

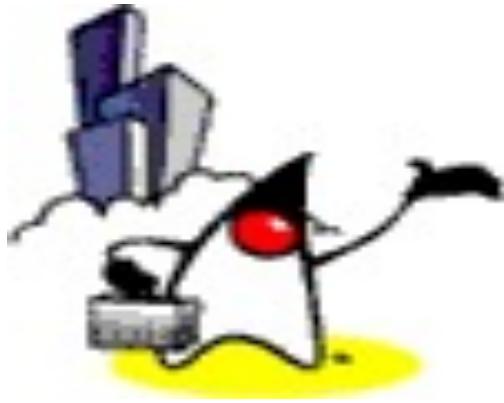
# Abstract Class & Java Interface

**“Code with Passion!”**



# Topics

- What is an Abstract method and an Abstract class?
- What is Interface?
- Why Interface?
- Interface as a Type
- Interface vs. Class
- Defining an Interface
- Implementing an Interface
- Implementing multiple Interface's
- Inheritance among Interface's
- Rewriting an Interface



**What is an Abstract Method?  
and  
What is an Abstract Class?**

# What is an Abstract Method?

- Method that does not have implementation (code block)
- To create an abstract method, just write the method declaration without the body (code block) and use the **abstract** keyword
  - Since there is no body of code, there is no { }

```
// Note that there is no body of code
public abstract void someAbstractMethod();
```

# What is an Abstract Class?

- An abstract class is a class that contains one or more **abstract methods**
  - Compile error occurs if no abstract method is present
- An abstract class cannot be instantiated - you cannot create an object instance from it
  - // You will get a compile error on the following code  
*MyAbstractClass a1 = new MyAbstractClass();*
- Another class (Concrete class) must be created to provide implementation of all abstract methods
  - Concrete class uses **extends** keyword to extend the abstract class
  - Concrete class must implement **all abstract methods** of the abstract class in order to be used for instantiation - compile error if not all abstract methods are implemented

# Example: Let's say we have an Abstract Class

```
public abstract class LivingThing {  
  
    // Regular method with code block - non abstract method  
    public void breath(){  
        System.out.println("Living Thing breathing...");  
    }  
  
    public void eat(){  
        System.out.println("Living Thing eating...");  
    }  
  
    /**  
     * Abstract method walk()  
     * We want this method to be implemented by  
     * Concrete classes.  
     */  
    public abstract void walk();  
}
```

# Extending an Abstract Class

- When a concrete class extends the *LivingThing* abstract class (of previous slide), it must implement all abstract methods

```
// Human class below is a concrete class
// because it implements all abstract methods
// (there is only one in this example) of LivingThing
// abstract class
public class Human extends LivingThing {

    public void walk() {
        // implementation of walk() method
        System.out.println("Human walks...");
    }
}
```

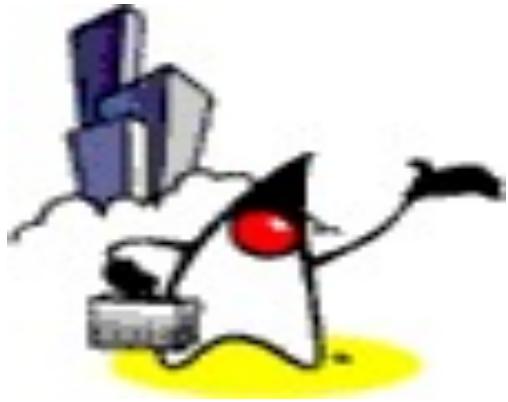
# When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses (Concrete classes) are expected to fulfill a similar role in different ways through different implementations (this is Polymorphism)
  - These subclasses extend the same Abstract class and provide different implementations for the abstract methods of the Abstract class
- Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy and use its subclasses to provide implementation details of the abstract methods of the abstract class

# Lab:

**Exercise 1: Abstract Class  
1027\_javase\_interface.zip**





# What is Interface?

# What is an Interface?

- All methods of an interface are abstract methods (while in an abstract class, only some methods are abstract methods)
  - Defines the signatures of a set of methods, without the body (implementation of the methods)
  - No need to use *abstract* modifier for the methods since all methods in an interface are considered as abstract methods
- A concrete class must implement the interface
  - All abstract methods must be implemented (otherwise, it is a compile error)
- Use *interface* modifier to the declaration of an interface

# What is an Interface for?

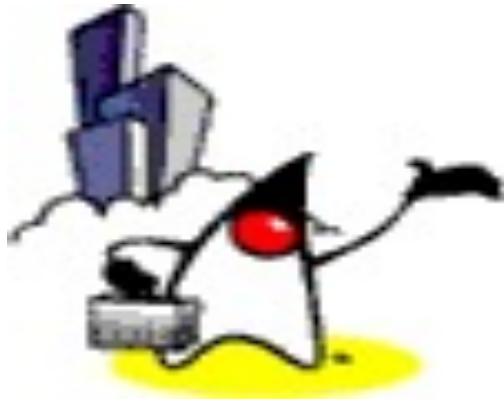
- It defines a public way of specifying the behavior of classes
  - Defines a contract between “client” (user of the interface) and “provider” (implementation) of some task

# Example: Interface Relation

```
// Note that Interface contains just a set of method  
// signatures without any implementations.  
// No need to say abstract modifier for each method  
// since it assumed.  
  
public interface Relation {  
  
    public boolean isGreater( Object a, Object b);  
    public boolean isLess( Object a, Object b);  
    public boolean isEqual( Object a, Object b);  
  
}
```

## Example 2: Interface OperateCar

```
public interface OperateCar {  
    // Constant declarations, if any  
  
    // Method signatures  
    int turn(Direction direction,  
             double radius, double startSpeed, double endSpeed);  
    int changeLanes(Direction direction, double startSpeed,  
                     double endSpeed);  
    int signalTurn(Direction direction, boolean signalOn);  
    int getRadarFront(double distanceToCar,  
                      double speedOfCar);  
    int getRadarRear(double distanceToCar,  
                     double speedOfCar);  
    ....  
    // More method signatures  
}
```



# Why Interface?

(These are critical concepts  
you must understand!!!)

# Why do we use Interfaces? Reason #1

- In order to reveal programming interface (functionality of the object) of a class without revealing its implementation detail
  - This is the concept of encapsulation
- The implementation of the class can change without affecting the caller (user) of the interface
- The caller does not need the implementation class at compile time - only interface is needed
  - Different groups can work in parallel with only interfaces defined
  - During run-time, object instance gets created (from implementation class) and is associated with the interface type

# Why do we use Interfaces? Reason #2

- To have unrelated classes implement similar methods (behaviors)
- Example: Let's say there are Class `Line` and class `MyInteger`
  - They are not related through inheritance
  - But you want both to implement comparison methods
    - `checkIsGreater(Object x, Object y)`
    - `checkIsLess(Object x, Object y)`
    - `checkIsEqual(Object x, Object y)`
- Define `Comparison` interface which has the three abstract methods above and have `Line` and `MyInteger` classes to implement the `Comparison` interface

# Why do we use Interfaces? Reason #3

- To model multiple inheritance - you want to impose multiple sets of behaviors to your class
  - Each set is defined as an interface
  - A class can implement multiple interfaces while it can extend only one class

# Interface vs. Abstract Class

- All methods of an Interface are abstract methods while some methods of an Abstract class are abstract methods
  - Abstract methods of Abstract class must have **abstract** modifier
- An interface can only define constants while abstract class can have fields



# Interface as a Type (This is a critical concept!!!)

# Interface as a Type (Very important) #1

- When you define a new interface, you are defining a new type - **you can use an interface anywhere you can use a class as a type**
- Use of Interface rather than Class as a type is strongly recommended
- Let's say Person class implements *PersonInterface* interface, now you can do

```
Person p1 = new Person(); // Class is used as reference type
```

```
PersonInterface pi1 = p1; // Interface is used as reference type
```

```
PersonInterface pi2 = new Person(); // Interface is used as reference type
```

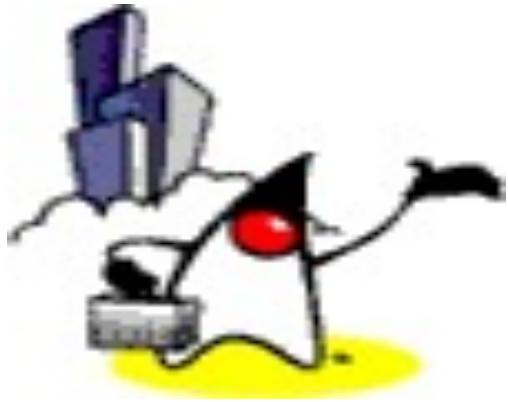
Recommended practice

# Interface as a Type (Very important) #2

```
// Class is used as an argument type  
aMethod(Person p){  
    ...  
}
```

```
// Interface is used as argument type  
aMethod(PersonInterface p){  
    ...  
}
```

recommended practice



# Interface vs. Class

# Interface vs. Class: Commonality

- Interface and Class can both define methods
- Interfaces and classes are both types - this means that an interface can be used in places where a class can be used

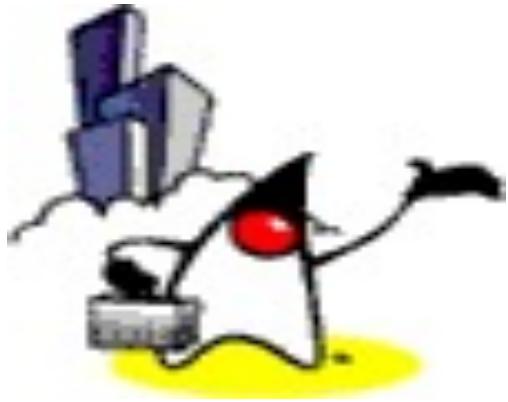
```
// Recommended practice
PersonInterface pi = createPerson();

// Not recommended practice because
// - Person class implementation must be
//   present even during compile time
// - Tightly coupled to Person class
//   implementation
Person pc = createPerson();
```

# Interface vs. Class: Differences

- The methods of an Interface are all abstract methods
  - They cannot have bodies
- You cannot create an instance from an interface - you must create a concrete class that implements the interface for object instantiation

```
// Compile error
PersonInterface pi = new PersonInterface();
```



# Defining Interface

# Defining an Interface

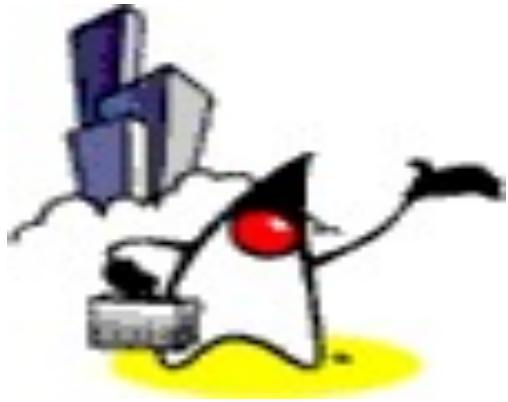
- To define an interface, use *interface* keyword

```
public interface [InterfaceName] {  
    //some methods without the body  
}
```

# Defining an Interface

- As an example, let's create an interface that defines relationships between two objects according to "some order" of the objects.

```
public interface Relation {  
    public boolean isGreater( Object a, Object b );  
    public boolean isLess( Object a, Object b );  
    public boolean isEqual( Object a, Object b );  
}
```



# Implementing Interface

# Implementing an Interface

- To create a concrete class that implements an interface, use the **implements** keyword.

```
/**  
 * Line class implements Relation interface  
 */  
public class Line implements Relation {  
  
    private double x1;  
    private double x2;  
    private double y1;  
    private double y2;  
  
    public Line(double x1, double x2, double y1, double y2) {  
        this.x1 = x1;  
        this.x2 = x2;  
        this.y1 = y1;  
        this.y2 = y2;  
    }  
  
    // More code follows
```

# Implementing an Interface

```
public double getLength() {
    double length = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
    return length;
}

// Line class has to implement the following abstract methods
public boolean isGreater( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen > bLen);
}

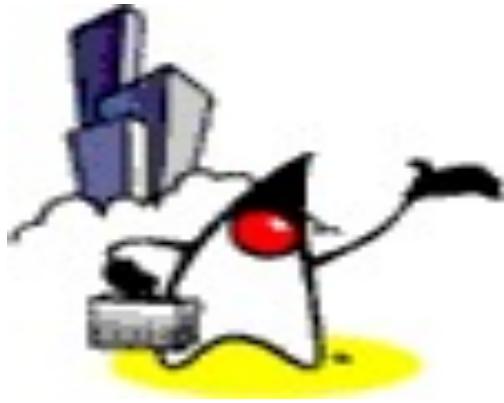
public boolean isLess( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen < bLen);
}

public boolean isEqual( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen == bLen);
}
```

# Implementing an Interface

- When your class tries to implement an interface, always make sure that you implement all methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override  
abstract method  
    isGreater(java.lang.Object,java.lang.Object) in  
    Relation  
  
public class Line implements Relation  
    ^  
  
1 error
```



**So, Which One to Use?**  
**(Person class**  
**vs.**  
**PersonInterface & PersonImpl)**

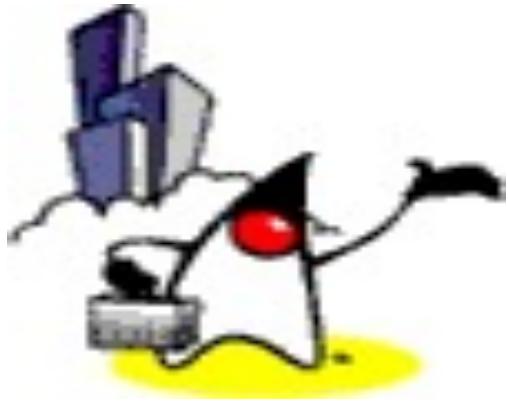
# Which one to use Class vs Interface/Impl

- Let's assume you are provided a class that will be used by other people
- You have two choices
  - Option #1: Provide Person class
  - Option #2: Provide PersonInterface and PersonImpl class
- Choose Option #2 for most cases
  - Other people do need just interface at compile time
  - You can change the implementation without affecting them
- Choose Option #1
  - When the class will be used rarely and do not change

# Lab:

**Exercise 2: Interface as a Type**  
**1027\_javase\_interface.zip**





# Implementing Multiple Interfaces

# Relationship of an Interface to a Class

- A concrete class can only extend one super (parent) class through inheritance relationship, but **it can implement multiple Interfaces**
  - The Java programming language does not permit multiple inheritance, but interfaces provide an alternative
- All abstract methods of **all** interfaces must be implemented by the concrete class, however
  - Otherwise, a compile error will occur

# Example: Implementing Multiple Interfaces

- A concrete class extends one super class but multiple Interfaces:

```
public class ComputerScienceStudent
    extends Student
    implements PersonInterface,
               AnotherInterface,
               ThirdInterface{

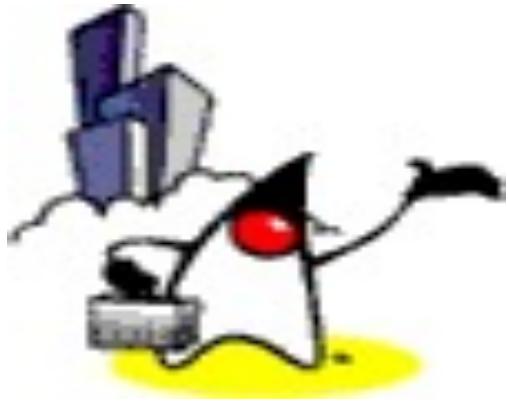
    // All abstract methods of all interfaces
    // need to be implemented.

}
```

# Lab:

**Exercise 3: Multiple Interfaces**  
**1027\_javase\_interface.zip**





# Inheritance Among Interfaces

# Inheritance Among Interfaces

- Interfaces are not part of the class hierarchy
- However, interfaces can have inheritance relationship among themselves

```
public interface PersonInterface {  
    void doSomething();  
}  
  
public interface StudentInterface  
        extends PersonInterface {  
    void doExtraSomething();  
}
```

# Lab:

**Exercise 4: Inheritance Among Interfaces**  
**1027\_javase\_interface.zip**





# Rewriting Interfaces

# Problem of Rewriting an Existing Interface

- Consider an interface that you have developed called *Dolt*:

```
public interface Dolt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
}
```

- Suppose that, at a later time, you want to add a third method to *Dolt*, so that the interface now becomes:

```
public interface Dolt {  
    void doSomething(int i, double x);  
    int doSomethingElse(String s);  
    boolean didItWork(int i, double x, String s);  
}
```

- If you make this change, however, all classes that implement the old *Dolt* interface will break because they don't implement all methods of the new *Doit* interface anymore

# Solution#1 of Rewriting an Existing Interface

- Create another interface
- For example, you could create a *DoltAnother* interface

```
public interface DoltAnother {  
    boolean didItWork(int i, double x, String s);  
}
```
- Now, old code of *Dolt* interface still work while new code can be written using the *Dolt* and *DoltAnother* interfaces

# Solution#2 of Rewriting an Existing Interface

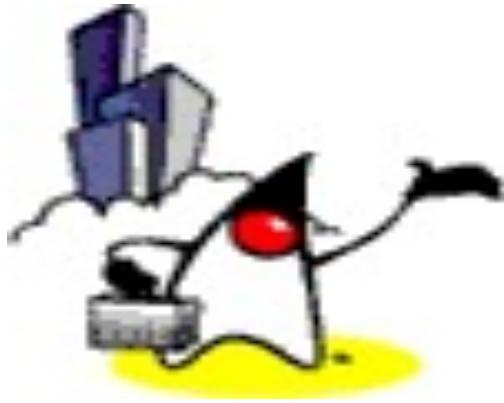
- Create a child interface from an existing interface
- For example, you could create a *DoltPlus* interface that extends *Dolt*:

```
public interface DoltPlus extends Dolt {  
    boolean didItWork(int i, double x, String s);  
}
```
- Now, old code of *Dolt* interface still work while new code can be written using the *DoltPlus* interface

# Lab:

**Exercise 5: Rewriting Interfaces**  
**1027\_javase\_interface.zip**





# When to Use an Abstract Class over an Interface?

# When to use an Abstract Class over Interface?

- For non-abstract methods, you want to use them **when you want to provide common implementation code for all subclasses**
  - Reducing duplication of code
- For abstract methods, the motivation is the same with the ones in the interface – to impose a common behavior for all child-classes without dictating how to implement it
- Remember a concrete class can extend only one super class whether that super class is in the form of concrete class or abstract class

# Code with Passion!

