# Interfaces

# Why Use Methods?

Write and test code once,
use it multiple  times: avoid duplication

Eg. Library.addBook()

# Why Use Methods?

Use it without understanding *how* it works:
**encapsulation** / **information hiding**

Eg. How does System.out.println() work?

# Why Use Objects?

Objects combine a related set of variables and methods

Provide a simple *interface*

(encapsulation again)

# Implementation / **Interface**

**Library**

```
Book[] books;
int numBooks;
String address;

void addBook(Book b) {
  books[numBooks] = b;
  numBooks++;
}
```

**Library**

```
void addBook(Book b);
```

# Java Interfaces

Manipulate objects, without knowing how they work

Useful when you have similar but not identical objects

Useful when you want to use code written by others

# Interface Example

```
public interface RemoteControl {

        public void turnOn();

        public void turnOff();

        public void printInfo();
}
```

# Implementing Interface

```
public class Television implements RemoteControl {
        public boolean onOff = false;

        public void turnOn() {
                onOff = true;
        }
        public void turnOff() {
                onOff = false;
        }
        public void printInfo() {
                System.out.println(onOff);
        }
}
```

# Testing Interface

```
public class RemoteControlTest {

        public static void main(String[] args) {
                RemoteControl rc = new Television();
                rc.turnOn();
                rc.printInfo();
        }

}
```

# Interfaces

Set of classes that share methods

Declare an *interface* with the common methods

Can use the interface, without knowing an object's specific type

# Interface Notes

Only have methods (mostly true)

Do not provide code, only the definition (called *signatures*)

A class can implement any number of interface

# Using Interfaces

Can only access stuff in the interface.

```
Drawable d = new BouncingBox(…);
d.setMovementVector(1, 1);
```

*The method setMovementVector(int, int) is undefined for the type Drawable*

# Down-casting

If you know that a variable holds a specific type, you can use a cast:

```
Drawable d = new BouncingBox(…);
BouncingBox box = (BouncingBox) d;
box.setMovementVector(1, 1);
```

# Interfaces? Interfaces!

- It's a contract!
- If you must implement **ALL** the methods
- All fields are **`final`** (cannot be changed)

```
public interface ICar {
  boolean isCar = true;

  int getNumWheels();
}
```

# BigRig

```
class BigRig implements ICar {
    int getNumWheels() {
          return 18;
    }
}
```

# Lab#6-1: Creating Interfaces

Let's create the Refrigerator class

Let’s Make it inherits from RemoteControl

Let's use it as like the Television class in RemoteControlTest.

# Abstract class

Controlling Inheritance – Enforcing

Inheritance can be forced by using the `abstract` keyword

A class with at least one abstract method must be declared abstract

```
public abstract class SomeOtherClass {
  public int aMethod() { … }
  public abstract void otherMethod();
}
```

# The abstract keyword applies to both methods and classes

```
public abstract class ElectricProduct {
        public boolean onOff = false;

        public abstract void turnOn();

        public abstract void turnOff();

        public void printInfo() {
            System.out.println(onOff);
        }
}
```

# An abstract method must be overrided by a sub-class

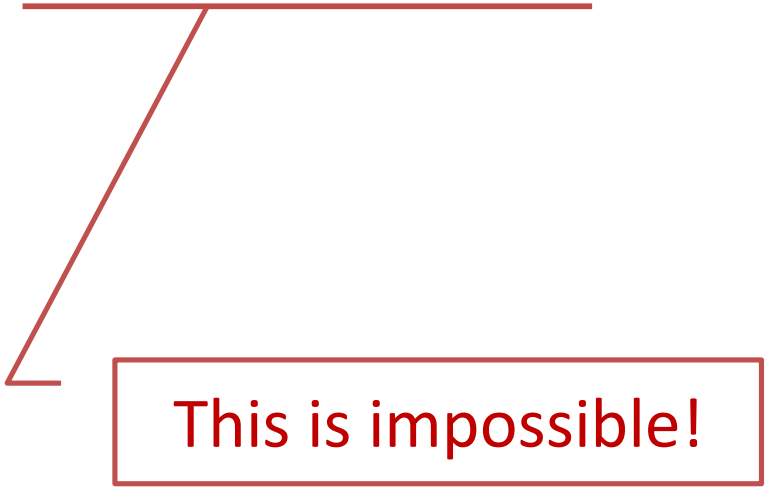
```
public class Television extends ElectricProduct {

        public void turnOn() {
                onOff = true;
        }
        public void turnOff() {
                onOff = false;
        }

}
```

# An abstract class must be extended, cannot be used to create objects

```
public class RemoteControlTest {
        public static void main(String[] args) {
                ElectricProduct ep = new ElectricProduct ();
                ep.turnOn();
                ep.printInfo();
        }
}
```



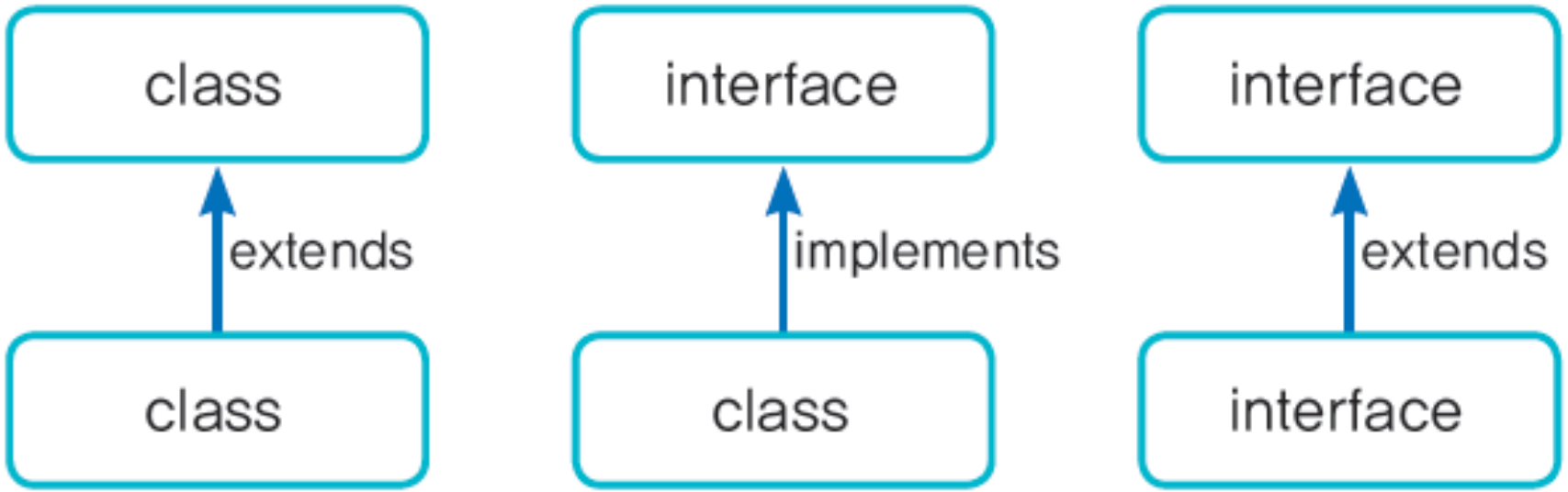
# An abstract class must be extended, cannot be used to create objects

```
public class RemoteControlTest {
        public static void main(String[] args) {
                ElectricProduct ep = new Television();
        ep.turnOn();
      ep.printInfo();
        }
}
```

# Possible Inheritance

A interface can inherit from B interface

A class can inherit from B and C interfaces



# Lab#6-2: Creating the following inheritance

1. Phone interface has the sendCall() method
2. MobilePhone interface inherits from PhoneInterface
3. MobilePhone interface has the sendSMS() method
4. Camera interface has the takePicture() method
5. PDA abstract class has playMusic() method
6. SmartPhone class inherits from PDA, MobilePhone, and Camara

(Each method is just a test, printing its own name)