COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

3. Abstract classes and interfaces

Aim

Introduces abstract classes and interfaces in Java

Objectives

At the end of this lecture, you will understand

- the role of abstract classes in object-oriented program design
- how abstract classes and interfaces support the use of polymorphism
- the difference between abstract classes and interfaces, and know when it is appropriate to use each

3. Abstract classes and interfaces

Outline

- Polymorphism
- Abstract classes
- Interfaces

Reading

Core Java (vol 1) Chapters 5 and 6

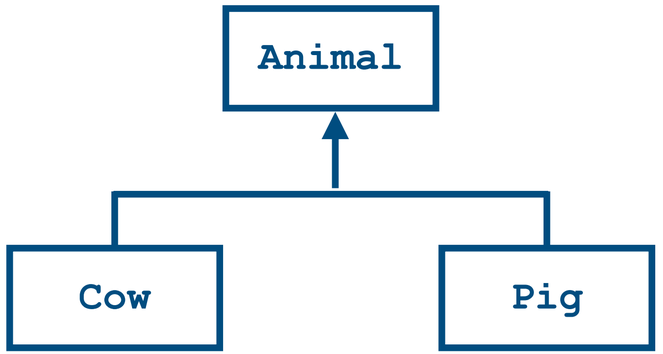
`http://download.oracle.com/javase/tutorial/java/IandI/abstract.html`
`http://download.oracle.com/javase/tutorial/java/IandI/createinterface.html`

Polymorphism

Polymorphism is a powerful feature of object-oriented systems.

* The basic idea of polymorphism is that a variable of type superclass X can also refer to an object of any subclass of X.

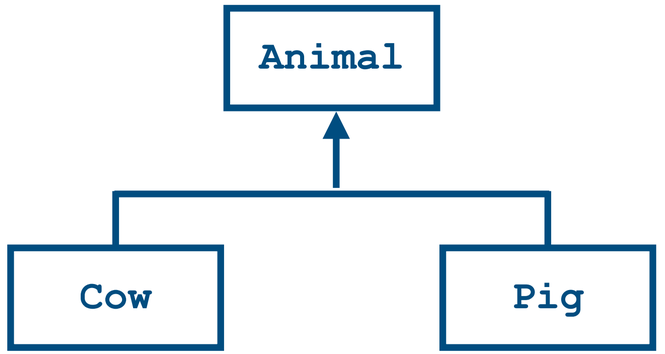
* Consider the following example. Variables of type `Animal` can refer to `Cow`
or `Pig` objects, but the converse is not true.



Polymorphism

* Consider the following example. Variables of type `Animal` can refer to `Cow` or `Pig` objects, but the converse is not true.

```
public class Animal {
   public void talk() {
     System.out.println("Animals can't talk");
   }
}
public class Cow extends Animal {
   public void talk() {
     System.out.println("Moo!");
   }
}
public class Pig extends Animal {
   public void talk() {
     System.out.println("Grunt!");
   }
}
```



Polymorphism

```
public class AnimalTest {
  public static void main(String[] args){
    Cow daisy = new Cow();
    Pig wilbur = new Pig();
    Animal animal = new Animal();
    animal.talk();




  }
}
```

We initially declare the variable `animal` to be of type `Animal`, and calling `animal.talk()` results in

Animals can't talk!

Polymorphism

```
public class AnimalTest {
  public static void main(String[] args){
    Cow daisy = new Cow();
    Pig wilbur = new Pig();
    Animal animal = new Animal();
    animal.talk();

    animal = daisy;
    animal.talk();



  }
}
```

We initially declare the variable `animal` to be of type `Animal`, and calling `animal.talk()` results in **Animals can't talk!**

We then reassign `animal` to refer to the `Cow` object daisy. Calling `animal.talk()` then results in a call to the `talk()` method associated with `Cows`, and so we get **Moo!**

Polymorphism

```
public class AnimalTest {
  public static void main(String[] args){
    Cow daisy = new Cow();
    Pig wilbur = new Pig();
    Animal animal = new Animal();
    animal.talk();

    animal = daisy;
    animal.talk();

    animal = wilbur;
    animal.talk();
  }
}
```

We initially declare the variable `animal` to be of type `Animal`, and calling `animal.talk()` results in

Animals can't talk!

We then reassign `animal` to refer to the `Cow` object daisy. Calling `animal.talk()` then results in a call to the `talk()` method associated with `Cows`, and so we get

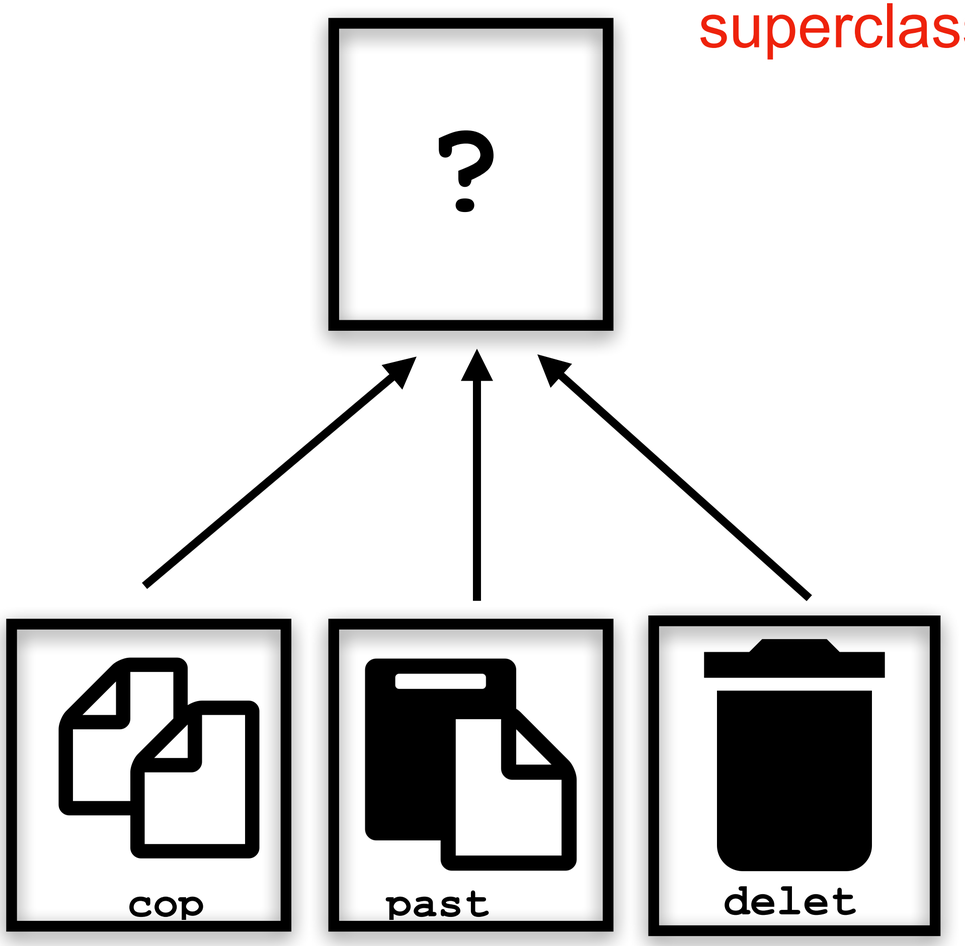
Moo!

Reassigning `animal` to refer to the `Pig` object `wilbur`, and calling the `animal.talk()` method, results in

Grunt!

Polymorphism in user interfaces

The basic idea of polymorphism is that a variable of type superclass X can also refer to an object of any subclass of X.




```
public class Button {
  private Action action;
  private String label;

  public Button(String label,Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

Polymorphism in user interfaces

```
public class Action {
  public void perform() {
    System.out.println(”nothing to do");
  }
}
```

```
public class Button {
  private Action action;
  private String label;
  public Button(String label,Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

Polymorphism in user interfaces

```
public class Action {
  public void perform() {
    System.out.println(”nothing to do");
  }
}
public class Delete extends Action {
  public void perform () {
    // delete the selected text
  }
}
```

```
public class Button {
  private Action action;
  private String label;
  public Button(String label,Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

Polymorphism in user interfaces

```
public class Action {
  public void perform() {
    System.out.println(”nothing to do");
  }
}
public class Delete extends Action {
  public void perform () {
    // delete the selected text
  }
}
public class Copy extends Action {
  private final Clipboard systemClipboard;
    public void perform () {
    // copy to systemClipboard
  }
}
```

```
public class Button {
  private Action action;
  private String label;
  public Button(String label,Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

Polymorphism in user interfaces

```
public class Action {
  public void perform() {
    System.out.println(”nothing to do");
  }
}
public class Delete extends Action {
  public void perform () {
    // delete the selected text
  }
}
public class Copy extends Action {
  private final Clipboard systemClipboard;
    public void perform () {
    // copy to systemClipboard
  }
}
public class Paste extends Action {
  private final Clipboard systemClipboard;
    public void perform() {
    // copy from systemClipboard
  }
}
```

```
public class Button {
  private Action action;
  private String label;
  public Button(String label,Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

Polymorphism in user interfaces

```
Button[] buttons = new Button[3];
buttons[0] = new Button("delete",
                        new Delete());
buttons[1] = new Button("copy",
                        new Copy());
buttons[2] = new Button("paste",
                        new Paste());
```

```
public class Button {
  private Action action;
  private String label;
  public Button(String label,
                Action a){
    this.label = label;
    this.action = action;
  }

  public void click() {
    action.perform();
  }
}
```

```
// later on inside a for loop …

buttons[i].click();
// this calls buttons[i].action.perform()
// and runs the perform() for the
// specific subclass of Action
```

We're initialising array of `Button` objects but with *different* subclasses of `Action` == one type of polymorphism

Polymorphism

We can assign objects of a subclass to a variable of the superclass, but *not vice versa*:

```
Animal animal = new Cow(); // OK - a Cow is-an animal
Cow daisy = new Animal (); // illegal - an animal is-not-a Cow!
```

Polymorphism

Two rules underlie polymorphism:

1. An object always retains the identity of the class from which it was created. Reassigning the `animal` variable to refer to `daisy` (an object of type `Cow`) does not affect the objects, only the object references.

```
Animal animal = new Animal();
Cow daisy = new Cow ();
animal = daisy; // <=== still refers to a Cow object;
```

reference changed

2. When a method is invoked on an object, the method associated with the class of the object is always used. So when `animal` refers to a `Cow` object, the methods of the `Cow` object are invoked.

```
Animal animal = new Cow();
animal.talk();  //<=== "moo"
```

Concrete and abstract classes

Concrete versus abstract

* To date, we have defined classes that can have direct instances

* These are called ***concrete*** classes:

```
Item item = new Item();
```

* An **abstract** class is a class that cannot have a direct instance.

* An abstract class is used to provide funcionality that will be inherited by concrete subclasses

Concrete and abstract classes

Example: a shape drawing application

* This needs to represent different kinds of geometrical shape

* All such shapes have *common attributes*, e.g., x and y coordinates

* The shapes also have some *different atributes*, e.g., a radius for a circle

* All such shapes have common methods, e.g., compute the area, draw the shape, which may be different for each kind of shape

Concrete and abstract classes

We might initially define the following for the `Shape` class, but how do we deal with the area and draw methods?

Different way to calculate; use abstract instead

```
import sheffield.*;
public class Shape {
  private double x, y;
  public Shape() { this(0.0,0.0); }
  public Shape(double xval, double yval)
                          { x= xval; y=yval; }
  public void setPosition(double xval, double yval)
                          { x = xval; y = yval; }
  public double getX() { return x; }
  public double getY() { return y; }

 // How to deal with these two methods?
 // public double area();
 // public void draw(EasyGraphics g);
```

Note: EasyGraphics is part of the Sheffield package

Concrete and abstract classes

We could define the methods **`area()`** and **`draw()`** in each subclass:

i

```
import sheffield.*;
public class Rectangle extends Shape {
  private double width, height;
  public Rectangle(double x, double y, double w, double h)
    { super(x, y); width = w; height = h; }
  public double area() { return width * height; }
  public void draw(EasyGraphics g) {
    g.moveTo(getX(),getY());
    g.lineTo(getX()+width, getY());
    g.lineTo(getX()+width, getY()+height);
    g.lineTo(getX(),getY()+height);
    g.lineTo(getX(), getY());
  }
}
```

Concrete and abstract classes

But what would happen if the user created an instance of the `Shape` class and called the `area` method?

- One solution would be to put empty methods for `area` and `draw` in the `Shape` class, and allow the subclasses to override them.
- But this is unsatisfactory, as either method invoked on an instance of the `Shape` class would produce worthless results.
- A better solution would be to not allow the user to create instances of the `Shape` class at all.
- This is the basic idea behind abstract classes.

Abstract classes

If the `Shape` class is declared to be abstract, then instances of the `Shape` class cannot be created:

```
public abstract class Shape {
  protected double x;
  protected double y;
  public void setPosition(double xval, double yval) {
    x=xval;
    y=yval;
  }
  public abstract double area();
  public abstract void draw(EasyGraphics g);
}
```

Circle class

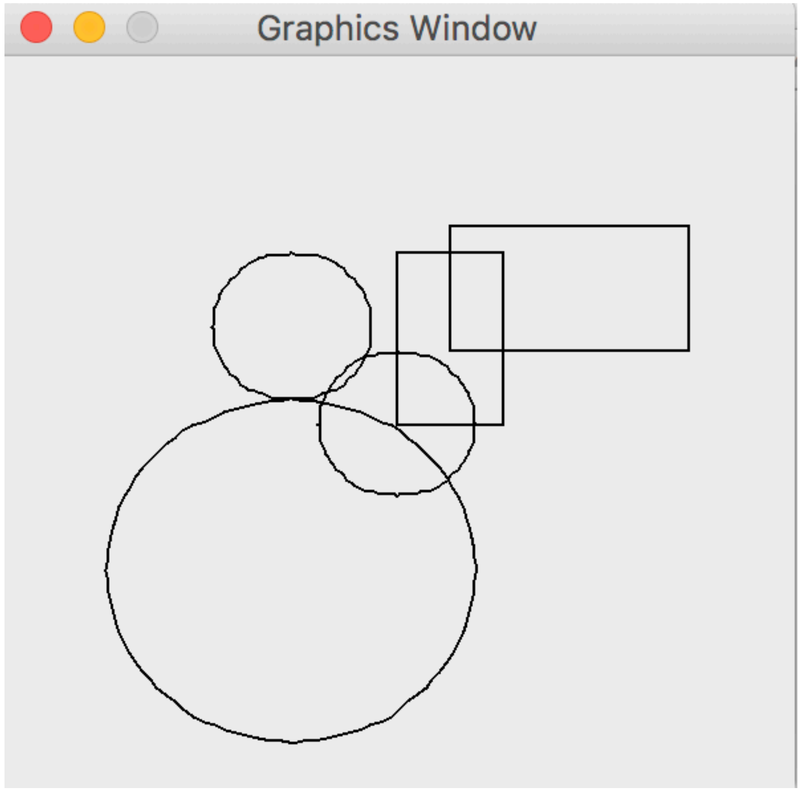
```
import sheffield.*;
public class Circle extends Shape {
  private static final int NUM_STEPS = 100;
  private double radius;
  public Circle(double x, double y, double r)
                          { setPosition(x, y); radius=r; }
  public double area() { return Math.PI*radius*radius; }
  public void draw(EasyGraphics g) {
    g.moveTo(getX(),getY()+radius);
    for (int i=0; i<=NUM_STEPS; i++) {
      double w = i*2*Math.PI/NUM_STEPS;
      g.lineTo(getX()+radius*Math.sin(w),
      getY()+radius*Math.cos(w));
    }
  }
}
```

Abstract classes and polymorphism

Why bother with an abstract Shape class?

* By abstracting common features (of shapes in this example) we can now exploit polymorphism by storing a collection of shapes in an array.
* Consider what this code would look like without using an abstract class.

```
import sheffield.*;
public class ShapeDemo {
  Public attic void main(String[] args) {
    EasyGraphics g = new EasyGraphics(300,300,15050);
    Shape[] list = new Shape[5];
    // fill the array with shapes
    list[0] = new Rectangle(20,20,40,70);
    list[1] = new Circle (-40,-60,70);
    list[2] = new Triangle (-40,40,30);
    list[3] = new Rectangle (20,50,90,50);
    list[4] = new Circle(0,0,30);
    // now update the display
    for (int i=0; i<5; i++) {
      list[i].draw(g);
    }
}
```



```
import sheffield.*;
public class ShapeDemo {
    public static void main(String[] args) {
    EasyGraphics g = new EasyGraphics(300,300,150,150);
    Shape[] list = new Shape[5];
    list[0] = new Rectangle(20,20,40,70);
    list[1] = new Circle(-40,-60,70);
    list[2] = new Triangle(-40,40,30);
    for (int i=0; i<list.length; i++)
       list[i].draw(g);
    }
}
```

```
import sheffield.*;
public class ShapeDemo {
    public static void main(String[] args) {
    EasyGraphics g = new EasyGraphics(300,300,150,150);
    Shape[] list = new Shape[]{
       new Rectangle(20,20,40,70),
       new Circle(-40,-60,70),
       new Triangle(-40,40,30)
    };
    for (Shape s:list)
       s.draw(g);
}
```

Different ways of initialising an array

Different ways of going through an array

Interfaces

The idea of abstract superclasses does have limitations.

* What do we do if we want our Shape classes to inherit methods or instance fields from *more than one* superclass?

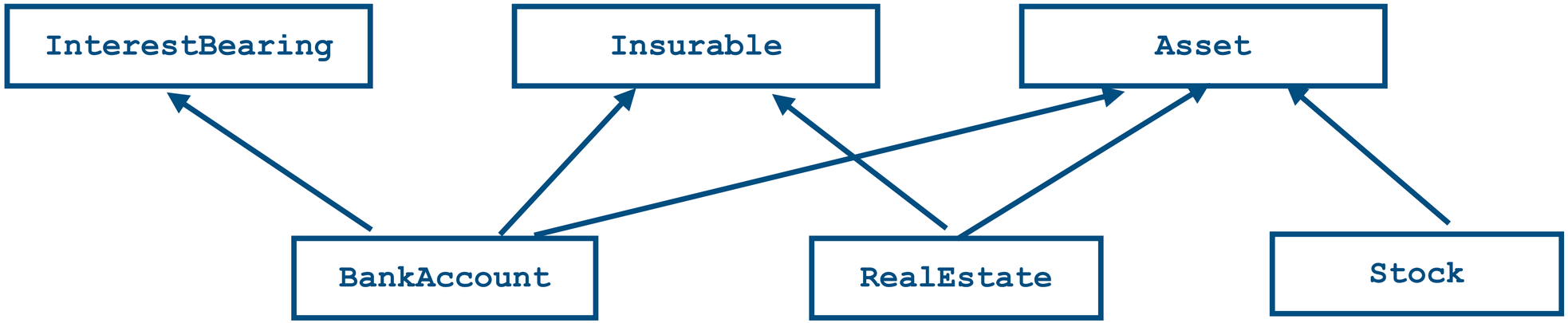
* Java does not permit multiple inheritance (unlike C++ and Eiffel).

* But we can specify an interface in Java, and a class can implement more than one interface.

Interfaces

As an example, consider a model of financial assets:

- `Stock`, `RealEstate`, and `BankAccount` are types of `Asset` (**is-an**).
- `BankAccounts` are *interest bearing.*
- `BankAccounts` and `RealEstate` can be *insured against loss.*



Inheritance hierarchy for assets. Note that the hierarchy exhibits both single inheritance (**`Stock`** inherits from **`Asset`**) and multiple inheritance (e.g., **`RealEstate`** inherits from **`Asset`** and **`Insurable`**)

Interfaces

If we define `Asset`, `Insurable`, and `InterestBearing` as superclasses, then this example requires multiple inheritance

But a class definition with multiple inheritance in Java is **illegal**, so we can't use statements like

```
public class RealEstate extends Asset,
Insurable { ... }
        // Not allowed
```

- Instead, we can use an ***interface***

Interfaces

In Java, an *interface* describes what a class does, but not how it does it.

An interface is essentially a set of requirements for a class

A class can `implement` more than one interface

Interfaces

The `Asset` superclass uses the structure we saw before for the **`abstract`** `Shape` class:

```
public abstract class Asset {
  protected double value;
  public double getValue(){
     return value;
   }
}
```

The `Insurable` and `InterestBearing` **`interfaces`** have the following structure:

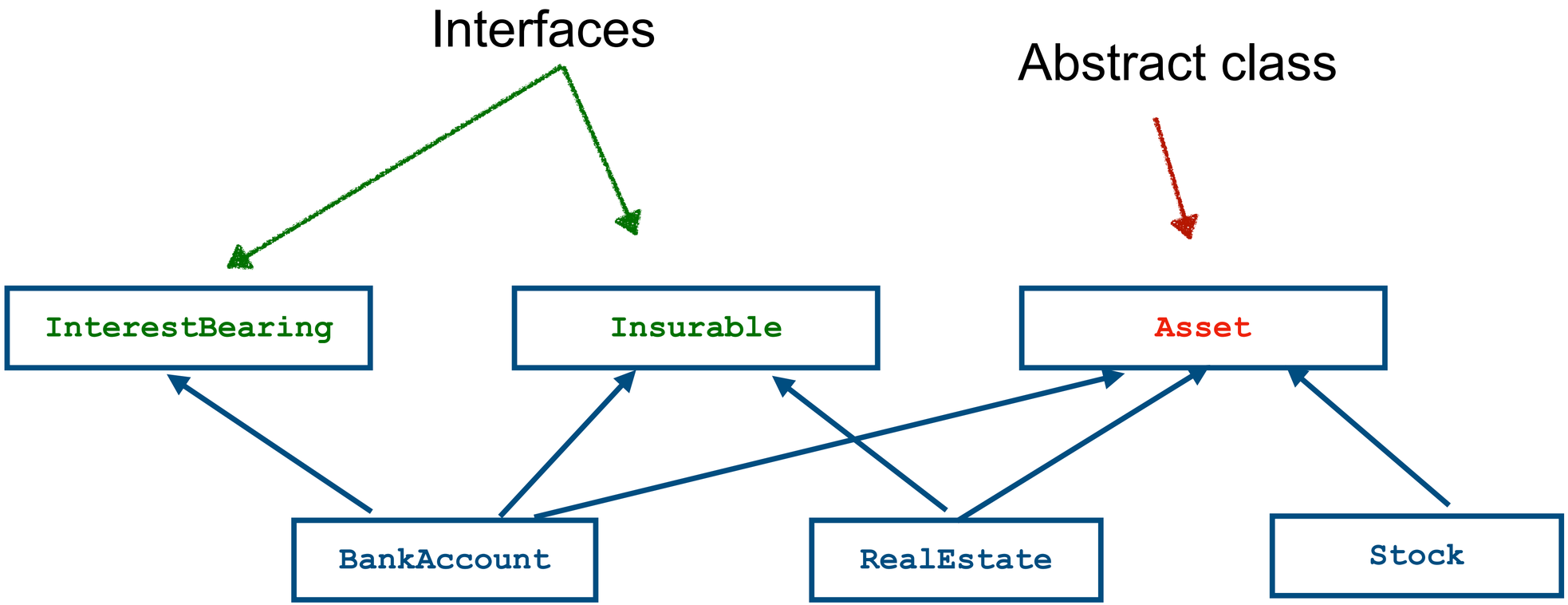
```
public interface Insurable {
   public abstract double getPremimum();
}

public interface InterestBearing {
   public abstract double getInterestRate();
}
```

All methods in an interface are automatically **`public`** and **`abstract`**.

These are not method definitions, so no braces {}but a semicolon.

Interfaces


Interfaces
Abstract class
InterestBearing
Insurable
Asset
BankAccount
RealEstate
Stock

Interfaces

We can now define a `RealEstate` class that **`extends`** the `Asset` class and **`implements`** the `Insurable` interface:

```
public class RealEstate extends Asset implements Insurable {
  public RealEstate (double v) { value = v; }
  public double getPremium () {
    // insurance premium is 5% of the value
    return value * 0,05;
}
```

`RealEstate` does not include a **`getValue()`** method because this method is inherited from **`Asset`**.

But **`RealEstate`** does include **`getPremimum()`**, because implementing the **`Insurable`** interface is a commitment to including a **`getPremium()`** method.

Interfaces

Classes can implement more than one interface:

```
public class BankAccount extends Asset implements InterestBearing, Insurable {
    public BankAccount (double v) {
      value = v;
    }
    public double getInterestRate () {
      return 5.2;
    }
    public double getPremimum () {
      // insurance premium is 1% of value
      return value * 0.01;
    }

}
```

This class inherits from one superclass, but implements two interfaces

Interfaces

Interfaces are not classes and cannot be instantiated:

```
Insurable x = new Insurable ();
```

Nope!!

You can declare a variable within an interface type if it refers to an object that implements the interface

```
Insurable x;
```

Ok :)

```
x = new RealEstate (…);
```

Ok, if RealEstate implements Insurable

Interfaces can be used in a similar way as abstract classes to generate collections of different objects with common features, which allows us to make use of polymorphism.

However, some care is needed:

```
Insurable myAsset = new BankAccount (354.53);
System.out.println (myAsset.getPremium());
System.out.println (myAsset.getInterestRate());
```

Illegal because `Insurable` does not include `getInterestRate ()`

Interfaces

- Never have instance fields.
- Can have zero or more methods, but they never implement methods.
- Can define constants:
  - Constants defined in interfaces are always **`public static final`**, but you do not need to declare them as such.
  - The following example will print 70.0 as output.

```
public interface MyInterface{
        double MY_CONSTANT = 70.0;
}

public class InterfaceTest implements MyInterface{
    public static void main(String[] args) {
        System.out.println( MY_CONSTANT );
    }
}
```

Interfaces

Can use `instanceof` to check if an object implements an interface, for example:

```
Object x = new BankAccount (3.50);

if (x instanceof BankAccount) {
    // true - obviously!
}

if (x instanceof Insurable) {
    // this is also true
}

if (x instanceof Animal) {
    // this is false
}
```

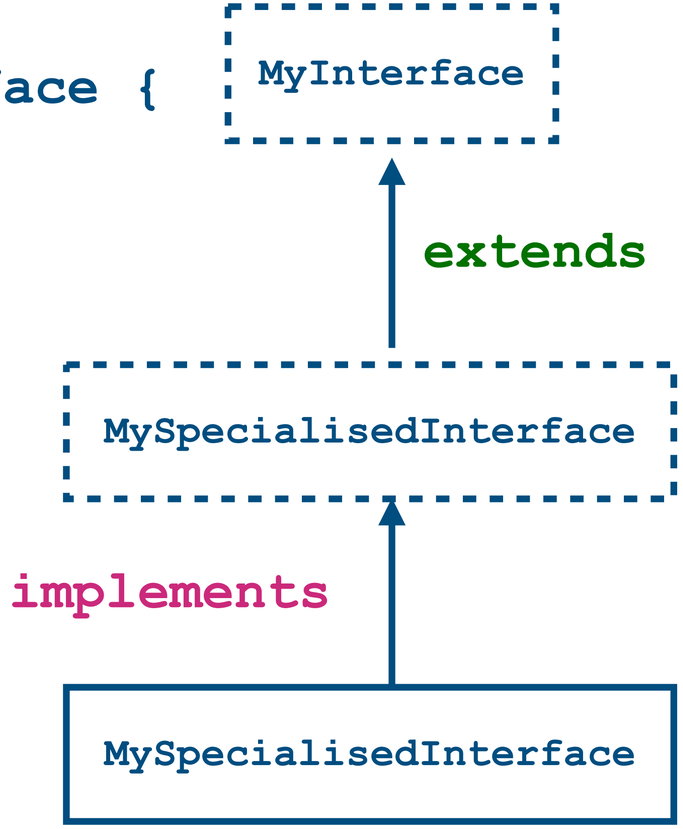
Interfaces

Can be extended into hierarchies:

```
public interface MyInterface {
  public static final double MY_CONSTANT = 70.0;
}

public interface MySpecialisedInterface extends MyInterface {
  public double anAdditionalMethod ();
}

public class MyImpl implements MySpecialisedInterface {
  public double anAdditionalMethod (){
     // implementation goes here
     // can use MY_CONSTANT
  }
}
```



Summary

- Abstract classes and interfaces provide a way of modelling data and functionality in OO systems.
- Important for developing large pieces of software in groups because they force classes to behave in well specified ways.
- This allows code to be elegant, efficient, and maintainable.
- The idea of separating interface from implementation is a powerful one: It enables classes to be constructed that can operate on objects of any type, so long as they implement the appropriate interface.

When you use an interface, and when you use an abstract class?

http://download.oracle.com/javase/tutorial/java/IandI/abstract.html