Description

This week we will learn how to define common behavior for unrelated classes by writing an interface.

Interfaces

An interface is a class like construct that contains only constants and abstract methods. In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects, even if objects are alike.

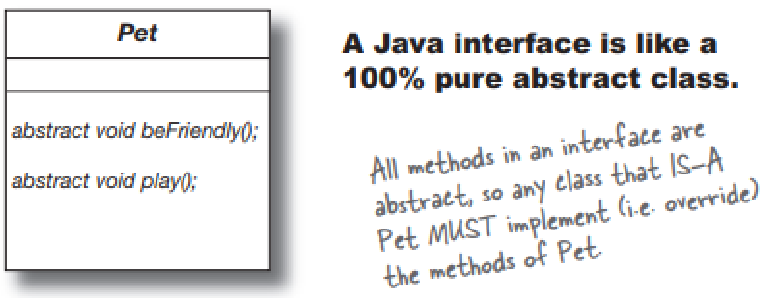
Java syntax to define an interface:

Access specifier/modifier **interface** InterfaceName{

/* constant declarations */

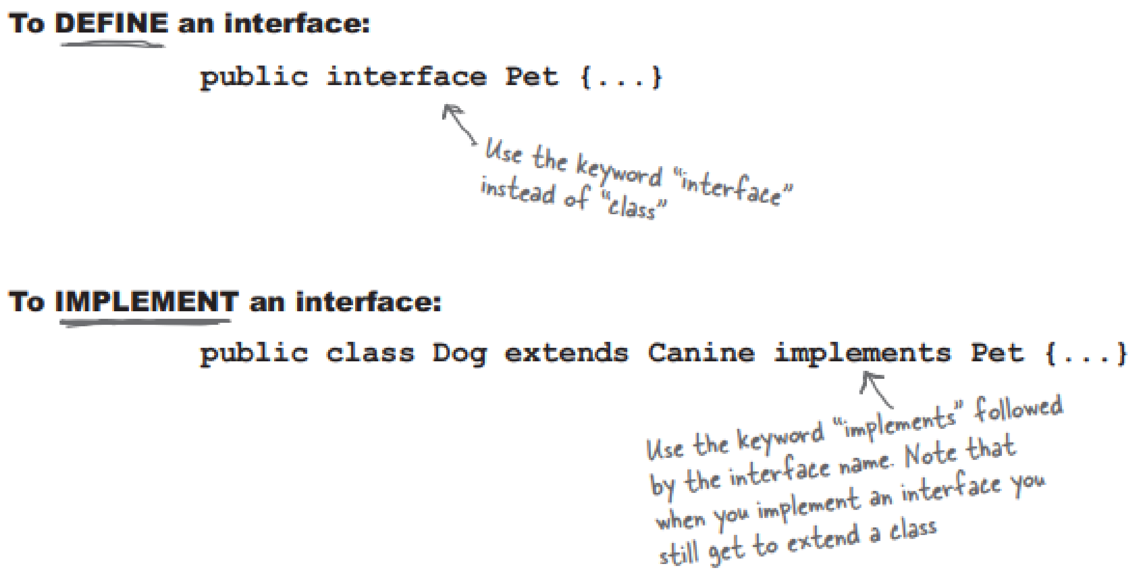
/* method signatures */

}



A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e. override) the methods of Pet.



To DEFINE an interface:

```
public interface Pet {...}
```

Use the keyword "interface" instead of "class"

To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet {...}
```

Use the keyword "implements" followed by the interface name. Note that when you implement an interface you still get to extend a class

An example of an interface:

```
public interface Edible{
public abstract String howToEat();
}
```

As with an abstract class, we cannot create an instance from an interface using the new operator.

Let’s use the Edible interface to specify whether an object is edible. This is achieved by letting the class/object implement this interface using the **implements** keyword.

```
class Animal{ }
```

```
class Tiger
extends Animal{
 }
```

```
class Chicken extends Animal
implements Edible{
   public String howToEat(){
      return "Chicken: Fry it";
}
```

The Chicken class extends Animal and implements Edible to specify that chickens are edible. When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type. The Chicken class implements howToEat method.

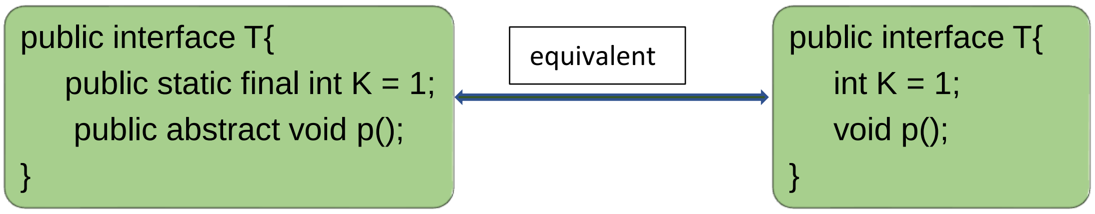
```
abstract class Fruit implements Edible{ }
```

```
class Apple extends Fruit{
      public String howToEat(){
            return "Apple: Make apple cider";
      }
}
```

```
class Orange extends Fruit{
      public String howToEat(){
            return "Orange: Make orange juice";
      }
}
```

The Fruit class implements Edible. Since it does not implement howToEat method, Fruit must be labelled abstract. The concrete subclasses of Fruit must implement howToEat method. The Apple and Orange classes implements the howToEat method.

Note: Since all data fields are public final static and all methods are public abstract in an interface, Java allows these modifiers to be omitted.



```
public interface T{
    public static final int K = 1;
    public abstract void p();
}
```

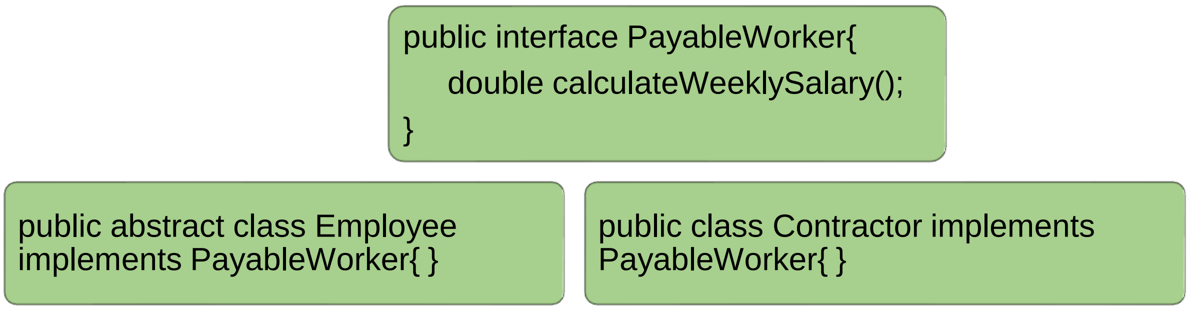
```
public interface T{
    int K = 1;
    void p();
}
```

Problem Statement:

The company has hired some contractors to do jobs for a fixed fee. Payroll wants to store the contractors with the employees who need to be paid but, a contractor is not an employee and does not have an employee number. To solve this problem, we have to think about what do contractors and employees have in common.

Solution

Payroll wants to calculate the amount to pay each person whether it is an employee or a contractor. As contractor does not have employee ID but both gets a pay and that is something common in them.
I have thought of defining an interface with calculateWeeklySalary method. Both Employee and Contractor classes will implement this interface.



```
public interface PayableWorker{
    double calculateWeeklySalary();
}
```

```
public abstract class Employee
implements PayableWorker{ }
```

```
public class Contractor implements
PayableWorker{ }
```

Once an object is stored in an array of PayableWorker

- the only methods you can call are those defined by the PayableWorker interface
- storing the objects in the array causes them to temporarily forget their previous type and behave as PayableWorkers.