

Lesson - 5: Inheritance & Interfaces

Inheritance

- Inheritance occurs when several classes are seen to naturally belong to the same general type – this is *generalization*.
- In the "figures" example (next slide), it seems natural to generalize the curves Triangle, Circle, Square
- Demo Package : `closedcurve.bad`

```

final class Triangle {
    final double base;
    final double height;
    Triangle(double base, double height){
        this.base = base;
        this.height=height;
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

final class Square {
    final double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return(side*side);
    }
}

final class Circle {
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}

```

//Illustrates a **non-OO** (= bad) way of computing areas

```

class Test {
    public static void main(String[]
args) {
        Object[] objects = {new Triangle(5,5,5),
                                new Square(3),
                                new Circle(3)};
        //compute areas
        for(Object o : objects) {
            if(o instanceof Triangle) {
                Triangle t = (Triangle)o;
                System.out.println(t.computeArea());
            }
            if(o instanceof Square) {
                Square s = (Square)o;
                System.out.println(s.computeArea());
            }
            if(o instanceof Circle) {
                Circle c = (Circle)o;
                System.out.println(c.computeArea());
            }
        }
    }
}

```

Points to observe

- Notice we can arrange Triangle, Square, Circle into an array of type Object[] by polymorphism. But Object does not have a computeArea() method, so we cannot polymorphically compute areas by using a single superclass method computeArea.
- Instead, if we use an array Object[], we have to repeatedly test the type of the Object in the array in order to execute the correct computeArea() method (using the instanceof operator).
- This approach needs improvement!
- Improvement shown by using Abstract class

Abstract Classes

- An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in subclasses.
- Some times you may create a class just to represent a concept rather than to represent the object.
- If a class is declared *abstract*, it cannot be instantiated.
- If a method is declared abstract, it cannot have a body -- it can only be declared.
- If a class has at least one abstract method, the class must be declared abstract.

Abstract Classes

- Abstract classes may include instance variables and other non-abstract (implemented) methods
- We use the modifier **abstract** on the class header to declare a class as abstract
- An abstract method is created by specifying the **abstract** type modifier. An abstract method contains no body and is, therefore, not implemented by the superclass.

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent
- An abstract method cannot be defined as `final` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate

- Can **generalize** the behavior of these geometric shape classes to support *polymorphic* access to a general `computeArea()` method.

```
abstract class ClosedCurve {  
    abstract double computeArea();  
}
```

See Demo pack : `closedcurve.good`
Demo Code : `AbstractDemo.java`


```

final class Triangle extends ClosedCurve {
    final double base;
    final double height;
    Triangle(double base, double height){
        this.base = base;
        this.height=height;
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

final class Square extends ClosedCurve{
    final double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return(side*side);
    }
}

final class Circle extends ClosedCurve{
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}

```

//This is the **OO** (= good) way of computing areas

```

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

```

```

        ClosedCurve[] objects = {
            new Triangle(5,5,5),
            new Square(3),
            new Circle(3)};

```

//compute areas

```

for(ClosedCurve cc : objects) {
    System.out.println(cc.compute
Area());

```

```

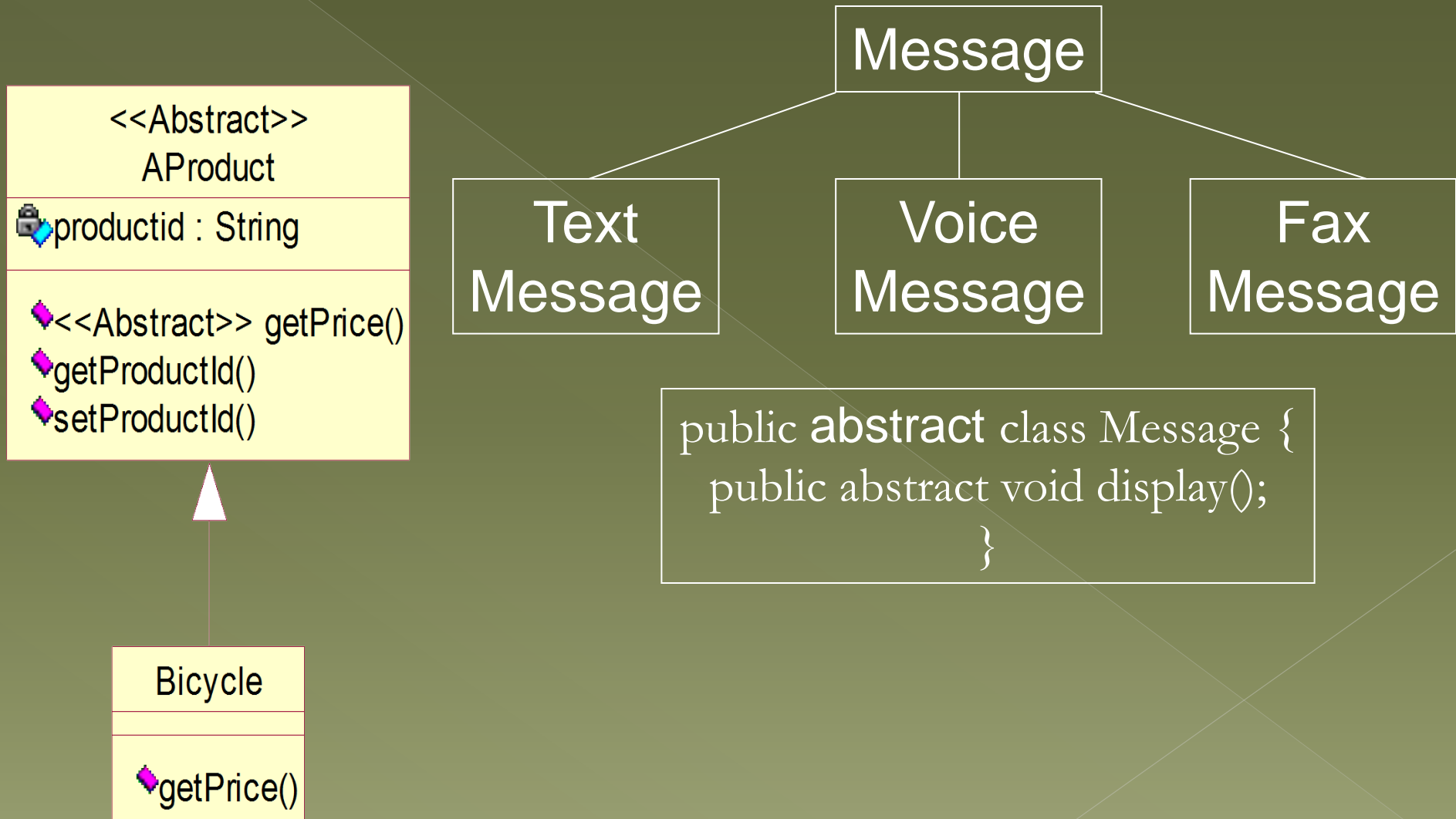
    }
}
}

```

Points to Observe

- No testing of types is required to access the `computeArea()` method
- New types of objects (such as `Rectangle`) can now be introduced by adding new subclasses to `ClosedCurve`. The only change to the code that is needed is inclusion of new instances in the `ClosedCurve[]` array, when it is initialized.
- This is an example of the **Open-Closed Principle**: a well-designed OO program is open to extension but closed to modification.

Other Example



```
public abstract class AProduct
{
    private String productid;

    public void setProductId (String id){
        productid=id;
    }
    public String getProductId (){
        return productid;
    }

    public abstract double getPrice();
}
```

```
public class Bicycle extends AProduct
{
    public double getPrice(){
        return 230.45;
    }
}
```

Not possible

AProduct mybicycle = new AProduct()

Bicycle mybicycle = new Bicycle()

Disabling Inheritance(final class and final method)

- The `final` class cannot be extended:

```
final class Math {  
    ...  
}
```

- The `final` variable is a constant. Value can not be altered.

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

- Benefits : **Security** – does not inherit by others.

Example

```
final class A {  
    final void meth1()  
    {  
        System.out.println("This is a final method.");  
    }  
}  
  
class B extends A // Error  
{  
    void meth1() { // ERROR! Can't override.  
        System.out.println("Illegal!");  
    }  
}
```

Summary about Abstraction

- A class may be declared abstract even if it does not have an abstract method
- Cannot create an object of an abstract class. However, you can declare a variable of the abstract class type and call methods using it.
- An abstract class cannot be declared as final.
- An abstract class should not declare constructors as private.
- An abstract method can not be declared static and private.

Tell the valid definition : e,f

Invalid definition : a b,c,d

```
class A {  
    abstract void unfinished() {  
    }  
}
```

(a)

```
public class abstract A {  
    abstract void unfinished();  
}
```

(b)

```
class A {  
    abstract void unfinished();  
}
```

(c)

```
abstract class A {  
    protected void unfinished();  
}
```

(d)

```
abstract class A {  
    abstract void unfinished();  
}
```

(e)

```
abstract class A {  
    abstract int unfinished();  
}
```

(f)

Interface

- A Java *interface* is a way of describing what classes should do, without specifying how they should do it.
- Instance variables and methods can occur. Methods should not have body.
- Interfaces can't have constructors because we can't instantiate them.
- By default any attribute of interface is **public**, **static** and **final**, so we don't need to provide access modifiers to the attributes but if we do, compiler doesn't complain about it either.
- By default interface methods are implicitly **abstract** and **public**, it makes total sense because the method don't have body and so that subclasses can provide the method implementation.

Interface

- ◉ An interface is declared using the *interface* keyword.
- ◉ A class that implements an interface uses the *implements* keyword rather than the *extends* keyword.
- ◉ Interface can have multiple methods.

Syntax to create an interface

```
<modifier> interface <interface name>
```

```
//public/default and abstract
```

```
{
```

```
    //interface fields(By default Public, Static  
    and Final)
```

```
        //interface methods(By default Public and  
        Abstract);
```

```
}
```

```
interface Speaker {  
    public void speak( );  
}
```

- Can implement more than one interface.

Syntax:

MyClass implements Intface1, Intface2,
Intface3

- Can also extend *and* implement.

Syntax:

MyClass extends SuperClass implements
Intface1, Intface2

```

final class Triangle implements ClosedCurve {
    final double base;
    final double height;
    Triangle(double base, double height){
        this.base = base;
        this.height=height;
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

final class Square implements ClosedCurve {
    final double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return(side*side);
    }
}

final class Circle implements ClosedCurve {
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}

```

```

public interface ClosedCurve {
    double computeArea();
}

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

        ClosedCurve[] objects = {

            new Triangle(5,5),
            new Square(3),
            new Circle(3)};

        //compute areas

        for(ClosedCurve cc : objects) {

            System.out.println(cc.computeArea());

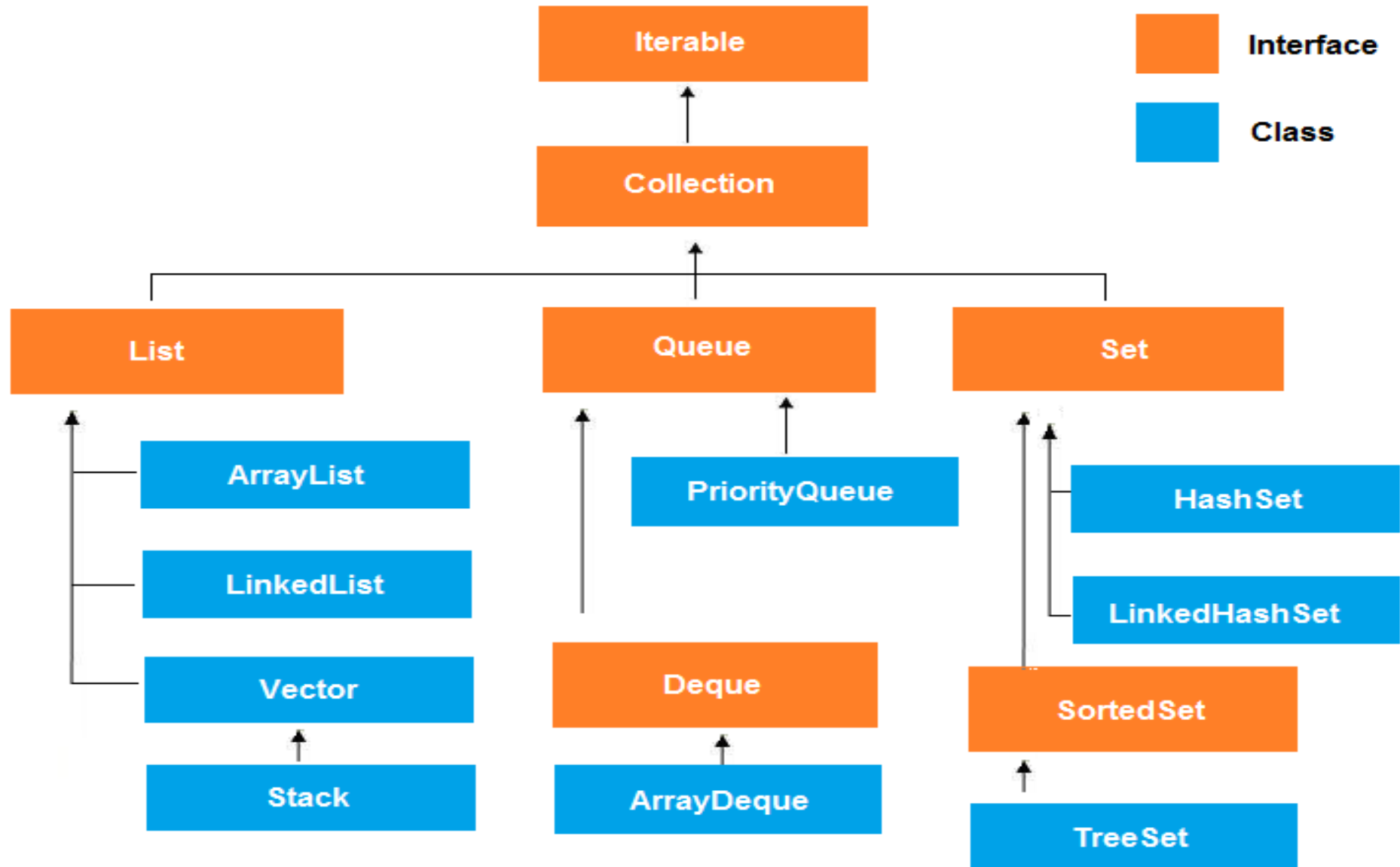
        }

    }
}

```

Note : This example illustrates the fact that a class that implements an interface may be as the type of that interface.

Predefined Interface Collection



Interfaces in Software Development

- *Interfaces* provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. But more than one *Interfaces can be implemented*.

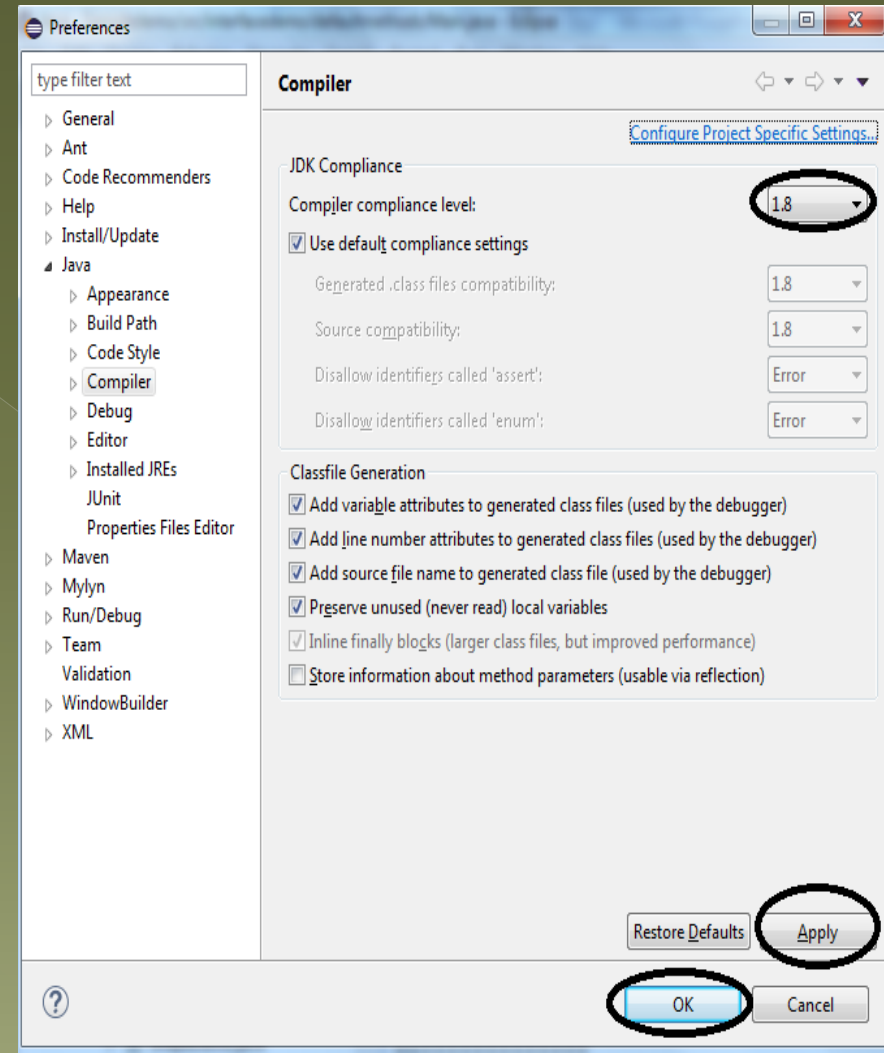
New Interface Features in Java 8

- Before Java 8, as we have seen, none of the methods in an interface had a method body; all were unimplemented.
- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.
 - A default method is a fully implemented method within an interface, whose declaration begins with the keyword `default`
 - A static method is a fully implemented method within an interface, having the same characteristics as any static method in a class.

Example – default methods

```
public interface NameAddress {  
    String getFirstName();  
    String getLastName();  
    String getStreet();  
    String getCity();  
    String getState();  
    default String getFullName() {  
        return getFirstName() + " " +  
getLastName();  
    }  
    default String getFullAddress() {  
        return getStreet() + "\n" +  
getCity() + ", " + getState();  
    }  
}
```

To Run Java8 Features



Rules About Default Methods confliction

- If a class implements an interface with a default method, that class inherits the default method (or can override it). However, there are rules to handle the cases in which two interfaces have the same method, or one interface and a superclass have the same method:
- Rule 1 : *Superclass vs Interface- Superclass wins!*
 - When a class extends a superclass and also implements an interface, and both super class and interface have a method with the same name, the superclass implementation wins – this is the version that is inherited by the class.
- Rule 2 : *Interface vs Interface – clash!*
 - If the inherited default method comes from multiple super interfaces , the method from the most specific super interface is inherited by the class

Conflicts

```
Interface I1{  
default void display(){  
System.out.println(" Interface I1");  
}  
}
```

```
class C1{  
void display(){  
System.out.println(" Class C1");  
}  
}
```

```
Interface I2{  
default void display(){  
System.out.println(" Interface I2");  
}  
}
```

```
// Interface conflicts  
Class Test implements I1, I2{  
}
```

```
// Super class & Interface conflicts  
Class Test extends C1 implements I1{  
}
```

Example – static methods

```
interface FPP
```

```
{
```

```
    static String CourseID(){  
        return "CS390";
```

```
    }
```

```
}
```

```
public class StaticMethodsDemo {
```

```
    public static void main(String args[])
```

```
    {
```

```
        System.out.println("Course ID for FPP is : " + FPP.CourseID());
```

```
    }
```

```
}
```

Interfaces in Java 7 and Java 8

Interview Question: What is the difference between an abstract class and an interface?

● Answer from the perspective of Java 7 (and before)

- Abstract classes may have fully implemented and unimplemented methods, but interfaces may not
- Abstract classes may contain static methods while interfaces may not
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public static final variables
- All methods in an interface are public and abstract, but abstract classes may have implemented methods of any visibility and abstract methods that are either public or protected

● Answer from the perspective of Java 8 (and after)

- Abstract classes may have fully implemented methods; interfaces may also have implemented methods, but they must bear the keyword “default” / “static”
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public final static variables
- All methods in an interface are public, but abstract classes may have implemented methods of any visibility and abstract methods that are either public or protected

MAIN POINT

- Interfaces are used in Java to specify publicly available services in the form of method declarations.
- A class that implements such an interface must make each of the methods operational.
- Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance.
- The concept of an interface is analogous to the creation itself – the creation may be viewed as an “interface” to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

DAY - 3

Introduction to the Reflection Library

- Reflection allows a program to inspect the contents of arbitrary objects at run time and to invoke arbitrary methods on them.
- Java have the API library to work with reflection is called **java.lang.reflect.***;

Example : ReflectionDemo.java

Challenge: Accessing Types Through Reflection

- In the ClosedCurve example, how can we make it so that not only is each area printed out in the for each loop, but also the *type* of closed curve, as in the following:

The area of this Triangle is 12.5

The area of this Square is 9.0

The area of this Circle is 28.274

- We do not want to test the type of each object in the array – this would undermine our implementation of polymorphism. How can we output the type of each object in a generic way?

```
public class Test {  
    public static void main(String[] args) {  
  
        ClosedCurve[] objects = {new Triangle(5,5), new Square(3), new Circle(3)};  
        //compute areas  
        for(ClosedCurve cc : objects) {  
            System.out.println(cc.computeArea());  
        }  
    }  
}
```

- Application: These features of the Class class allow us to solve the Challenge:

```
public class TestSecond {  
  
    public static void main(String[] args) {  
  
        ClosedCurve[] objects = {new Triangle(10,9,6), new Square(3),  
                                   new Circle(3)};  
        //compute areas  
        for(ClosedCurve cc : objects) {  
            String nameOfCurve =  
cc.getClass().getSimpleName());  
  
            System.out.println("The area of this "+nameOfCurve+ "  
is "+ cc.computeArea());  
        }  
    }  
}
```

MAIN POINT

The classes in the Java reflection package can be used to construct an instance of a class.

Likewise, reflection on the infinite creative power of consciousness reveals the truth of every thing and gives rise to the creation of any object.

The Object Class & Methods

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies
- Every class has access to the following methods :
 - > `public String toString()`
 - > `public boolean equals(Object o)`
 - > `public int hashCode()`
 - > `protected Object clone()` throws `CloneNotSupportedException`

The Object Class – toString()

- If a class does not override the default implementation of toString() given in the Object class, it produces output like the following:

```
public static void main(String[] args) {  
    System.out.println(new Object());  
}
```

//output

java.lang.Object@18d107f

This is a concatenation of the fully qualified class name with the hexadecimal version of the "hash code" of the object

- Most Java API classes override this default implementation of `toString()`. The purpose of the method is to provide a readable String representation which can be printed to the console of the state of an object.
- Example from the Exercises:

// the Account object has this implementation of `toString()`

```
public String toString(){  
    String newline = System.getProperty("line.separator");  
    String ret = "Account type: " + acctType + newline +  
    "Current bal: " + balance;  
        return ret;  
}
```

The Object Class – equals()

- Implementation in Object class:

`ob1.equals(ob2)`

if and only if `ob1 == ob2`

if and only if references point to the same object

- Using the '==' operator to compare objects is usually not what we intend (though for comparison of *primitives*, it is just what is needed). For comparing objects, the `equals` method should (usually) be overridden to compare the *states* of the objects.

Example

//an overriding equals method in the Person class

 @Override

```
public boolean equals(Object aPerson) {  
    if(aPerson == null) return false;  
    if(!aPerson instanceof Person) return false;  
    Person p = (Person)aPerson;  
    boolean isEqual =  
this.name.equals(p.name)  
    return isEqual;  
}
```


The hashCode() Method

Java provides every object with a hashCode() method that provides an integer representation of the object

What is important now is to know that hash codes must satisfy the following property:

If A is a class and a1 & a2 are instances of the class A and
a1.equals(a2) is true
then the following must be true.

$$a1.hashCode() == a2.hashCode()$$

This means the hashCode() method should always be overridden if equals is overridden.

The clone() Method

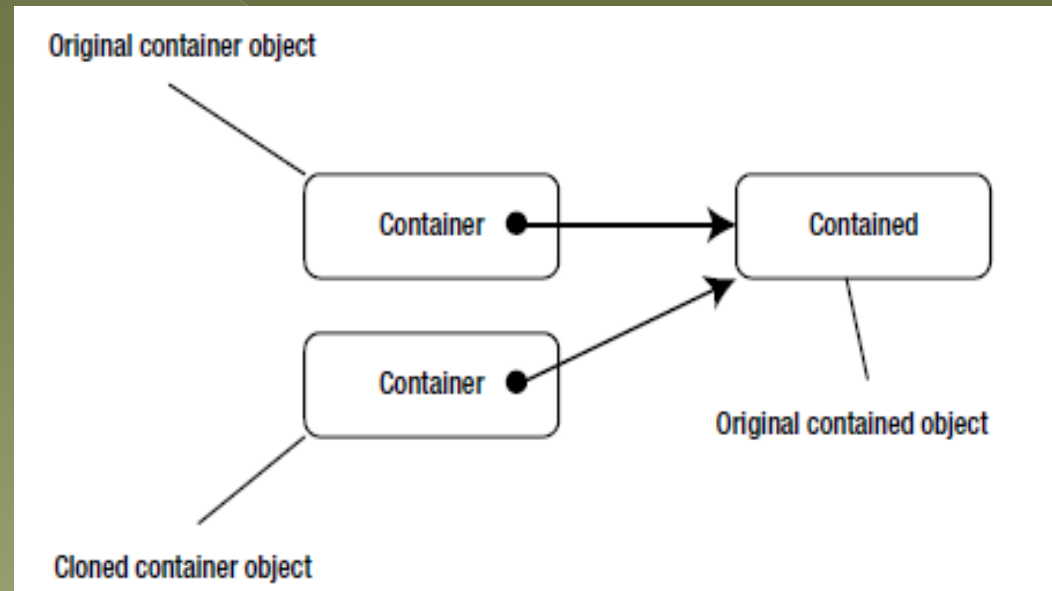
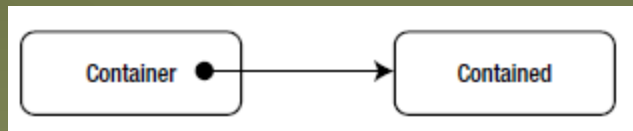
- Java does not provide an automatic mechanism to clone (make a copy) an object.
- Recall that when you assign a reference variable to another reference variable, only the reference of the object is copied, not the content of the object.
- Cloning an object means copying the content of the object bit by bit.
- If you want objects of your class to be cloned, you must reimplement the clone() method in your class. Once you reimplement the clone() method, you should be able to clone objects of your class by calling the clone() method.
- One important thing that you must do is add an "implements Cloneable" clause in your class declaration.
- Cloneable is an interface declared in the java.lang package.

The declaration of the clone() method in the Object class is as follows:

```
protected Object clone() throws CloneNotSupportedException
```

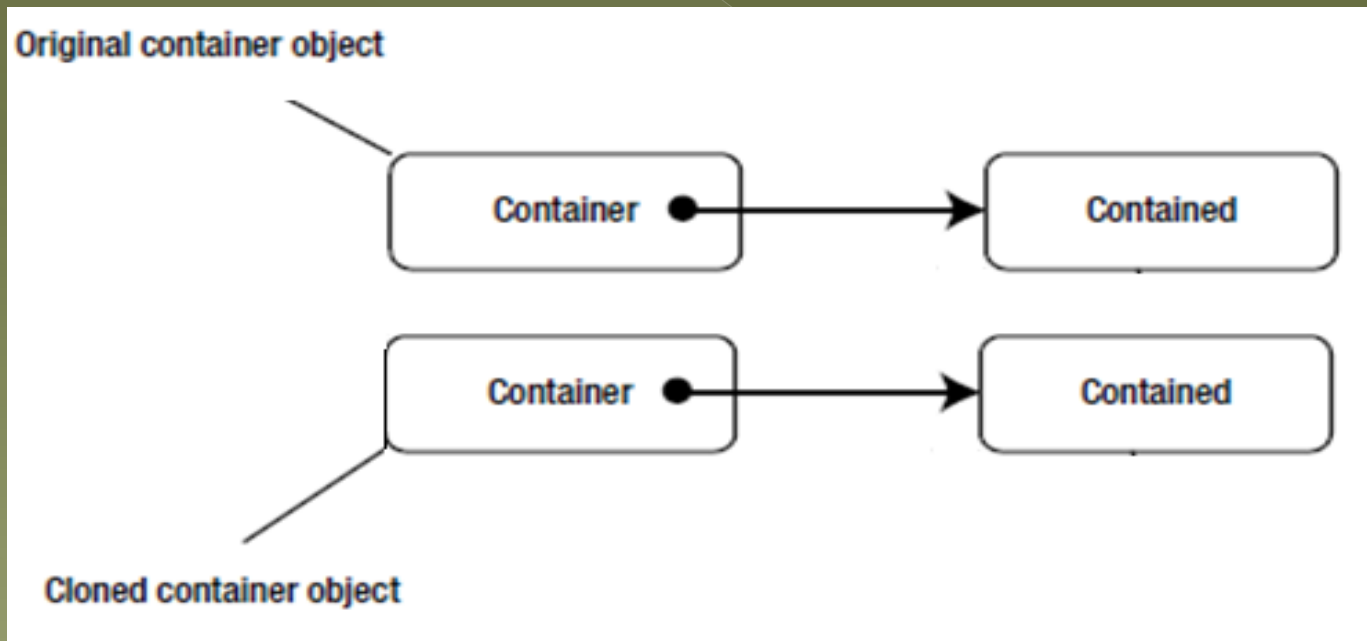
Cloning by Shallow copy

- An object may be composed of another object. In such cases, two objects exist in memory separately—a contained object and a container object.
- The container object stores the reference of the contained object.
- When you clone the container object, the reference of the contained object is cloned. After cloning is performed, there are two copies of the container object; both of them have references to the same contained object.
- This is called a shallow cloning because references are copied, not the objects.



Cloning by Deep copy

- When the contained objects are copied rather than their references during cloning of a compound object, it is called deep cloning.
- You must clone all the objects referenced by all reference variables of an object to get a deep cloning.



Class Relationships

- Object of one class may be related to the objects of another class in the following ways.
 - Is-A Relationship (Inheritance)
 - Has-A Relationship (Aggregation / Composition)
- A class can have reference to the object of other class as member is Has-A relationship. One form of software reuse is composition.
- Eg : Date class

PRACTICE