

Quentin Tarantino's "Learn Java in a Minute"

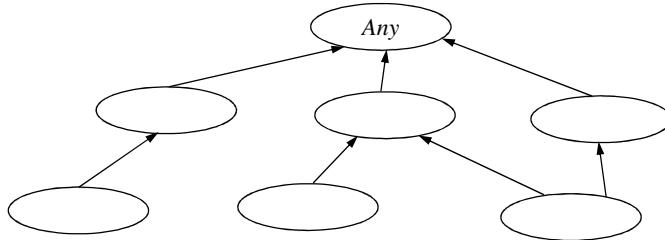
OBJECT-ORIENTED PROGRAMMING

# Abstract Classes and Interfaces

Lecture #4

## Class hierarchy

- By class inheritance we can model class hierarchy



- Is it good to have one class hierarchy in the system?
  - All classes derive from *Any*, i.e. “**classes can be of any type**”
  - What is in the top of the hierarchy?

## Advantages of class hierarchy

- We can use **polymorphic references to any object** in the system
  - Reusable functions work with any objects of the system
  - We do not need to rewrite functions
    - For example method `Shop.sell()` works with both `Article` and `DiscountArticle`
- In the top of the hierarchy we can specify **general (universal) properties** of all objects in the system, e.g.
  - Clone – duplicating objects
  - Copy – copying content of the object to another
  - Equal – field-by-field comparison of the objects

## Subtype example

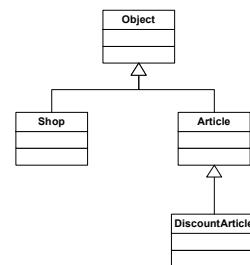
- Class `java.util.Vector` has reusable method  
`void addElement (Object obj)`

```
public class StringSet {
    private Vector elements;
    public void insert(String s) {
        this.elements.addElement(s);
    }
}
```

Why can we use a `String`  
where an `Object` is  
expected?

## Building class hierarchy

- Java (and other object-oriented languages too) implements general inheritance structure
- **Any class that does not include an inheritance clause, implicitly inherits from class `Object`**
- Class `Object` is a kernel class of Java and specifies some universal features of all objects
- In our project `Shop` and `Article` classes implicitly inherit from `Object` class



# Polymorphic reference

- A **polymorphic reference** can refer to different types of objects at different times
  - In Java every reference can be polymorphic except of references to base types and final classes
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
  - Polymorphic references are therefore resolved at run-time, not during compilation – **dynamic binding**
- Careful use of polymorphic references can lead to elegant, robust software designs

# Subtype example

- Class `java.util.Vector` has reusable method

```
void addElement(Object obj)
```

```
public class IntSet {
    private Vector elements;
    public void insert(int x) {
        this.elements.addElement(x);
        this.elements.addElement(new Integer(x));
    }
}
```

Primitive types are not subtypes of Object.

But Integer is...

# Abstract class

- Some classes describe an **abstract** idea rather than a **specific** one

```
public abstract class Food {
    ...
}

public class Pizza extends Food {
    ...
}

public class Salad extends Food {
    ...
}
```

- Abstract class is **declared as abstract** and cannot be **instantiated**

- We can create instances of Pizza and Salad but not of Food

# Abstract method

- Abstract class is just to **guarantee** that its closed subclasses **must** override its abstract methods
- Abstract class declares **abstract methods**
  - They do not have body, they just declare an abstract feature, which must be overridden in subclasses

```
public abstract class Food {
    public abstract void prepare();
}

public class Pizza extends Food {
    public void prepare() {
        ...
    }
}

public class Salad extends Food {
    public void prepare() {
        ...
    }
}
```

The abstract method

Have the dough, cut tomatoes, add cheese, insert the oven...

Cut vegetables, add all ingredients on the plate...

## Abstract method

- Let's have a kitchen where we can prepare a food – then test our example

```
public class Kitchen {
    public void prepareFood(Food f) {
        f.prepare();
    }
}

public class TestKitchen {
    public static void main(String[] args) {
        Kitchen kitchen = new Kitchen();
        kitchen.prepareFood(new Pizza());
        kitchen.prepareFood(new Salad());
        kitchen.prepareFood(new Food());
    }
}
```

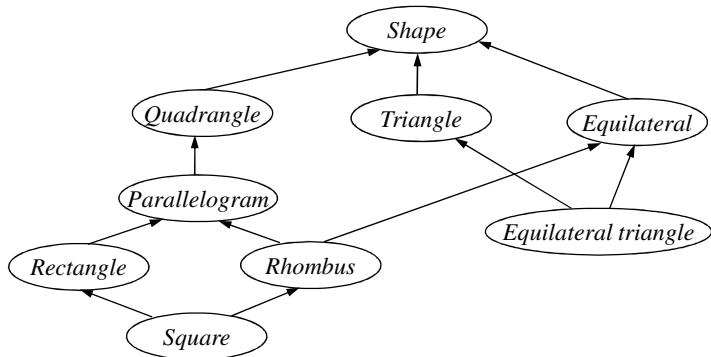
Parameter f is **polymorphic reference**

Abstract class **cannot be instantiated** (because abstract implementation is missing)

## Constructors in abstract class

- Do we need a constructor for abstract class?
  - Remember the inheritance: At the beginning of each constructor of subclass the default constructor of superclass is called
- Do not define public** constructors in abstract classes
  - Constructors with public are for types that can be instantiated. Abstract types can never be instantiated.
- Do define a protected** constructor in abstract classes
  - The base class can perform initialization tasks when instances of a derived class are created

# A class hierarchy abstraction



- What are the supertypes of *Square*?
- What are the subtypes of *Parallelogram*?

# Reusing implementation

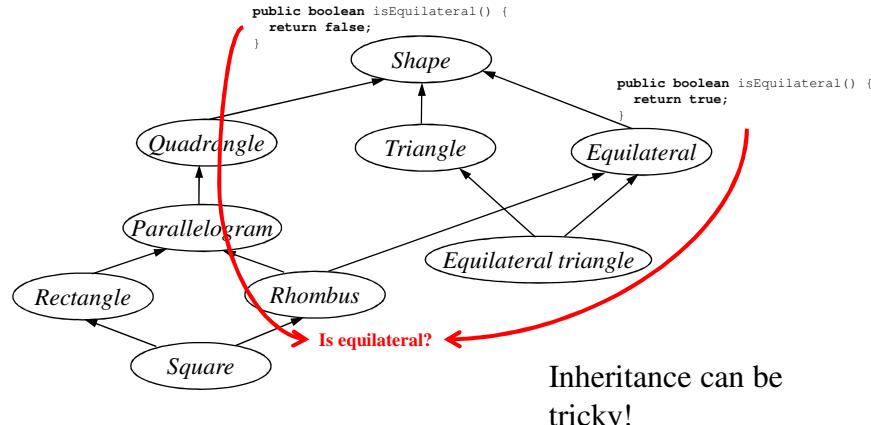
- All shapes reuse (inherit from *Shape*) an *isEquilateral()* method

```

public class Shape {
    ...
    public boolean isEquilateral() { return false; }
    ...
}

public class Equilateral extends Shape {
    ...
    public boolean isEquilateral() { return true; }
    ...
}
    
```

# Is a Rhombus equilateral?



# Solutions of multiple inheritance problems

- Java, C#
  - Allow multiple supertypes using interfaces, but only one implementation
  - Pro: Safe and simple
  - Con: Limits reuse
- C++
  - Allows it, let programmers shoot themselves if they want

# Interface

- An interface is the set of methods one object must implement
- In many ways, **interface** is very similar to **abstract class**

```
public interface InterfaceName {
}
```

- Unlike abstract class which can also contain non-abstract methods, interface contains **only** abstract methods and constants

# Interface example

- Let's have interface that provides the specification for maneuverable vehicles

```
public interface Maneuverable {
    public static final double MAX_SPEED = 1000.0;
    public static final double MIN_SPEED = 0.0;
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

# Interface example

- Let's apply our interface to some classes of vehicles

```
public class Car implements Maneuverable {
    ...
}

public class Boat implements Maneuverable {
    ...
}

public class Submarine implements Maneuverable {
    ...
}
```

Code here which implements all  
Maneuverable methods for Car

Code here which implements all  
Maneuverable methods for Boat

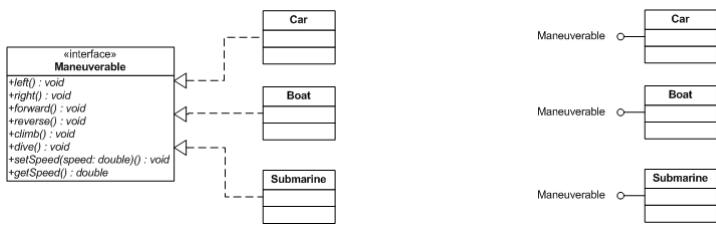
Code here which implements all  
Maneuverable methods for Submarine

# Interface example

- A method `travel()` in some other class can maneuver the vehicle without being concerned about what the actual class (Car, Boat, Submarine) is or what inheritance hierarchy it is in

```
public void travel(Maneuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

# Interfaces in UML



# Interface

- An object can have **many** interfaces
  - Essentially, an interface is a subset of all the methods that an object implements
  - We can inherit only **one class**, but we can implement **many interfaces**

```
public class A extends B implements I1, I2, I3 {
    ...
}
```

- Interface can inherit from another interface

```
public interface A {
    public void f();
}
public interface B extends A {
    public void g();
}
public class C implements B {
    public void f() { ... }
    public void g() { ... }
}
```

Class C must implement method `g()`  
from interface B and also method `f()`  
from inherited interface A

# Interfaces as types

- **A type is a specific interface of an object**
- Different objects can have the same type and the same object can have many different types
- An object is known by other objects only through its interface
- Interface is an implementation of subtyping in object oriented language
  - Describes when one object can be used in place of another object

# Abstract class vs interface

- Why not use abstract class instead of interface?
  - In C++, a class can inherit multiple superclasses which is known **multiple inheritance**
  - Java does not allow multiple inheritance and a class can only have a single inheritance
- In **interface**, you **cannot** include **non-abstract methods** at all
  - Classes that implement the interface **must override every method**
- In **abstract class**, you can **mix non-abstract and abstract methods** together
  - Subclasses could reuse some non-abstract methods without override

## Interface like abstract class

- If a class implements an interface, you have to override the interface's methods in the class
- You cannot create instances from an interface by using **new** operator
- Interface can be a type as well as class
- The purpose of creating interface is because of **polymorphism**

## Interface unlike abstract class

- You can have multiple interfaces in one class
- Interface is **not** designed to be **superclass**, but interface is designed to **add some behaviors** to a class
- A relationship between **(abstract) class and class** is a **strong relationship** and it is known as **IS-A relationship**
  - “A duck is a bird” – It clearly means the duck is really a bird, so the bird can be a superclass of a duck and it could be either concrete or abstract class
- A relationship between **class and interface** is a **weak relationship** and it is known as **IS-KIND-OF relationship**.
  - “A duck is flyable” – Flyable can never ever be the superclass of the duck, it just means this duck can fly, so flyable is interface

# Conventions for interfaces

- Because the interface is just designed to add some behaviors or some features to classes, usually it contains only one or two general methods

```
public interface Runnable {
    public void run();
}
```

- The reason for this is that interface is not a superclass, so it does not specify who can use its methods. Generally, **its method might be used by everyone**
- By Java code convention, the name of interface is usually **adjective**, because adjective adds some meaning to a noun
  - Runnable, Comparable, Cloneable, Accessible
- The interface names for event driven listener are usually ended with Listener
  - ActionListener, MouseMotionListener, KeyListener
- Some programmers use the “I” prefix for interface names (Hungarian notation)
  - ICommand, IMessage

# Readings

- ECKEL, B.: *Thinking in Java*. 4th Edition, Prentice Hall, 2006
  - Interfaces (pp. 219 – 242)