

Object Oriented Design

Dr. Anirban Ghosh

School of Computing
University of North Florida



Inheritance

What is it?

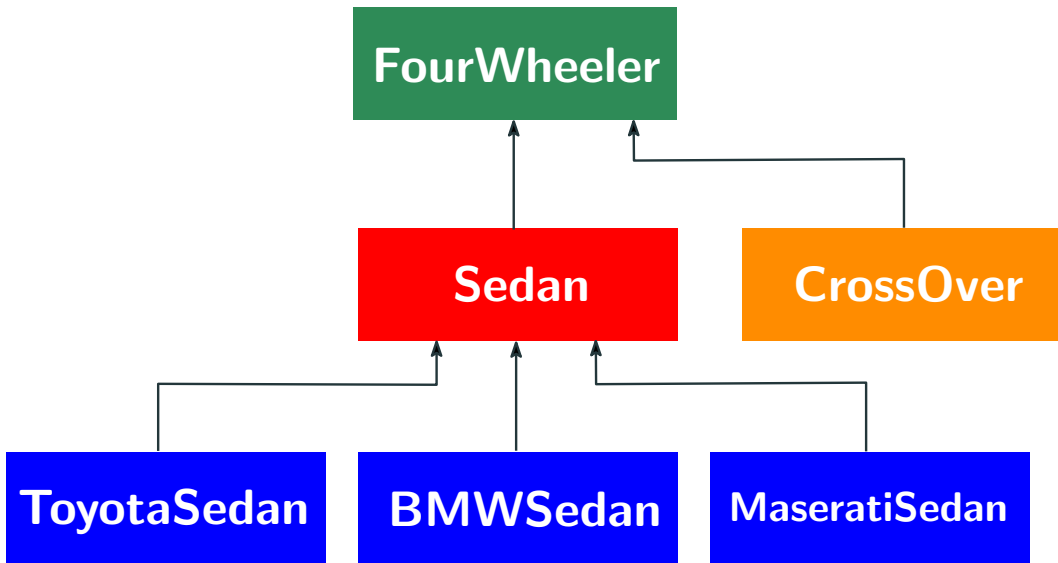
It is a mechanism by which a class can **inherit** some properties (variables and methods) of another related (coder decides) class or classes

👉 Helps to establish **meaningful relations** between various objects in a problem and also **reduces redundancy** in code

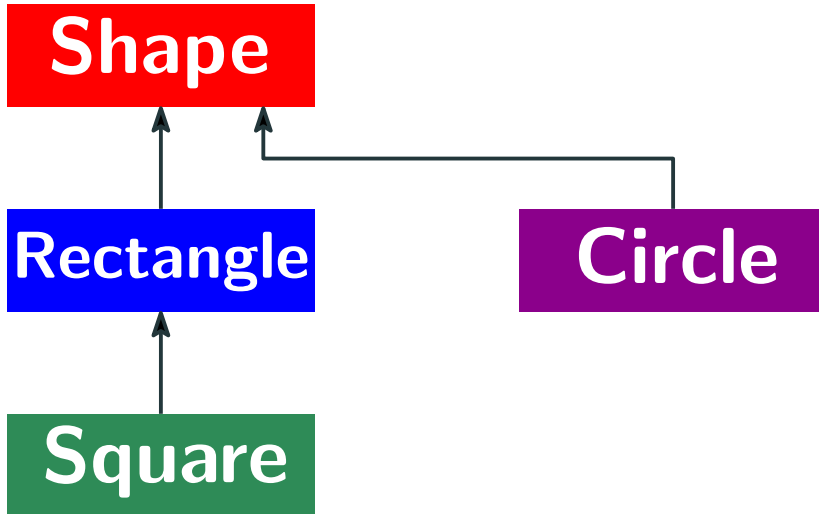
Parent and subclasses

If a class X **inherits** (can use) some properties (variables and methods) of another class Y, then with respect to this relationship, X is the **subclass/child class** and Y is the **parent** class

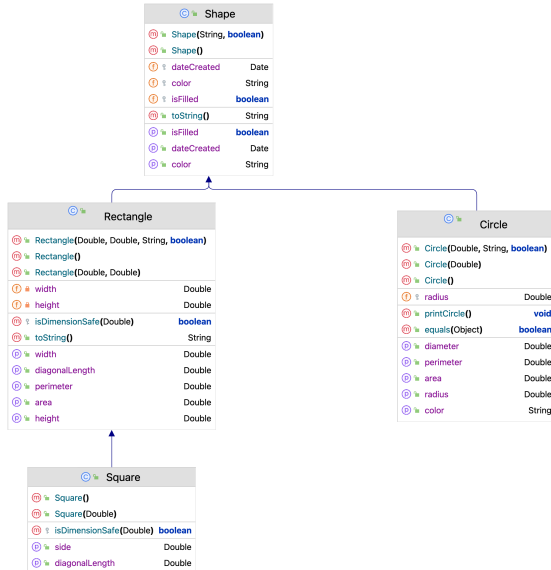
An example of inheritance



Another example



UML (Unified Modeling Language) diagram generated using IntelliJ



Generating UML diagrams in IntelliJ

- ➊ Select the related classes and interfaces from the left pane
- ➋ Right click
- ➌ Click on Diagrams
- ➍ Then, click on Show Diagram

The Shape class

```
public class Shape {

    protected String color;
    protected boolean isFilled;
    protected final Date dateCreated; // once an object is created we cannot change its date of creation

    public Shape() {
        color = "black";
        isFilled = false;
        dateCreated = new Date();
        System.out.println("No arg constructor in Shape is being executed.");
    }

    public Shape(String color, boolean filled) {
        this.color = color;
        isFilled = filled;
        dateCreated = new Date();
        System.out.println("Shape(String color, boolean filled) constructor in Shape is being executed.");
    }

    public String getColor() {
        System.out.println("getColor() from Shape class is being executed");
        return color;
    }

    // continued on the next slide ...
}
```

The Shape class

```
public class Shape {  
  
    // continued from the previous slide  
  
    public void setColor(String color) { this.color = color; }  
  
    public boolean isFilled() { return isFilled; }  
  
    public void setFilled(boolean filled) { this.isFilled = filled; }  
  
    final public Date getDateCreated() { // a child class cannot redefine this method since it is declared as final  
        return dateCreated;  
    }  
  
    public String toString() {  
        String str = "";  
        str += "COLOR: " + this.color + ", IS_FILLED: ";  
        str += this.isFilled + ", DATE_CREATED: ";  
        str += this.dateCreated;  
        return str;  
    }  
}
```


The Rectangle class

```
public class Rectangle extends Shape { // a child class of the Shape class
    private Double height, width; // not accessible to any other class

    public Rectangle() {
        super("dark green",true);
        System.out.println("No arg constructor in Rectangle is being executed.");
        height = 1.0;
        width = 2.0;
    }

    public Rectangle(Double height, Double width, String color, boolean filledOrNot) {
        super(color,filledOrNot);
        System.out.println("Rectangle(Double height, Double width, String color, boolean filled) constructor in Circle
                           is being executed.");
        if(!isDimensionSafe(height) || !isDimensionSafe(width) )
            throw new IllegalArgumentException("Impermissible side-length.");

        this.height = height;
        this.width = width;
    }
    // continued on the next slide ...
}
```

The Rectangle class

```
public class Rectangle extends Shape {  
    // continued from the previous slide  
  
    public Rectangle(Double height, Double width) {  
        System.out.println("Rectangle(Double height, Double width) constructor in Circle is being executed.");  
        if(!isDimensionSafe(height) || !isDimensionSafe(width) )  
            throw new IllegalArgumentException("Impermissible side-length.");  
  
        this.height = height;  
        this.width = width;  
    }  
  
    public Double getWidth() {    return width; }  
  
    public void setWidth(Double width) {  
        if(!isDimensionSafe(width) )  
            throw new IllegalArgumentException("Impermissible side-length.");  
  
        this.width = width;  
    }  
    // continued on the next slide ...  
}
```

The Rectangle class

```
public class Rectangle extends Shape {
    // continued from the previous slide
    public Double getHeight() { return height; }

    public void setHeight(Double height) {
        if(!isDimensionSafe(height)) throw new IllegalArgumentException("Critical runtime failure.");
        this.height = height;
    }

    public Double getArea() { return height * width; }
    public Double getPerimeter() { return 2 * (height + width); }

    public String toString() {
        String str = super.toString();
        str += ", HEIGHT: " + height + ", ";
        str += "WIDTH: " + width;
        return str;
    }

    protected boolean isDimensionSafe(Double length) { return length <= 100; }

    public Double getDiagonalLength() {
        System.out.println("getDiagonalLength from Rectangle");
        return Math.sqrt( Math.pow(height, 2) + Math.pow(width, 2) );
    }
}
```

The Square class

```
public class Square extends Rectangle { // a child class of the Rectangle; this class does not have any instance variable!
    public Square() {
        super(1.0,1.0);
        System.out.println("No arg constructor in Square is being executed.");
    }
    public Square(Double side) {
        System.out.println("Square(Double side) constructor in Square is being executed.");
        if(!isDimensionSafe(side))
            throw new IllegalArgumentException("Impermissible side-length.");
        super.setHeight(side);    super.setWidth(side);
    }
    public void setSide(Double side) {
        if(!isDimensionSafe(side))
            throw new IllegalArgumentException("Impermissible side-length.");
        super.setHeight(side);    super.setWidth(side);
    }
    public Double getSide() { return super.getHeight(); }

    @Override
    protected boolean isDimensionSafe(Double side) { return side <= 50; }

    @Override
    public Double getDiagonalLength() {
        System.out.println("getDiagonalLength from Square");
        return Math.sqrt(2) * getSide();
    }
}
```

The Circle class

```
public class Circle extends Shape {
    protected Double radius;

    public Circle() {
        this(1.0);
        System.out.println("No arg constructor in Circle is being executed.");
    }

    public Circle(Double radius) {
        if(radius < 0)
            throw new IllegalArgumentException("Radius cannot be negative.");

        this.radius = radius;
        System.out.println("Circle(Double radius) constructor in Circle is being executed.");
    }

    public Circle(Double radius, String color, boolean filled) {
        this.radius = radius;
        this.color = color;
        isFilled = filled;
        System.out.println("Circle(Double radius, String color, boolean filled) constructor in Circle is being executed.");
    }

    public Double getRadius() { return radius; }
    // continued on the next slide ...
}
```

The Circle class

```
public class Circle extends Shape {  
    // continued from the previous slide  
    public void setRadius(Double radius) { this.radius = radius; }  
  
    @Override  
    public String getColor() {  
        System.out.println("getColor() from Circle class is being executed");  
        return color;  
    }  
  
    public Double getArea() { return Math.PI * radius * radius; }  
    public Double getPerimeter() { return 2 * Math.PI * radius; }  
    public Double getDiameter() { return 2 * radius; }  
  
    // Gives compilation error since this method is already declared to be final in the superclass Shape  
    //public Date getDateCreated() { return dateCreated; }  
  
    public void printCircle() {  
        System.out.println("This circle this created on " + getDateCreated() + " having radius " + radius);  
    }  
  
    public boolean equals(Object ob) {  
        if (ob instanceof Circle) return Math.abs(radius - ((Circle) ob).radius) < 0.000001;  
        else return false;  
    }  
}
```

Access modifiers

Modifier	Class	Package	Subclass	World
public	<i>Y</i>	<i>Y</i>	<i>Y</i>	<i>Y</i>
protected	<i>Y</i>	<i>Y</i>	<i>Y</i>	<i>N</i>
<i>none used</i>	<i>Y</i>	<i>Y</i>	<i>N</i>	<i>N</i>
private	<i>Y</i>	<i>N</i>	<i>N</i>	<i>N</i>

<https://docs.oracle.com/javase/tutorial/java/java00/accesscontrol.html>


Important things in inheritance

- **extends** keyword: used to define a child class
- the 3 access modifiers: **public**, **protected**, **private**
- if X is a child class of class Y , then when an object of type X is created, first, a constructor from the Y class is invoked
- **super** constructor: used to invoke a constructor from parent class
- for invoking a constructor from the same class use the **this** constructor
- **instanceof** operator: used to check whether an object is of a specified type (note that the left operand, which is a reference variable, must be initialized)
- the **@Override** annotation: an optional annotation used by developers to declare that a certain method is redefined/overridden in some child class
- define a method using the **final** keyword if you do not the method to be overridden by a child class

The Object class

Object class

It is a parent class of every other Java class in this universe! This means no matter which Java class you are working with, **inheritance** is always present.

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html>

println(Object x) and print(Object x)

These two methods call the **toString()** method from the class of **x** for printing. In case, a **toString** method has not been defined inside the class, the **toString()** method from the **Object** class is called that creates some string representation of the object from printing purposes. Such representations look like **introduction.Point@36baf30c** and usually are not very useful to users.

equals() method

Checks if two objects have the exact same content. If the **equals()** method is not defined inside a class, the **equals()** method from the **Object** class is used whenever needed for equality checks.

The Number class

Class Number

java.lang.Object
java.lang.Number

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, DoubleAccumulator, DoubleAdder, Float, Integer, Long, LongAccumulator, LongAdder, Short

Classes such as Double, Integer, etc are child classes of the Number class, which is again a child class of the Object class

Reference. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Number.html>

Inheritance examples from Java's library

The IllegalArgumentException class

```
public double findSquareRootOf(double i) {  
    if(i < 0) throw new IllegalArgumentException("Hey, i cannot be negative!");  
    ....  
}
```

Class IllegalArgumentException

```
java.lang.Object  
  java.lang.Throwable  
    java.lang.Exception  
      java.lang.RuntimeException  
        java.lang.IllegalArgumentException
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

IllegalChannelGroupException, IllegalCharsetNameException, IllegalFormatException, IllegalSelectorException, IllegalThreadStateException, InvalidKeyException, InvalidOpenTypeException, InvalidParameterException, InvalidPathException, InvalidStreamException, KeyAlreadyExistsException, NumberFormatException, PatternSyntaxException, ProviderMismatchException, UnresolvedAddressException, UnsupportedAddressTypeException, UnsupportedCharsetException

Reference. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/IllegalArgumentException.html>

Other exceptions you should be aware of in this course

- **ArithmeticException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ArithmeticException.html>

- **NumberFormatException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/NumberFormatException.html>

- **IllegalArgumentException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/IllegalArgumentException.html>

- **NullPointerException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/NullPointerException.html>

- **IOException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/IOException.html>

- **IndexOutOfBoundsException**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/IndexOutOfBoundsException.html>

Good programmers usually can anticipate the possible exceptions in advance and they take care of those in their code by writing a few extra lines of code for exception handling

Generic classes

A generic class is a class that is parameterized over types

- The built-data structures in Java are all generic
- For instance, the `ArrayList` class is generic.

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html>

- `public class ArrayList<E> ... { ... }`
- This means one can declare an `ArrayList` of any data type
- `ArrayList<Point> points = new ArrayList<>();`
`ArrayList<Double> doubles = new ArrayList<>();`

👉 In this course, we will design generic classes for building the data structures

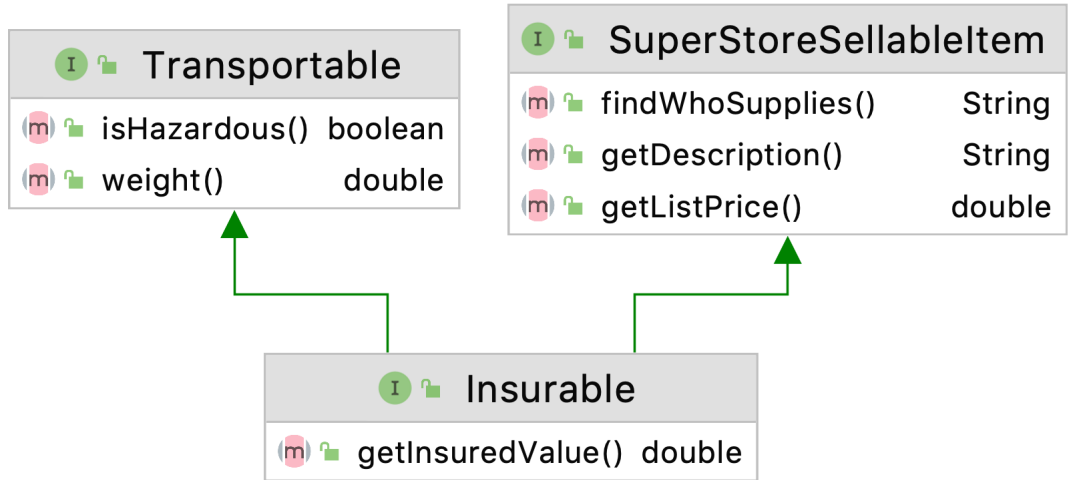
What is it?

An interface is a group of related methods with empty bodies; basically a starving class with no method bodies and possibly with no variables

Interfaces

```
public interface SuperStoreSellableItem {  
    String getDescription();  
    double getListPrice();  
    String findWhoSupplies();  
}
```

```
public class Photograph implements SuperStoreSellableItem{  
    final private String description, supplier;  
    final private double listPrice;  
  
    public Photograph(String description, double listPrice, String supplier) {  
        this.description = description;  
        this.listPrice = listPrice;  
        this.supplier = supplier;  
    }  
  
    public String getDescription() { return description; }  
    public double getListPrice() { return listPrice; }  
    public String findWhoSupplies() { return supplier; }  
}
```



Interfaces

```
import java.util.ArrayList;

public class TestPhotograph {
    public static void main(String[] args) {

        ArrayList<Photograph> wareHouse = new ArrayList<>();
        wareHouse.add(new Photograph("Dali Painting (Print)", 8753.67, "JAX Paintings"));
        wareHouse.add(new Photograph("Local Painting (Original)", 625.99, "Independent Painter"));

        System.out.print(wareHouse.get(0).getListPrice());
    }
}
```

Output

8753.67

Multiple inheritance and interfaces

```
public interface SuperStoreSellableItem {  
    String getDescription();  
    double getListPrice();  
    String findWhoSupplies();  
}
```

```
public interface Transportable {  
    double weight();  
    boolean isHazardous();  
}
```

```
public interface Insurable extends SuperStoreSellableItem, Transportable {  
    double getInsuredValue();  
}
```

Multiple inheritance and interfaces

```
public class BoxedItem implements Insurable {  
    private String description, supplier;  
    private double listPrice, weight;  
    private boolean haz;  
  
    public String getDescription() { return description; }  
    public double getListPrice() { return listPrice; }  
    public String findWhoSupplies() { return supplier; }  
  
    public double weight() { return weight; }  
    public boolean isHazardous() { return haz; }  
  
    public double getInsuredValue() { return listPrice * 1.5; }  
}
```

Interfaces

- We can declare variables inside an interface but they must be initialized; they are `static` and `final` by default
- We **cannot** define any method inside an interface
- Since an interface is not a class, we cannot create any object of its type
- An interface can inherit from multiple interfaces (not possible for Java classes)

Abstract classes

```
public abstract class AbstractSuperStoreSellableItem {  
    String description;  
  
    public String getDescription() { return description; }  
    public abstract double getListPrice();  
    public abstract String findWhoSupplies();  
}
```

👉 We **cannot** create an object of an abstract class

```
public class StoreSellablePhotograph extends AbstractSuperStoreSellableItem {  
    private String supplier;  
    private double listPrice;  
  
    public StoreSellablePhotograph(String description, double listPrice, String supplier) {  
        this.description = description;  
        this.listPrice = listPrice;  
        this.supplier = supplier;  
    }  
    public String getDescription() { return description; }  
    public double getListPrice() { return listPrice; }  
    public String findWhoSupplies() { return supplier; }  
}
```

Abstract classes

```
public class TestStoreSellablePhotograph {  
    public static void main(String[] args) {  
        ArrayList<StoreSellablePhotograph> wareHouse = new ArrayList<>();  
        wareHouse.add(new StoreSellablePhotograph("Dali Painting (Print)", 8753.67, "JAX Paintings"));  
        wareHouse.add(new StoreSellablePhotograph("Local Painting (Original)", 625.99, "Independent Painter"));  
  
        System.out.print(wareHouse.get(0).getDescription());  
    }  
}
```

Output

Dali Painting (Print)

Note

Interfaces and abstract classes can be generic too

- ☞ The popular `List` interface in Java is generic. See <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>.
- ☞ Generic abstract classes can be defined in the usual way like other non-abstract generic classes.

Interfaces vs abstract classes

- **Multiple inheritance** needed? Use interface, classes cannot help you
- **Want to write bodies** of some methods? Use an abstract class, interface won't allow to do this
- **Working with unrelated classes?** Use interface, otherwise use an abstract class

for-each loops

- Unlike traditional loops, for-each loops do not require a loop variable and is thus very easy to use
- Syntax is short and sweet; no worries about updating loop variables or fear of infinite loops
- ```
ArrayList<Point> points = new ArrayList<>();
for(Point p : points)
 System.out.println(p);
```
- For the built-in data structures, such loops works
- But what if you design your own data structure? How to make it iterable? The for-each loop won't know how to iterate over it
- **Solution.** make your class iterable by implementing the Iterable interface

## An example

- for-each loop does not work on String objects
- The reason is the String class is not iterable
- So, let us design our own class and make it iterable using the **Iterable** and **Iterator** interfaces that are already included in Java

### Iterable interface in Java

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Iterable.html>

### Iterator interface in Java

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Iterator.html>

# Code

```
public class IterableString implements Iterable<Character>{
 String str;
 public IterableString(String str) { this.str = str; }
 public Character getCharAt(int position) { return str.charAt(position); }
 public int length() { return str.length(); }

 public Iterator<Character> iterator() { return new IterableStringIterator(this); }

 public static class IterableStringIterator implements Iterator<Character> {
 int position;
 IterableString s;
 public IterableStringIterator(IterableString s) {
 this.s = s;
 position = 0;
 }

 public boolean hasNext() { return position < s.length(); }

 public Character next() {
 Character c = s.getCharAt(position);
 position++;
 return c;
 }
 public String toString(){ return str; }
 }
}
```

## Now for-each loop works!

```
var str = new IterableString("Data Structures");
for(var c : str)
 System.out.print(c);
```

## Another way to iterate

```
Iterator<Character> it = str.iterator();
while(it.hasNext())
 System.out.print(it.next());
```

## Built-in iterable string class in Java

StringBuilder



<https://docs.oracle.com/javase/tutorial/index.html>