Inheritance provides common functionality, but only to related classes. What if your classes are not related? For those occasions you use Interfaces.

An Interface provides common functionality, to any class, whether related or not. Used in event handling as you will soon see.

Abstract classes are special also. They cannot be instantiated and provide functionality that must be implemented in subclasses.
You need abstract classes

**Problem:**

A superclass provides behavior \((\text{interface})\)

A subclass inherits such behavior even if not applicable

A subclass **may** override methods to alter non-applicable inherited behavior
You need abstract classes

Superclass however, cannot guarantee that subclass will override a desired behavior

**Conclusion:**

Not all subclasses share the same behavior as superclass. *(This creates a maintenance issue)*

Superclass cannot guarantee a subclass will offer a behavior suitable to the subclass
You need abstract classes

Solution:
Create an abstract superclass

Provide behavior that all subclasses must offer (override)

You guarantee that concrete subclasses will have this behavior
Abstract classes enforce functionality

**GeometricObject**
- **-color:** String
- **-filled:** boolean
- **-dateCreated:** Date

**Abstract class**

#GeometricObject()
#GeometricObject(color: String, filled: boolean)
+toString(): String
+getArea(): double
+getPerimeter(): double

---

**Protected**

Functionality subclasses must implement.

---

Abstract classes enforce functionality. This means that no instances of this class can be created. It serves the purpose of providing common functionality to all GeometricObject subclasses.

The constructors are defined as protected (#). This prevents clients from creating objects, but still allows subclasses from initializing their super class properly.

The two abstract methods, getArea() and getPerimeter(), are defined for all GeometricObject objects, but implemented in each specific type of GeometricObject. In other words the subclasses know how to compute their area and perimeter.
Abstract methods must be implemented in subclass

<table>
<thead>
<tr>
<th>GeometricObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>- color: String</td>
</tr>
<tr>
<td>- filled: boolean</td>
</tr>
<tr>
<td>- dateCreated: Date</td>
</tr>
<tr>
<td># GeometricObject()</td>
</tr>
<tr>
<td># GeometricObject(color: String, filled: boolean)</td>
</tr>
<tr>
<td>+ toString(): String</td>
</tr>
<tr>
<td>+ getArea(): double</td>
</tr>
<tr>
<td>+ getPerimeter(): double</td>
</tr>
</tbody>
</table>

Subclasses must implement:

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- radius: double</td>
</tr>
<tr>
<td>+ Circle()</td>
</tr>
<tr>
<td>+ Circle(radius: double)</td>
</tr>
<tr>
<td>+ Circle(radius: double, color: String, filled: boolean)</td>
</tr>
</tbody>
</table>

The Circle class must implement the two abstract methods defined in GeometricObject.

In a UML class diagram, such abstract method definitions are omitted from the sub's class diagram. It is understood that the subclass will implement such methods.
package geometricobjectnotes;

import java.util.Date;

public abstract class GeometricObjectAbstract {
    private String color;
    private boolean filled;
    private Date dateCreated;

    /*-----------------------------------------------*/
    protected GeometricObjectAbstract() { this("white", false); }

    protected GeometricObjectAbstract(String color, boolean filled) {
        setColor(color);
        setFilled(filled);
        dateCreated = new Date();
    }
}
public final String getColor() { return color; }
public final boolean isFilled() { return filled; }
public final Date getDateCreated() { return dateCreated; }

public final void setColor(String color) { this.color = color; }
public final void setFilled(boolean filled) { this.filled = filled; }

public abstract double getArea();
public abstract double getPerimeter();

@Override
public String toString() {
    return "GeometricObjectAbstract{" +
            "color=" + getColor() +
            ", filled=" + isFilled() +
            ", dateCreated=" +
            getDateCreated() + "}";
}

package geometricobjectnotes;

public class Circle extends GeometricObjectAbstract {
    private double radius;

    public Circle() { this(0.0, "white", false); }
    public Circle(double radius) { this(radius, "white", false); }
    public Circle(double radius, String color, boolean filled) {
        super(color, filled);
        setRadius(radius);
    }

    public final double getRadius() { return radius; }
    public final void setRadius(double radius) { this.radius = radius; }

    @Override
    public double getArea() {
        double r = getRadius();
        return Math.PI * r * r;
    }
}
Code Deconstructed

@override
public double getPerimeter() {
    double r = getRadius();
    return 2 * Math.PI * r;
}
/*---------------------------------------------------------------*/
@override
public String toString() {
    return "Circle{" +
            "radius=" + radius +
            ", " + super.toString() +
            "}";
}
/*---------------------------------------------------------------*/
package geometricobjectnotes;

public class Main {
    public static void main(String[] args) {
        GeometricObjectAbstract circle1 = new Circle(5.0, "red", true);
        GeometricObjectAbstract circle2 = new Circle(5.0, "blue", false);

        System.out.println("circle1: "+ circle1);
        System.out.println("circle2: "+ circle2);

        double areaCircle1 = circle1.getArea();
        double areaCircle2 = circle2.getArea();

        System.out.println("\nArea Comparison:"");
        System.out.println("circle1: " + areaCircle1);
        System.out.println("circle2: " + areaCircle2);
        System.out.println("Same? " + (areaCircle1 == areaCircle2));
    }
}
run:
circle1: Circle{radius=5.0, GeometricObjectAbstract{color=red, filled=true, dateCreated=Sun Sep 29 08:34:22 CDT 2013}}
circle2: Circle{radius=5.0, GeometricObjectAbstract{color=blue, filled=false, dateCreated=Sun Sep 29 08:34:22 CDT 2013}}

Area Comparison:
circle1: 78.53981633974483
circle2: 78.53981633974483
Same? true
BUILD SUCCESSFUL (total time: 0 seconds)
An abstract class cannot be instantiated

// This will not work!
GeometricObject go = new GeometricObject();
A concrete class may not contain *abstract* methods.
An abstract class does not have to contain abstract methods

The class at this point serves only the purpose of a superclass. Of course no instances of it can be instantiated.
An abstract method appears as a prototype

    public abstract String sound();
An abstract class can be used as a type

GeometricObject[] object = new GeometricObject[100];

The polymorphic idea is still valid and actually quite stronger.
Each subclass is guaranteed to have the same basic common functionality.
An Interface accounts for behavior variability

**Problem:**
A superclass provides behavior (*interface*)

One subclass may inherit behavior as is

Another subclass may override to ignore it (*this is a maintenance headache*)

Interfaces are very similar in spirit to an abstract class, but they offer their common features to unrelated classes as well.

Whereas an abstract class creates a relationship among related classes, an interface creates relationships among related and also unrelated classes. Common features shared by a whole spectrum of classes.
An Interface accounts for behavior variability

Solution:
Create an Interface that offers changeable behavior

Subclasses that need this behavior can implement the interface

Subclasses that do not, won't implement the interface

Interfaces are very similar in spirit to an abstract class, but they offer their common features to unrelated classes as well.

Whereas an abstract class creates a relationship among related classes, an interface creates relationships among related and also unrelated classes. Common features shared by a whole spectrum of classes.
Interfaces are very similar in spirit to an abstract class, but they offer their common features to unrelated classes as well.

Whereas an abstract class creates a relationship among related classes, an interface creates relationships among related and also unrelated classes. Common features shared by a whole spectrum of classes.
Solid lines indicate class inheritance, dashed lines interface inheritance
Unrelated classes can inherit the same Interface

```java
public interface Edible {
    public abstract String howToEat();
}

public class Animal {}

public class Chicken extends Animal implements Edible {
    public String howToEat() { return "Fry the chicken."; }
}

public class Tiger extends Animal {}

public abstract class Fruit implements Edible {}

public class Apple extends Fruit {
    public String howToEat() { return "Make apple pie, yum!"; }
}
```

The Fruit class is abstract since it does not implement the howToEat() method of the Edible interface.

Notice how both the Chicken and Fruit classes implement the Edible interface. Since not all animals are edible, the Animal class does not implement the Edible interface, leaving it instead up to the concrete classes to do so.

On the other hand, since all fruit is considered edible, the Fruit class implements the interface Edible.

Implementing an interface is like inheriting from a class, thus the same concepts apply. Functionality in the interface is inherited by the sub-classes.
Interfaces can provide ordering capabilities

- Ordering indicates how one object orders against another \((is\ a\ before\ b?)\)

- Classes don't offer proper \((natural)\) ordering by default

- Use the **Comparable** Interface instead
Implement the Comparable Interface and override `compareTo()`

```java
public interface Comparable<T> {
    public int compareTo(T o);
}

// returns < 0 : if receiving less than o
// returns 0   : if receiving same as o
// returns > 0 : if receiving greater than o
```
public abstract class Employee implements Comparable<Employee> {
    // code omitted

    @Override
    public int compareTo(Employee o) {
        String name = getLast() + " " + getFirst();
        String employeeName = o.getLast() + " " +
                               o.getFirst();

        return name.compareTo(employeeName);
    }
}

• Strings are comparable by default
package geometricobjectnotes;

import java.util.Date;

public abstract class GeometricObjectAbstract implements Comparable<GeometricObjectAbstract> {

    /* The superclass will implement the Comparable Interface and
     * thus assure that all subclasses are comparable. Each subclass
     * must override the compareTo() method.
     */
    private String color;
    private boolean filled;
    private Date dateCreated;

    /**********************************************************/

The superclass is implementing the Interface so all subclasses can override the compareTo() method.
We want to make sure that all GeometricObjects are comparable.
public class Circle extends GeometricObjectAbstract {
    private double radius;
    /*Rest of the code is the same. The interface is inherited from*
    *the superclass. The method compareTo() is implemented to*
    *provide a natural ordering of circle objects. This ordering*
    *is based on area.*
    */
    @Override
    public int compareTo(GeometricObjectAbstract o) {
        /*Compares this to o according to area.*/
        Circle oCircle = (Circle) o;
        double thisArea = getArea();
        double oArea = oCircle.getArea();
        int compareResult;
        if (thisArea < oArea) { compareResult = -1; }
        else if (thisArea == oArea) { compareResult = 0; }
        else { compareResult = 1; }
        return compareResult;
    }
}

All the subclass has to do is override the compareTo() method
All the subclass has to do is override the compareTo() method
The `Cloneable` Interface allows cloning

/* This interface is empty. It is called a marker interface. */
public interface Cloneable {}

- The `Object` class offers the `clone()` method that does the cloning.

protected native Object clone() throws CloneNotSupportedException {}

// native: native to the JVM
public abstract class Employee implements Cloneable {
    ...
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Hourly extends Employee {
    // Hourly employees are now Cloneable

Note: The clone() method performs a shallow copy of the object, so if you need a true copy (deep) you will have to add your own customized code to do so.
Recap: Abstract Class

- An abstract class enforces functionality on related classes.
- An abstract method makes a class abstract.
- Abstract methods have no implementation in superclass.
- Abstract methods must be overridden in subclass.
Recap: Interface

- An interface enforces functionality on unrelated classes.
- An interface consists of static constants and abstract methods.
- Implement Cloneable to allow an object to be cloned.
- Implement Comparable to allow two objects to be compared.