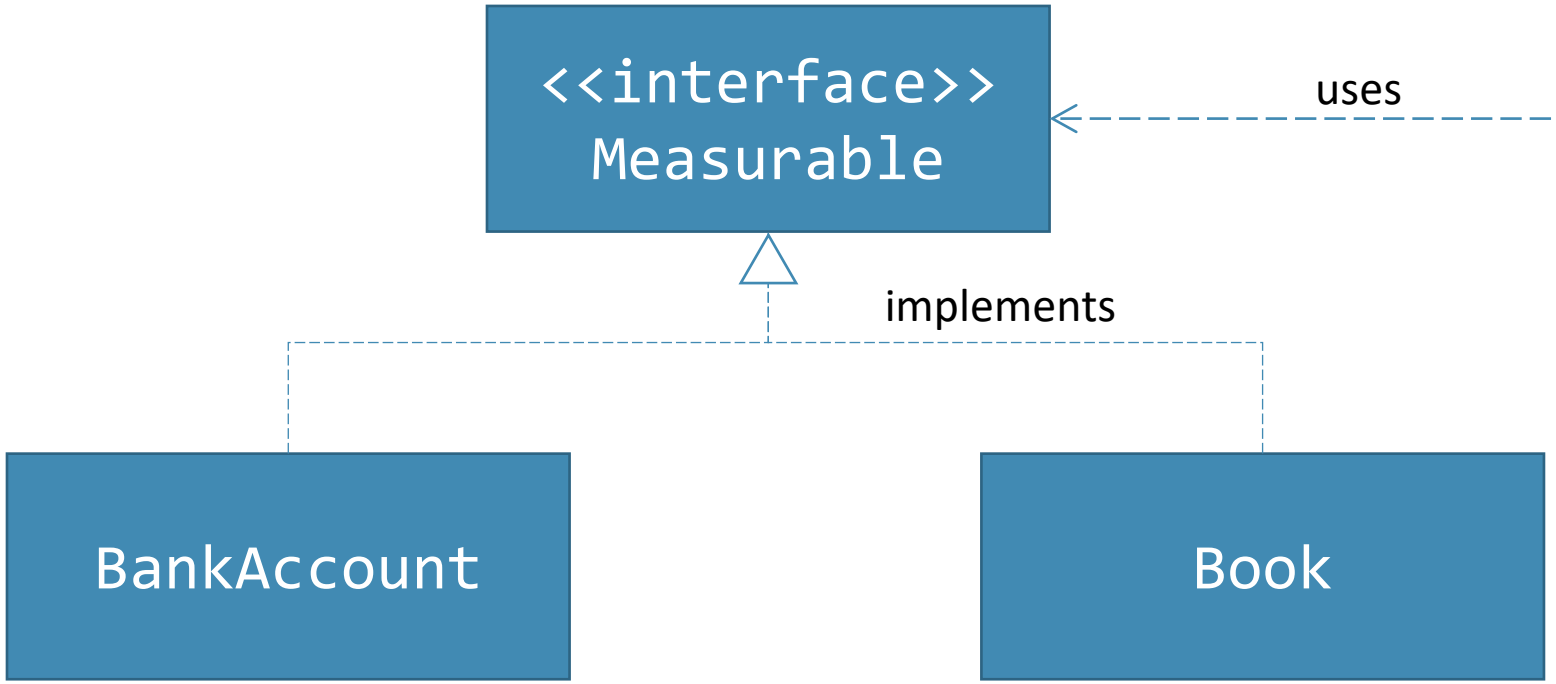
```java
public class BankAccount2 implements Measurable {
    private int id;
    private double balance;

    public BankAccount2(int id, double balance) {
        this.id = id;
        this.balance = balance;
    }

    // adding getter methods
    public int getId() {
        return id;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public double getMeasure() {
        return balance;
    }
}
```

Solution:

1) Add implements Measurable at the class header
2) Add a new method getMeasure() according to the Measurable interface

```java
public class Book2 implements Measurable {
    private String title;
    private double price;

    public Book2(String title, double price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }

    @Override
    public double getMeasure() {
        return price;
    }
}
```

UML Diagram



Suppose we write the `average` method in the `Data` class.

The `Data` class uses the `Measurable` type but not `BankAccount` jor `Book`

The `BankAccount` and `Book` classes implement the `Measurable` interface type

```java
public class Data {

    /*
     * finding average measures of any measurable objects
     * @param an array of Measurable objects
     * @return the average of the measures
     */
    public static double average(Measurable[] objects) {
        double sum = 0;
        for(Measurable obj: objects) {
            sum = sum + obj.getMeasure();
        }
        if(objects.length > 0) {
            return sum / objects.length;
        }
        return 0;
    }
}
```

```java
public class App2 {
    public static void main(String[] args) {
        BankAccount2[] accounts2 = new BankAccount2[3];
        accounts2[0] = new BankAccount2(1, 100);
        accounts2[1] = new BankAccount2(2, 200);
        accounts2[2] = new BankAccount2(3, 300);
        System.out.println("Average Measurement of BankAccount2: "
                + Data.average(accounts2));

        Book2[] books2 = new Book2[3];
        books2[0] = new Book2("Java Prog", 200);
        books2[1] = new Book2("OOP Concept", 400);
        books2[2] = new Book2("Python Wow", 600);
        System.out.println("Average Measurement of Book2: "
                + Data.average(books2));
    }
}
```

So, what is an Interface

An interface declares (describes) methods but does not supply bodies for them

```
public interface Skyability {

    public void fly();

}
```

```
public interface InterfaceName
{
    (Method headers . . .)
}
```

Notice that the keyword 'class' is replaced with 'interface' and all methods only have method headers

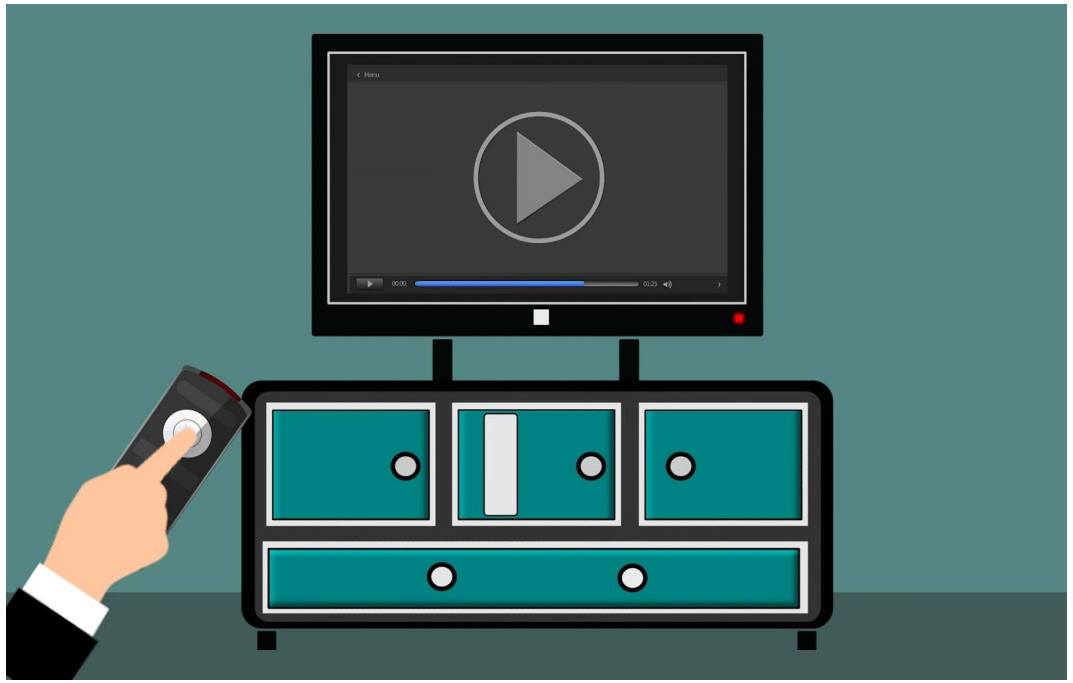
ALL the methods are implicitly `public` and `abstract`

You cannot instantiate an interface

- An interface is like a *very* abstract class—*none* of its methods are defined

An interface may also contain constants (`final` variables)

Real-world examples



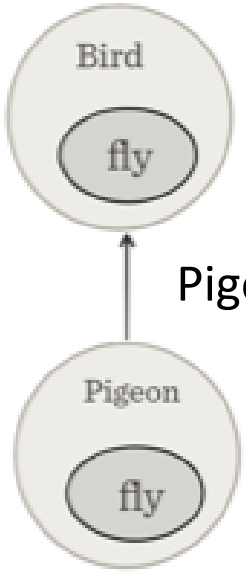
The buttons on the front of your television set are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

Class implements Interface

Lisa can choose to eat with her left hand or her right hand

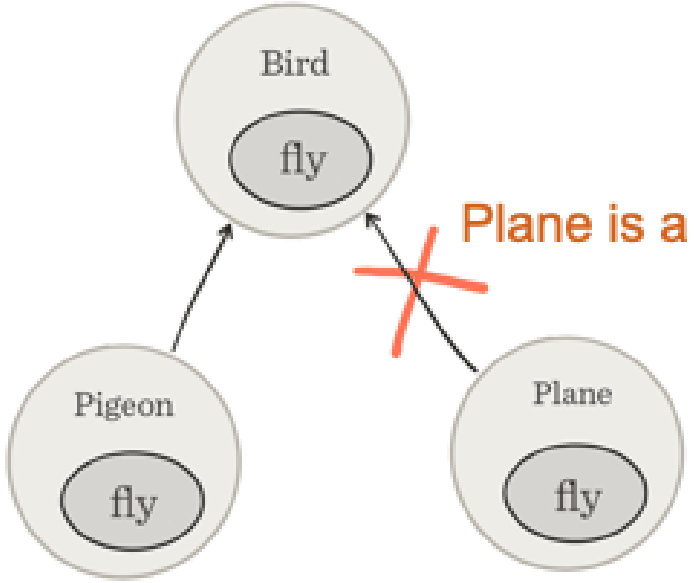
Interface

Lisa can eat!



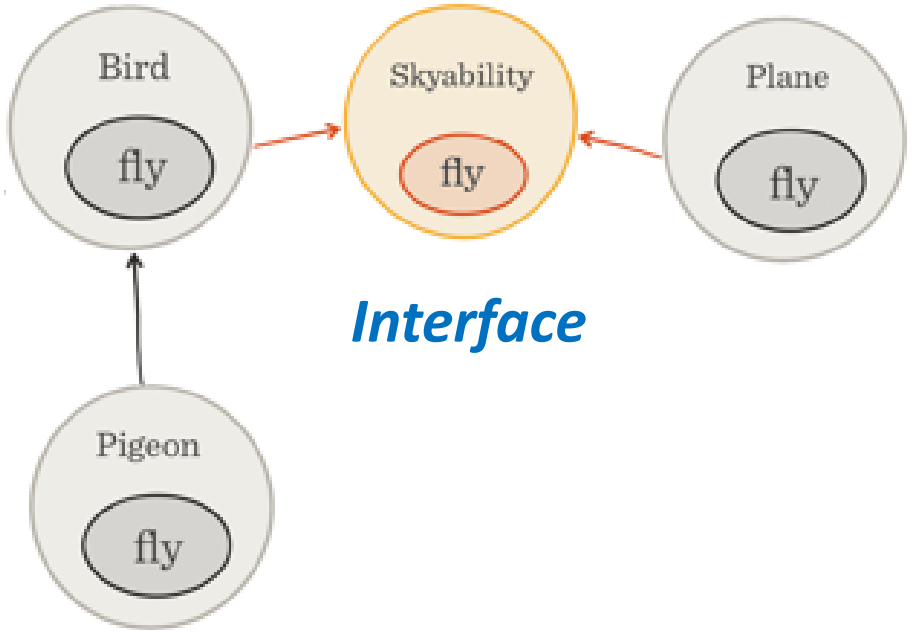
Inheritance

Pigeon **is a** Bird



2. Inheritance vs Interface

Bird **realizes** Skyability



Plane **realizes** Skyability

Interface

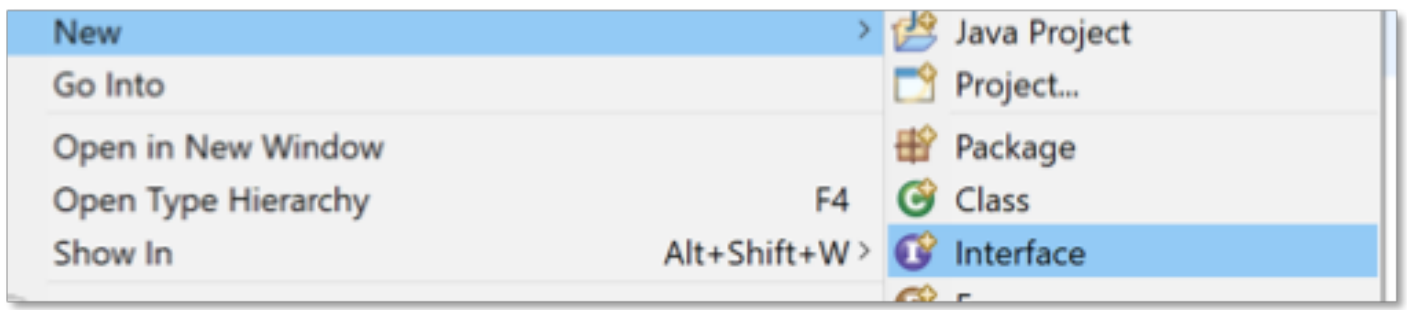
tzy.me/javainterface-42cbbd20bca6

2.1 Comparing Interfaces and Inheritance

- **Inheritance** model **"is-a" relationship** between objects
- While **interfaces** model some **common aspects** among objects.
  - Both BankAccount and Book can be measurable, but nothing else.
  - To model this common aspect enables other programmers to write tools that exploit the commonality (such as computing average)
- A class can **implement more than one interface**, but can **only extend (inherit from) one superclass**
- An interface usually specifies a behavior that an implementing class should supply. It **does not have any implementation**. While a **superclass provides some implementation** that a subclass can just simply inherit.

Interfaces

- An **interface** **must** have **ONLY** abstract methods
- An **interface** is similar to a class, expect the keyword **interface** is used instead of the keyword **class**
- In Eclipse, there is a menu to create **Interface**



Abstract Class

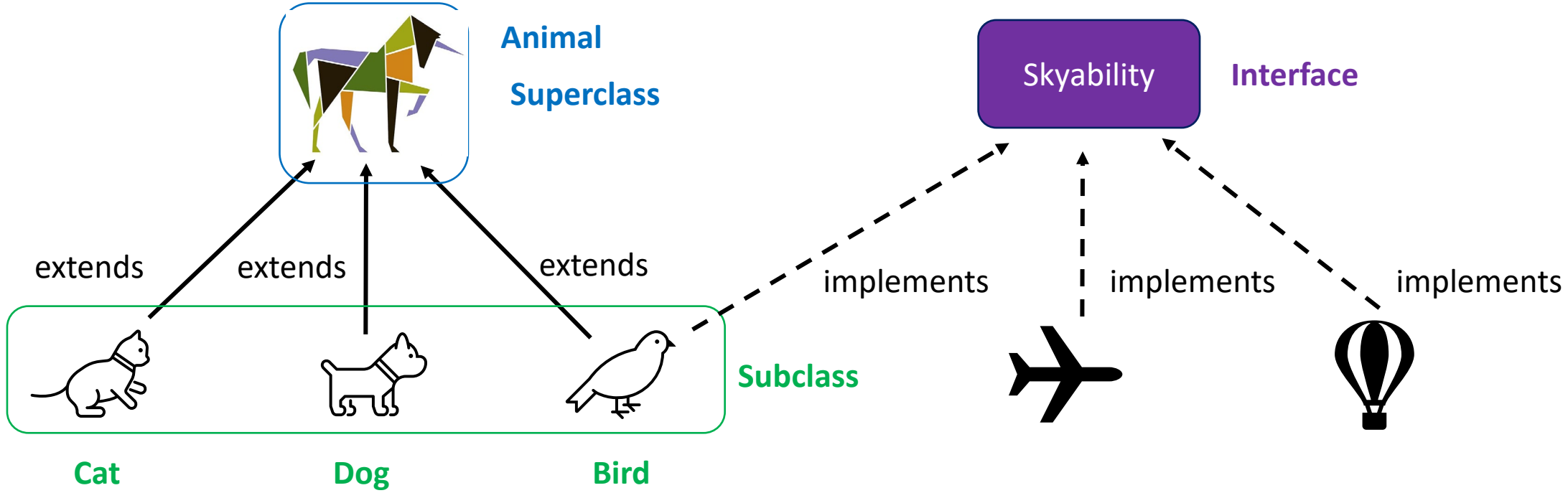
- An **abstract class** can have **both** abstract and non-abstract methods

```
public interface Relatable
{
    boolean equals(Shape t);
    boolean isSmaller(Shape t);
    boolean isBigger(Shape t);
}
```

*Notice that **no access specifier** and **abstract keyword** is used with the **method headers**, because all methods in the interface are **public** and **abstract** by default.*

```java
public abstract class Animal {
   public void print() {
      System.out.println(color + " " + type);
   }
   public abstract void speak();
}
```

```java
public interface Skyability {

   public abstract void fly();

}
```

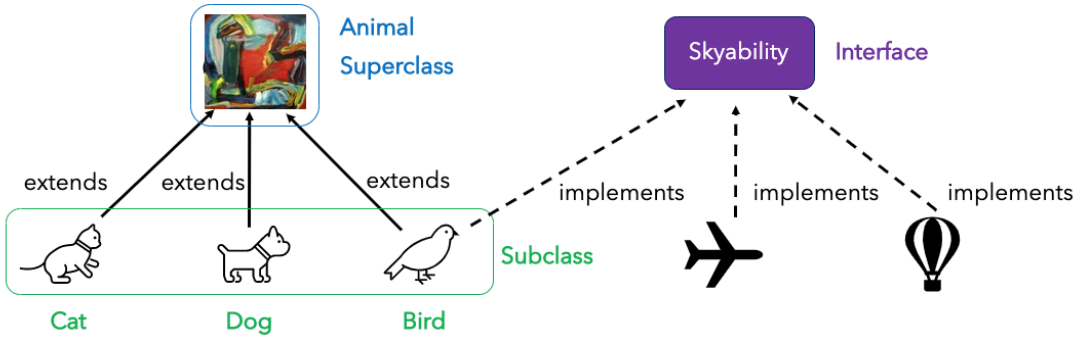



```java
public void speak() {
    System.out.println("Meow!");
}
```

```java
public void speak() {
    System.out.println("Tweet!");
}
```

```java
public void speak() {
    System.out.println("Bark!");
}
```

Question?
According to the Interface concept, the class **Bird**, **Airplane**, and **Balloon** must implement which method?




```java
public interface Skyability {

   public abstract void fly();
   // or void fly();
}
```

(methods in the interface are public and abstract by default)

```java
public class Bird extends Animal implements Skyability{

  public Bird(String color) {
    super("Bird", color);
  }


  @Override
  public void speak() {
    System.out.println("Tweet!");
  }


  @Override
  public void fly() {
    System.out.println("Flying with wings");
  }

}
```

```java
public class Airplane implements Skyability{

  @Override
  public void fly() {
    System.out.println("Flying with engines");
  }

}
```

```java
public class Balloon implements Skyability{

  @Override
  public void fly() {
    System.out.println("Flying with hot-air");
  }

}
```

Inherits multiple classes
- Not Allow -

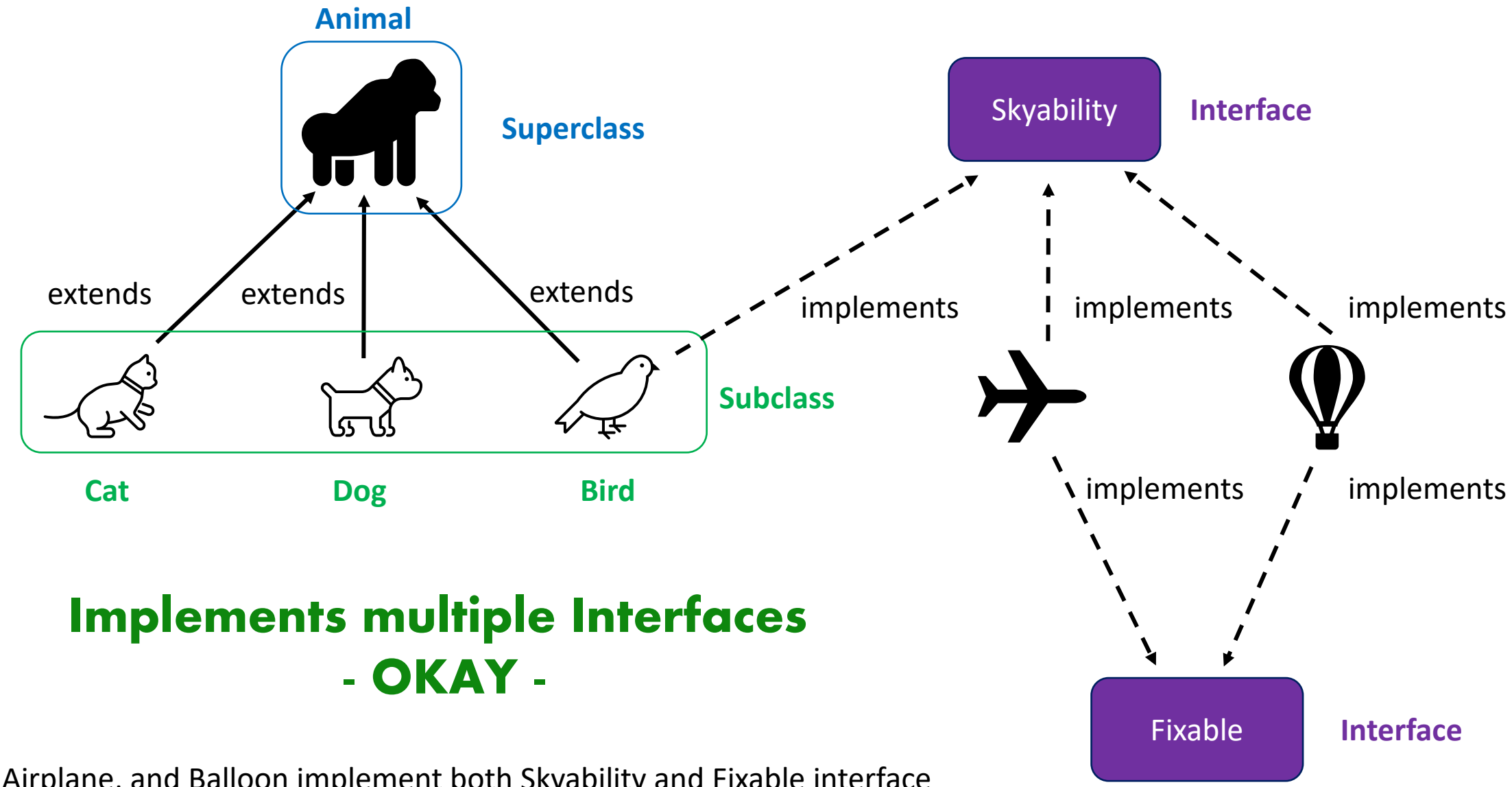
You can EAT but you must eat with your **LEFT** hand

You can EAT but you must eat with your **RIGHT** hand

Implements multiple Interfaces
- OKAY -

You can EAT!
How to eat is up to you.

You can EAT!
How to eat is up to you.



Implements multiple Interfaces
- OKAY -

Airplane, and Balloon implement both Skyability and Fixable interface

```java
public interface Skyability {

    public abstract void fly();
    // or void fly();
}
```

```java
public interface Fixable {
  final String ERROR = "Cannot fix";

  boolean fix();
}
```

```java
public class Airplane implements Skyability, Fixable{

    @Override
    public void fly() {
        System.out.println("Flying with engines");
    }

    @Override
    public boolean fix() {
        System.out.println("Fixing airplane");
        System.out.println(Fixable.ERROR);
        return false;
    }
}
```

```java
public class Balloon implements Skyability, Fixable{

    @Override
    public void fly() {
        System.out.println("Flying with hot-air");
    }

    @Override
    public boolean fix() {
        System.out.println("Fixing hot-air balloon");
        return true;
    }
}
```

Note that: If some of the required methods are not Overrides, the Class must be defined as an abstract class.

3. Comparison Table

	Class	Abstract Class	Interface
Instance Fields Variables	Yes	Yes	Yes Only final static variables
Constructor	Yes	Yes	No
Methods Body	Yes	Yes	No
Abstract Methods	Not allow	Zero or More	ALL
Able to instantiate object from this	Yes	No	No

```
Animal a = new Animal("animal","white"); // ERROR
Measurable m = new Measurable(100); // ERROR
```

4. Standard Java Library: `Comparable` Interface

- This interface involves two objects. To compare which object comes before another object.
- Comparable interface has a **`compareTo`** method -> `a.compareTo(b)`
  - Return a negative number if a should come before b,
  - Return zero (0) if a and b are the same,
  - Return a positive number if b should come before a
- If a class implements **`Comparable`** interface, you can use standard methods (that need comparable ability) such as **`Arrays.sort`**

```java
public class BankAccount implements Measurable, Comparable {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    // implement compareTo method from Comparable interface
    public int compareTo(Object otherObject) {
      BankAccount other = (BankAccount) otherObject;
      if (balance < other.balance) { return -1; }
      if (balance > other.balance) { return 1; }
      return 0;
    }

    ...
}
```

```java
public static void main(String[] args) {
    BankAccount[] accounts = new BankAccount[3];
    accounts[0] = new BankAccount("001", 300);
    accounts[1] = new BankAccount("002", 100);
    accounts[2] = new BankAccount("003", 200);

    for(int i = 0; i < accounts.length; i++){
      System.out.println(accounts[i]);
    }
    Arrays.sort(accounts);
    System.out.println("---- After Sorting ----");
    for(int i = 0; i < accounts.length; i++){
      System.out.println(accounts[i]);
    }
}
```

OUTPUT:

accNumber: 001, balance: 300.0
accNumber: 002, balance: 100.0
accNumber: 003, balance: 200.0
---- After Sorting ----
accNumber: 002, balance: 100.0
accNumber: 003, balance: 200.0
accNumber: 001, balance: 300.0

Checkpoint: Which statements cause ERROR?

- Suppose there are two classes and two interfaces as follow:
- `public class ClassA`
- `public abstract class ClassB`
- `public interface InterfaceC`
- `public interface InterfaceD`

Class Declaration

```
a) public class X extends ClassA
b) public class Y extends ClassB
c) public class Z implements InterfaceC
d) public class AC extends ClassA implements InterfaceC
e) public class AB extends ClassA, ClassB
f) public class CD implements InterfaceC, InterfaceD
```

Instantiate Objects

```
1) ClassA var = new ClassA();
2) ClassB var = new ClassB();
3) InterfaceC var = new InterfaceC();
4) ClassA var = new X();
5) ClassB var = new Y();
6) InterfaceC var = new CD();
```

Checkpoint: Which statements cause ERROR?

- Suppose there are two classes and two interfaces as follow:
- `public class ClassA`
- `public abstract class ClassB`
- `public interface InterfaceC`
- `public interface InterfaceD`

Class Declaration

```
a) public class X extends ClassA
b) public class Y extends ClassB
c) public class Z implements InterfaceC
d) public class AC extends ClassA implements InterfaceC
e) public class AB extends ClassA, ClassB
f) public class CD implements InterfaceC, InterfaceD
```

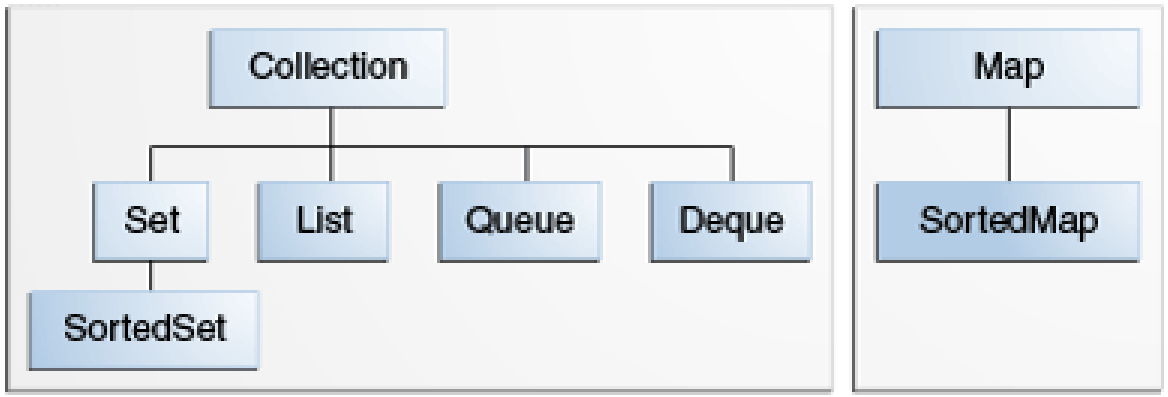
Instantiate Objects

```
1) ClassA var = new ClassA();
2) ClassB var = new ClassB();
3) InterfaceC var = new InterfaceC();
4) ClassA var = new X();
5) ClassB var = new Y();
6) InterfaceC var = new CD();
```

Answer:
Statements e, 2, and 3

Summary

- An abstract method is a method without implementation.
- An abstract class can contain abstract methods.
- An interface contains **only** abstract methods.
  - No need to put “abstract” in front of each method signature.



Java Collection

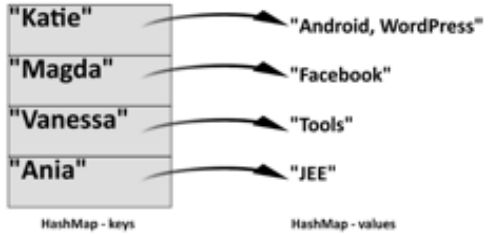
List, Set, Map

1. Java Collection

- A **`Collection`** (also known as container) is an object that contains a group of objects treated as a single unit
  - e.g., ArrayList of Dog

```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```

dogs = 



- Any type of objects can be stored, retrieved, and manipulated as elements of collections.

  - e.g.,

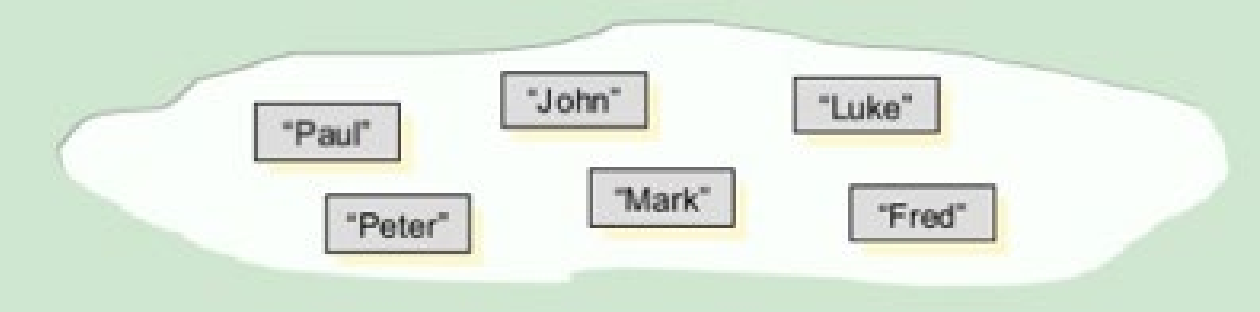
```
dogs.add(new Dog(12, "black"));
dogs.get(0);
```

Review Collection

List: Lists of things (classes that implement List)

* cares about the index

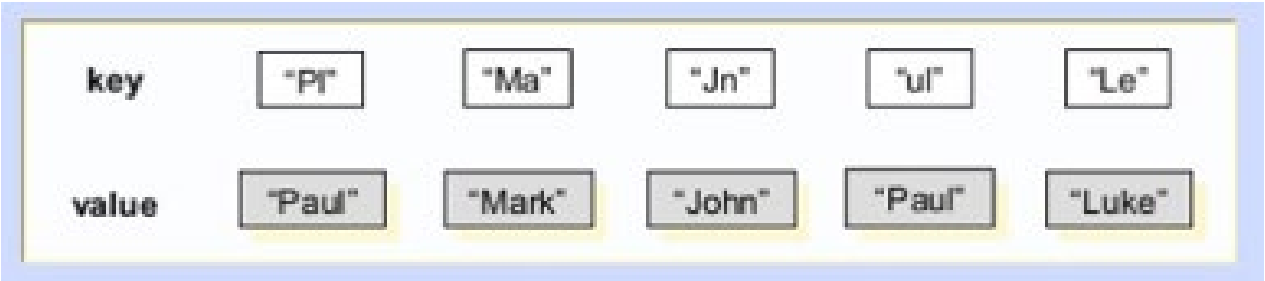
e.g., **ArrayList, Vector, LinkedList**



Set: Unique things (classes that implement Set)

* cares about uniqueness, no duplicate

e.g., **HashSet, LinkedHashSet, TreeSet**

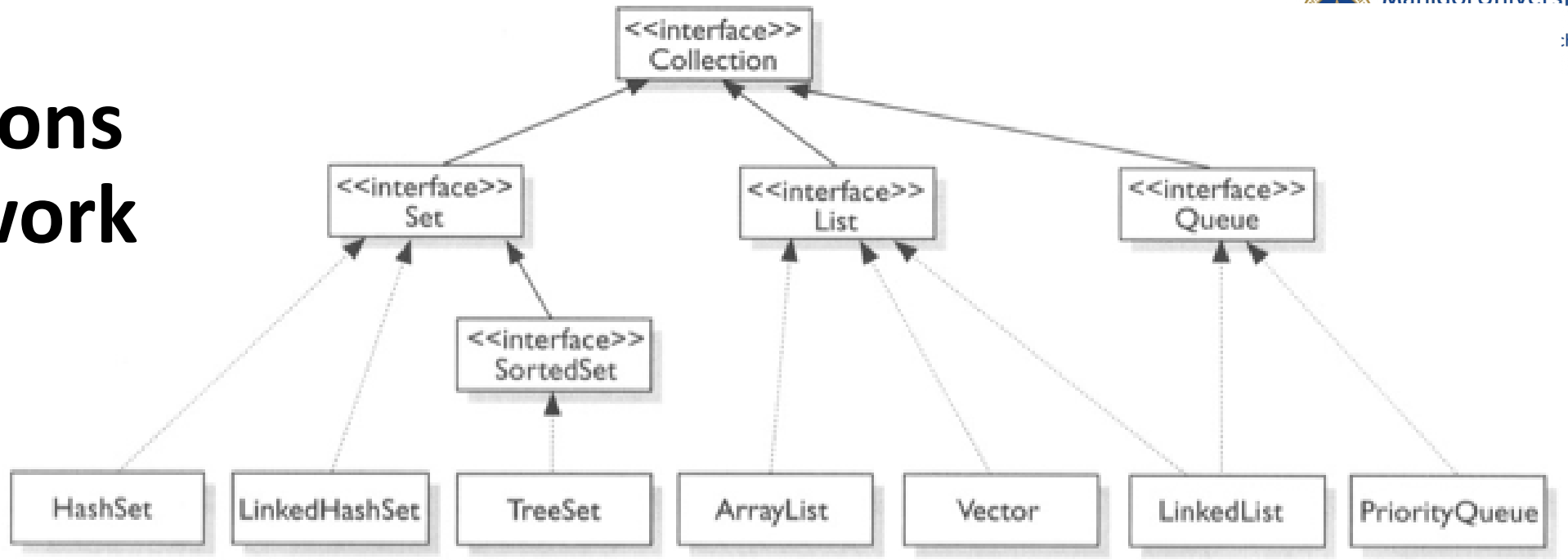


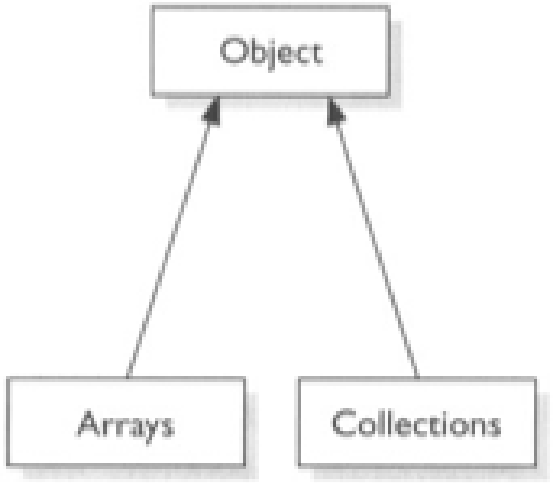
Map: Things with a unique ID (classes that implement Map)

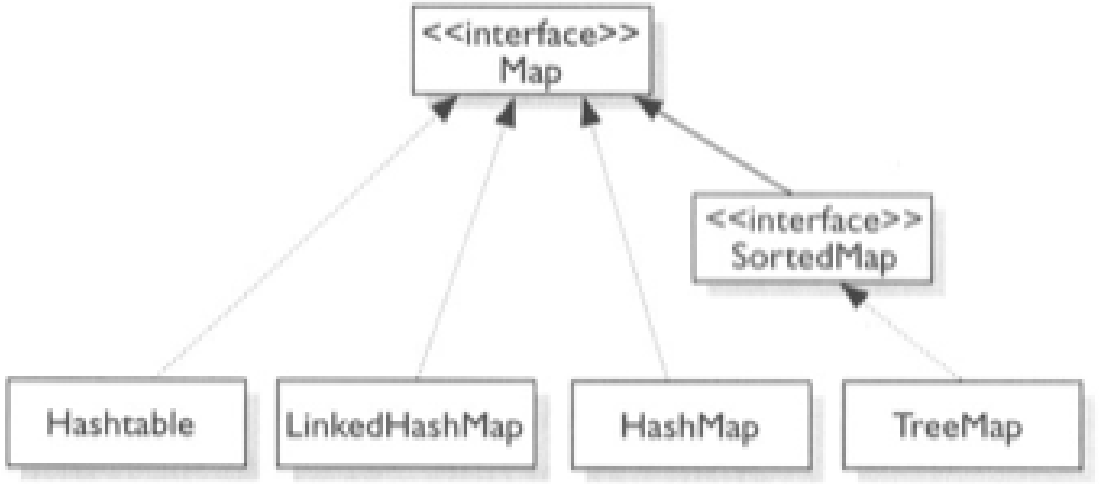
* cares about unique identifiers (key-value pair)

e.g., **HashMap, HashTable, TreeMap**

Java Collections Framework

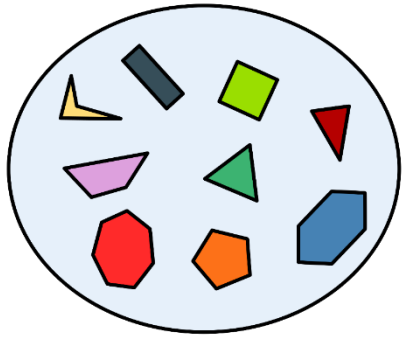






Sets

What is Sets



What is Sets Interface

- A Set is unordered and has no duplicates data structure.
- Operations are exactly those for Collection.
- All provided under java collection framework.

List of Common Methods in Collection

```
int size( );
boolean isEmpty( );
boolean contains(Object e);
boolean add(Object e);
boolean remove(Object e);
Iterator iterator( );
```

```
boolean containsAll(Collection c);
boolean addAll(Collection c); boolean
removeAll(Collection c);
boolean retainAll(Collection c);
void clear( );
```

```
Object[ ] toArray( );
Object[ ] toArray(Object a[ ]);
```

Sets implementation

- Set is an interface; you can't say new Set( )

- There are four implementations:
  - HashSet is best for most purposes
  - TreeSet guarantees that an iterator will return elements in sorted order
  - LinkedHashSet guarantees that guarantees that an iterator will return elements in the order they were inserted
  - AbstractSet is a "helper" abstract class for new implementations

Typical Operation of Sets

- Testing if s2 is a *subset* of s1
  s1.containsAll(s2)

- Setting s1 to the *union* of s1 and s2
  s1.addAll(s2)

- Setting s1 to the *intersection* of s1 and s2
  s1.retainAll(s2)

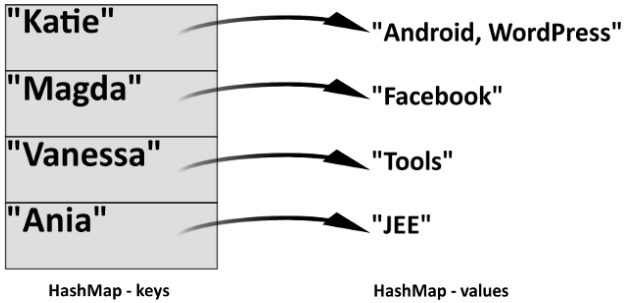
- Setting s1 to the *set difference* of s1 and s2
  s1.removeAll(s2)

Sets Equality

- `Object.equals(Object)`, inherited by all objects, really is an *identity* comparison
- Implementations of `Set` override `equals` so that sets are equal if they contain the same elements
- `equals` even works if two sets have different implementations
- `equals` is a test on entire sets; you have to be sure you have a working `equals` on individual set elements
- `hashCode` has been extended similarly
  - This is for *sets*, not *elements* of a collection!

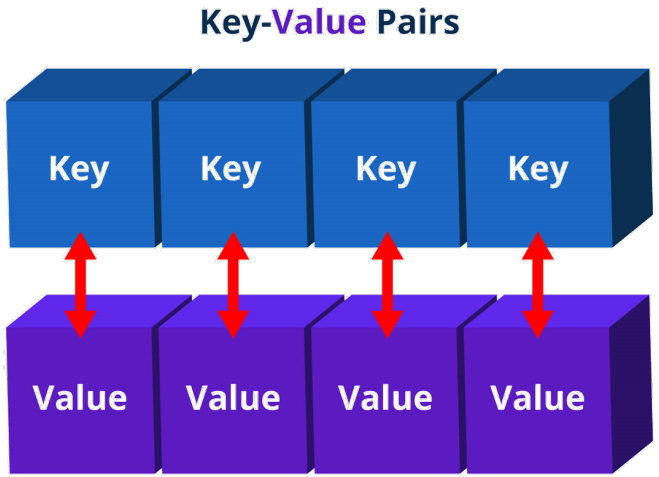
Maps

What is Maps



What is Maps Interface

- A Map is an object that maps keys to values
- A map cannot contain duplicate keys
- Each key can map to at most one value
- Examples: dictionary, phone book, etc.



- Map is an interface; you can't say new Map( )
- Here are two implementations:
  - HashMap is the faster
  - TreeMap guarantees the order of iteration

Basic Operation

```
Object put(Object key, Object value);

Object get(Object key);

Object remove(Object key);

boolean containsKey(Object key);

boolean containsValue(Object value);

int size( );

boolean isEmpty( );
```

Adding Element with put()

- If the map already contains a given key,
  `put(key, value)` replaces the value associated with that key
- This means Java has to do equality testing on keys
- With a `HashMap` implementation, you need to define `equals` and `hashCode` for all your keys
- With a `TreeMap` implementation, you need to define `equals` and implement the `Comparable` interface for all your keys
- `void putAll(Map t);`
  - Copies one `Map` into another Example: `newMap.putAll(oldMap);`
- `void clear();`
  - Example: `oldMap.clear();`

Example of using Maps

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {

        Map<String, String> fruit = new HashMap<String, String>();

        fruit.put("Apple", "red");
        fruit.put("Pear", "yellow");
        fruit.put("Plum", "purple");
        fruit.put("Cherry", "red");


        for (String key : fruit.keySet()) {
            System.out.println(key + ": " + fruit.get(key));
        }
    }
}
```

Things you should know when working with Java Collection (List, Set, Map)

ArrayList Syntax

- Add/Insert new element
- Retrieve (get) element
- Remove element
- Size / is it empty?
- Iterate (loop through all elements)
- Find specific element

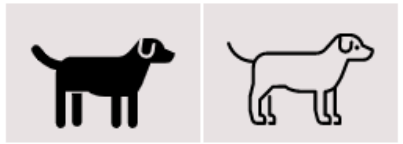
- dogs.add(new Dog(12, "black");
- dogs.get(0);
- dogs.remove(0); Index or dogObject
- dogs.size(); dogs.isEmpty()
- for(Dog d: dogs){ /* do s.th */ };
- dogs.contains(dogObject);

ArrayList Example – Using INDEX

Mahidol University
Faculty of Information and Communication Technology

```java
import java.util.ArrayList;

public class DogListTester {

    public static void main(String[] args) {
        ArrayList<Dog> dogs = new ArrayList<Dog>();
        Dog myDog = new Dog(12, "black");
        dogs.add(myDog);
        dogs.add(new Dog(10, "white"));

        System.out.println("Size: " + dogs.size());
        Dog dog0 = dogs.get(0); // get dog at specific index
        System.out.println("Dog'color at index 0: " + dog0.getColor());
        System.out.println("Contains myDog:" + dogs.contains(myDog));

        System.out.println("\nShow color of all dogs");
        for(Dog d: dogs) {
            System.out.println(d.getColor());
        }

        System.out.println("\n-- Remove dog at index 0 --");
        dogs.remove(0);     // remove at specific index
        System.out.println("Size: " + dogs.size());
        System.out.println("Contains myDog:" + dogs.contains(myDog));
    }

}
```

dogs =



[0] [1]

```
Size: 2
Dog'color at index 0: black
Contains myDog:true

Show color of all dogs
black
white

-- Remove dog at index 0 --
Size: 1
Contains myDog:false
```

```java
class Dog {
    int age;
    String color;

    Dog(int age, String color){
        this.age = age;
        this.color = color;
    }

    String getColor() {
        return this.color;
    }

}
```

Set -> HashSet

- Add/Insert new element
- Retrieve (get) element
- Remove element
- Size / is it empty?
- Iterate (loop through all elements)
- Find specific element

HashSet Syntax

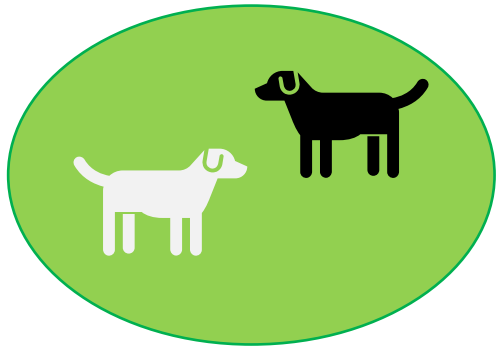
- dogs.add(new Dog(12, "black");
- Cannot directly get by index
- dogs.remove(dogObject);
- dogs.size(); dogs.isEmpty();
- for(Dog d: dogs) { /* do s.th */ };
- dogs.contains(dogObject);

HashSet Example – NO INDEX

Mahidol University
Faculty of Information and Communication Technology

```
import java.util.HashSet;

public class DogSetTester {

    public static void main(String[] args) {
        HashSet<Dog> dogs = new HashSet<Dog>();
        Dog myDog = new Dog(12, "black");
        dogs.add(myDog);
        dogs.add(new Dog(10, "white"));

        System.out.println("Size: " + dogs.size());
        System.out.println("Contains myDog:" + dogs.contains(myDog));

        System.out.println("Show color of all dogs");
        for(Dog d: dogs) {
            System.out.println(d.getColor());
        }

        System.out.println("-- Remove myDog --");
        dogs.remove(myDog); // unlike, ArrayList, you cannot use index
        System.out.println("Size: " + dogs.size());
        System.out.println("Contains myDog:" + dogs.contains(myDog));
    }
}
```

dogs =



```
Size: 2
Contains myDog:true

Show color of all dogs
black
white

-- Remove myDog --
Size: 1
Contains myDog:false
```

```
class Dog {
    int age;
    String color;

    Dog(int age, String color){
        this.age = age;
        this.color = color;
    }

    String getColor() {
        return this.color;
    }

}
```

Map -> HashMap

- Add/Insert new element
- Retrieve (get) element
- Remove element
- Size / is it empty?
- Iterate (loop through all elements)
- Find specific element

HashMap Syntax

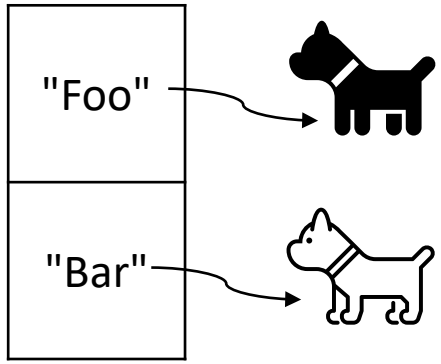
- dogs.put("key", new Dog(12, "black");
- dogs.get("key");
- dogs.remove("key");
- dogs.size(); dogs.isEmpty();
- **Loop using keySet(); method**
- dogs.containsKey("key"); // OR
  dogs.containsValue(dogObject);

HashSet Example – Using KEY

Mahidol University
Faculty of Information and Communication Technology

```
import java.util.HashMap;
import java.util.Set;

public class DogMapTester {

    public static void main(String[] args) {
        HashMap<String, Dog> dogs = new HashMap<String,Dog>();
        Dog myDog = new Dog(12, "black");
        dogs.put("Foo", myDog);
        dogs.put("Bar", new Dog(10, "white"));

        System.out.println("Size: " + dogs.size());
        System.out.println("Contains dog's name Foo:"
                           + dogs.containsKey("Foo"));

        System.out.println("\nShow all keys in dogs map");
        Set<String> keys = dogs.keySet();
        for(String k: keys) {
            System.out.println("Key in dogs.keySet() " + k);
        }

        System.out.println("\nShow color of all dogs");
        for(String key: dogs.keySet()) {
            Dog d = dogs.get(key);
            System.out.println(d.getColor());
        }

        System.out.println("\n-- Remove myDog --");
        dogs.remove("Foo"); // remove with specific key
        System.out.println("Size: " + dogs.size());
        System.out.println("Contains myDog:" + dogs.containsKey("Foo"));
    }
}
```

dogs =



Key (String) Value (Dog)

```
Size: 2
Contains dog's name Foo:true

Show all keys in dogs map
Key in dogs.keySet() Bar
Key in dogs.keySet() Foo

Show color of all dogs
white
black

-- Remove myDog --
Size: 1
Contains myDog:false
```

```
class Dog {
    int age;
    String color;

    Dog(int age, String color){
        this.age = age;
        this.color = color;
    }

    String getColor() {
        return this.color;
    }

}
```

2. Common Ways to Traverse a Collection

- **Normal For Loop**

```
for(int i=0; i < objects.size(); i++) { . . . }
```

- **For-Each Loop**

```
for(Object obj: objects) { . . . }
```

- **Iterator & While Loop**

```
Iterator<Object> it = objects.iterator();
while(it.hasNext()) { . . . }
```

- **`forEach()` method**

```
objects.forEach(obj -> { . . . } );
```

Example Collection of Dogs

```
class Dog {
  int age;
  String color;

  Dog(int age, String color){
    this.age = age;
    this.color = color;
  }

  public String getColor() {
    return this.color;
  }

  public String toString(){
    return "age: " + age + ", color: " + color;
  }
}
```

```
Dog myDog = new Dog(12, "black");

ArrayList<Dog> dogList = new ArrayList<Dog>();
dogList.add(myDog);
dogList.add(new Dog(10, "white"));

HashSet<Dog> dogSet = new HashSet<Dog>();
dogSet.add(myDog);
dogSet.add(new Dog(10, "white"));

HashMap<String, Dog> dogMap = new HashMap<String,Dog>();
dogMap.put("Foo", myDog);
dogMap.put("Bar", new Dog(10, "white"));
```

2.1 For-Each Loop

- Using for-loop to get each element from a collection without using index

```
//ArrayList
for(Dog dog: dogList){
  System.out.println("List => " + dog);
}

// Set
for(Dog dog: dogSet){
  System.out.println("Set => " + dog);
}

// Map -> have to loop through keySet() instead
for(String key: dogMap.keySet()){
  System.out.println("Map => key: " + key + ", value " + dogMap.get(key));
}
```

OUTPUT (The order of elements can be different)

```
List => age: 12, color: black
List => age: 10, color: white
Set => age: 12, color: black
Set => age: 10, color: white
Map => key: Bar, value age: 10, color: white
Map => key: Foo, value age: 12, color: black
```

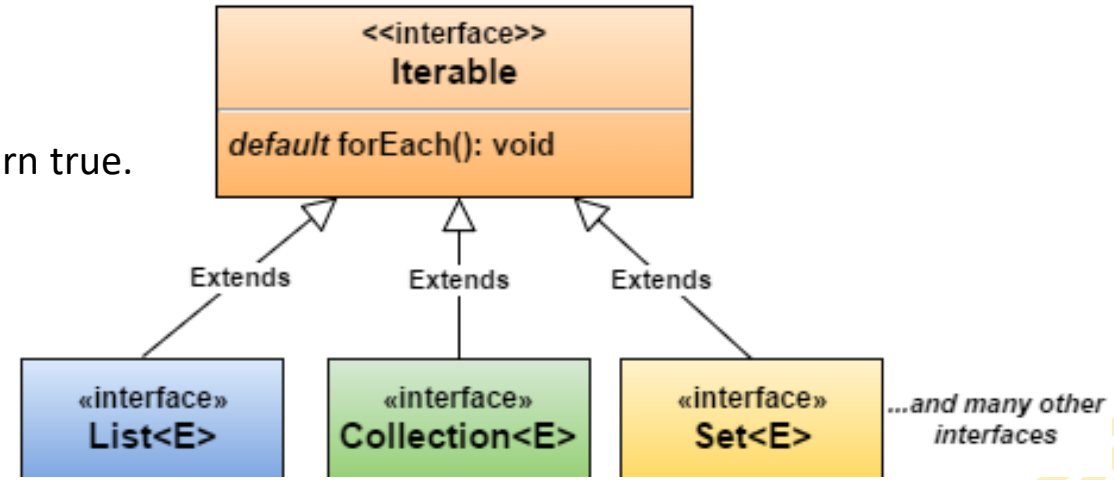
2.2 Iterator and Iterable Interface

- **Iterator** is a Java cursor used to traverse a Collection or Objects' elements.
- It allows to simply read and remove element in the Collection.
- Iterator object can be created by calling `iterator()` method in the Collection Interface.

```
Iterator cursor = obj.iterator();
```

obj represents any Collection objects such as List, Map, Set.

- Methods of Iterator object
  - **`hasNext()`** : check whether next element exist then return true.
  - **`next()`** : return the next element in the iteration.
  - **`remove()`** : remove the next element in the iteration.



Note that: All JAVA Collections implement Interface **Iterable.**

Java

Example Iterator: List vs Set vs Map

```
Iterator<Dog> cursorList = dogList.iterator();

while(cursorList.hasNext()) {
  System.out.println("List => " + cursorList.next());
}
```

```
Iterator<Dog> cursorSet = dogSet.iterator();

while(cursorList.hasNext()) {
  System.out.println("Set => " + cursorList.next());
}
```

Move cursor to next element

```
// For Map, we cannot get value (Dog) directly

// 1. Getting a Set of key-value pairs
Set entrySet = dogMap.entrySet();

// 2. Obtaining an interator for the entry set (key-value)
Iterator<Map.Entry<String, Dog>> it = entrySet.iterator();

while(it.hasNext()) {
  Map.Entry mapElement = it.next();
  System.out.println("Map => key: " + mapElement.getKey() +
                     ", value " + mapElement.getValue());
}
```

Map.Entry<key, value>
In this example, key is String and value is Dog.

mapElement.getKey() -> return key
mapElement.getValue() -> return value

OUTPUT
Map => key: Bar, value age: 10, color: white
Map => key: Foo, value age: 12, color: black

Dog Class

```
public String toString(){
  return "age: " + age + ", color: " + color;
}
```

2.3 forEach() method

- This a new and concise way to iterate over a collection in Java 8

```
void forEach(Consumer<? super T> action)
```

- Javadoc states that this method performs the given action for each element of the *Iterable* until all elements have been processed or the action throws an exception.
- For example, loop over String collections

For-each loop

```
for (String name : names) {
    System.out.println(name);
}
```

forEach() method

```
names.forEach(name -> {
    System.out.println(name);
});
```

Example forEach(): List vs Set vs Map

```
// ArrayList
dogList.forEach(dog -> {
   System.out.println("List => " + dog);
});

// Set
dogSet.forEach(dog -> {
   System.out.println("Set => " + dog);
});

// Map
dogMap.forEach((k, v)-> {
   System.out.println("key: " + k + ", value " + v);
});
```

OUTPUT

```
List => age: 12, color: black
List => age: 10, color: white

Set => age: 12, color: black
Set => age: 10, color: white

key: Bar, value age: 10, color: white
key: Foo, value age: 12, color: black
```

Try creating a list of animal with these elements

```
List<String> animal = new ArrayList<>();
animal.add("Cat");
animal.add("Dog");
animal.add("Bird");
animal.add("Ant");
```

1. How to navigate through all elements ?

1

```
for(String e: animal) {
  System.out.println(e);
}
```

2

```
Iterator<String> animalCursor = animal.iterator();
while(animalCursor.hasNext()) {
    System.out.println(animalCursor.next());
}
```

2. How to read and remove all element at the same time

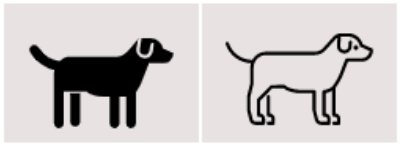
1

```
for(String e: animal) {
   System.out.println(e);
   animal.remove(e);
}
```

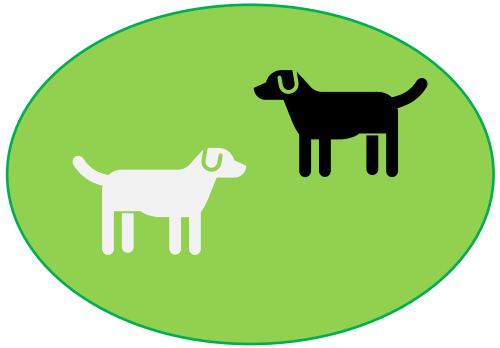
2

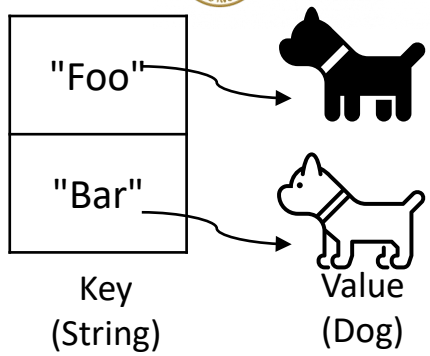
```
Iterator<String> animalCursor = animal.iterator();
while(animalCursor.hasNext()) {
     System.out.println(animalCursor.next());
     animalCursor.remove();
}
```

dogs =



[0] [1]





Summary

	ArrayList	Set (HashSet)	Map (HashMap)
Add/Insert new element	`dogs.add(new Dog(12, "black"));`	`dogs.add(new Dog(12, "black"));`	`dogs.put("key", new Dog(12, "black"));`
Retrieve element	`dogs.get(0); // index`	`Cannot directly get by index`	`dogs.get("key"); // key`
Remove element	`dogs.remove(0); // index`	`dogs.remove(dogObject); // Object`	`dogs.remove("key"); // key`
Size / is it empty?	`dogs.size(); dogs.isEmpty()`	`dogs.size(); dogs.isEmpty();`	`dogs.size(); dogs.isEmpty();`
Iterate (loop through all elements)	`for(Dog d: dogs) { /* do s.th */ };`	`for(Dog d: dogs) { /* do s.th */ };`	`Loop using keySet(); method`
Find specific element	`dogs.contains(dogObject);`	`dogs.contains(dogObject);`	`dogs.containsKey("key"); // OR dogs.containsValue(dogObject);`