BIRZEIT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Instructor :Murad Njoum
Office : Masri322

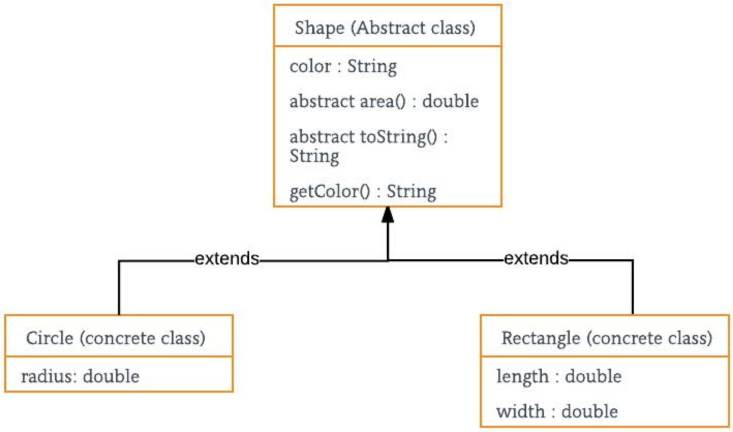
Chapter 13 Abstract Classes and Interfaces

Abstract Classes and Methods

- ✓ An **abstract class** is a class that is declared with abstract keyword.
- ✓ An **abstract method is a method** that is declared without an implementation.
- ✓ An abstract class **may or may not have all** abstract methods. Some of them can be concrete متماسك methods
- ✓ A **method defined abstract** must always be **redefined in the subclass**, thus making overriding compulsory(it must) OR either make subclass itself abstract.

- ✓ Any class that contains one or more abstract methods must also be declared with abstract keyword.
- ✓ There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
- ✓ An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, **leaving it to each subclass to fill in the details.**


Shape (Abstract class)
color : String
abstract area() : double
abstract toString() : String
getColor() : String

extends

extends

Circle (concrete class)
radius: double

Rectangle (concrete class)
length : double
width : double

```
abstract class Shape
{
   String color;

   // these are abstract methods
   abstract double area( );
   public abstract String toString( );

   // abstract class can have constructor
   public Shape(String color) {
      System.out.println(”Shape constructor called”);
      this.color = color;
   }

   // this is a concrete method
   public String getColor( ) {
      return color;
   }
}
```

```
class Circle extends Shape
{
   double radius;

   public Circle(String color,double radius) {

      // calling Shape constructor
      super(color);
      System.out.println(”Circle constructor called”);
      this.radius = radius;
   }

   @Override
   double area( ) {
      return Math.PI * Math.pow(radius, 2);
   }

   @Override
   public String toString( ) {
      return ”Circle color is ” + super.color +
                ”and area is : “ + area( );
   }
}
```

Instructor :Murad Njoum

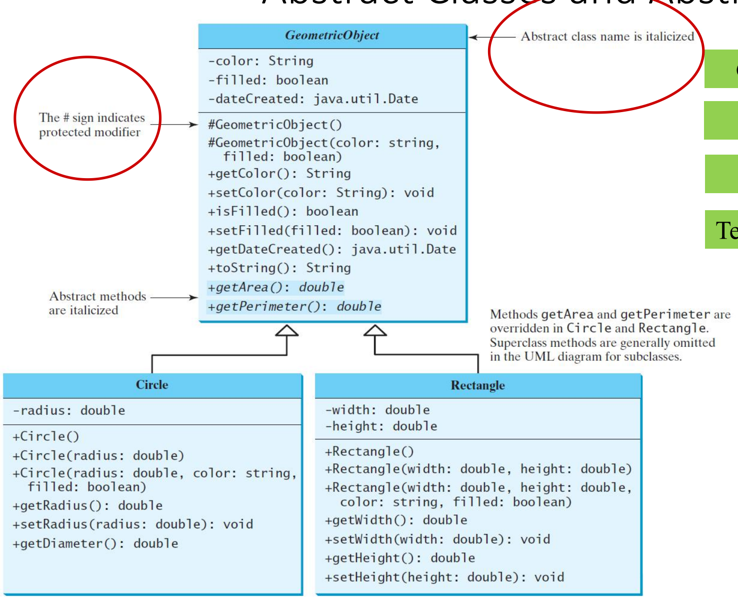
```
class Rectangle extends Shape{
     double length;
   double width;

   public Rectangle(String color,double length,double width) {
      // calling Shape constructor
      super(color);
      System.out.println(“Rectangle constructor called”);
      this.length = length;
      this.width = width;
   }

   @Override
   double area( ) {
      return length*width;
   }

   @Override
   public String toString( ) {
      return “Rectangle color is “ + super.color +
                  ”and area is : “ + area( );
   }

}
```

```
public class Test
{
 public static void main(String[] args)
 {
    Shape s1 = new Circle(“Red”, 2.2);
    Shape s2 = new Rectangle(“Yellow”, 2, 4);

    System.out.println(s1.toString( ));
    System.out.println(s2.toString( ));
 }
}
```

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Red and area is : 15.205308443374602 Rectangle color is
Yellow and area is : 8.0
```

Instructor :Murad Njoum

Abstract Classes and Abstract Methods



GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double

Abstract class name is italicized

The # sign indicates protected modifier

Abstract methods are italicized

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.

Circle

-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

GeometricObject

Circle

Rectangle

TestGeometricObject

Run

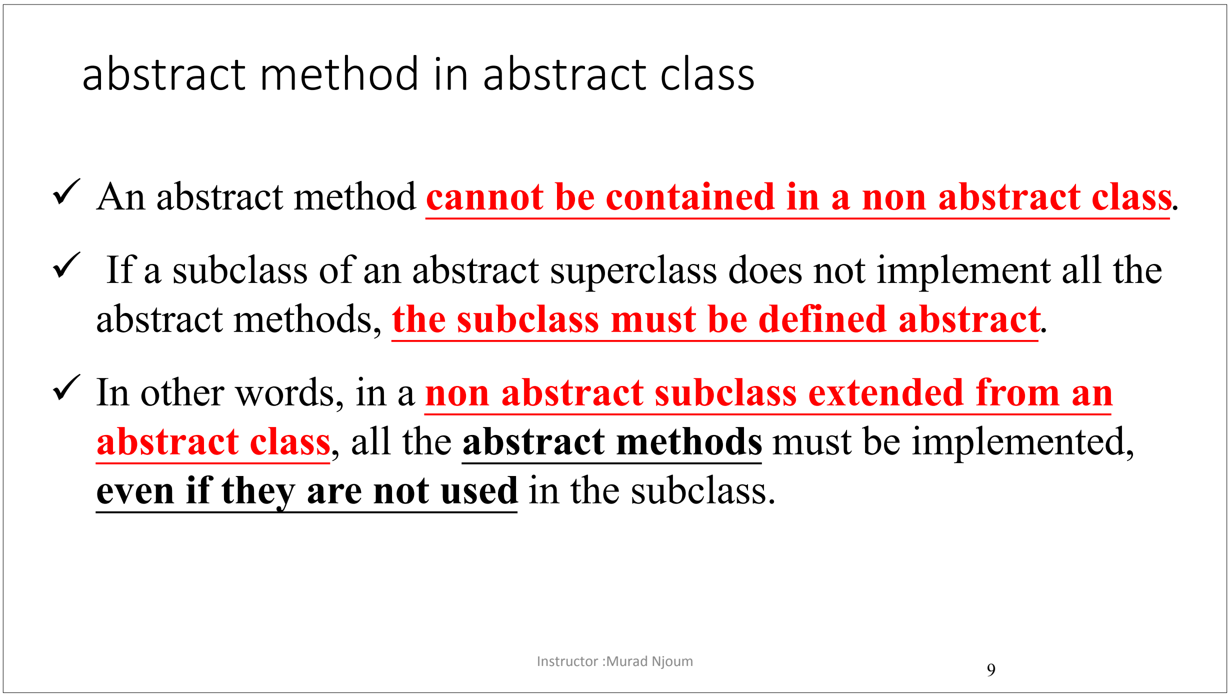
Encapsulation vs Data Abstraction

1.Encapsulation is data hiding (information hiding) while Abstraction is detail hiding(**implementation hiding**).

2.While encapsulation **groups together data and methods** that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

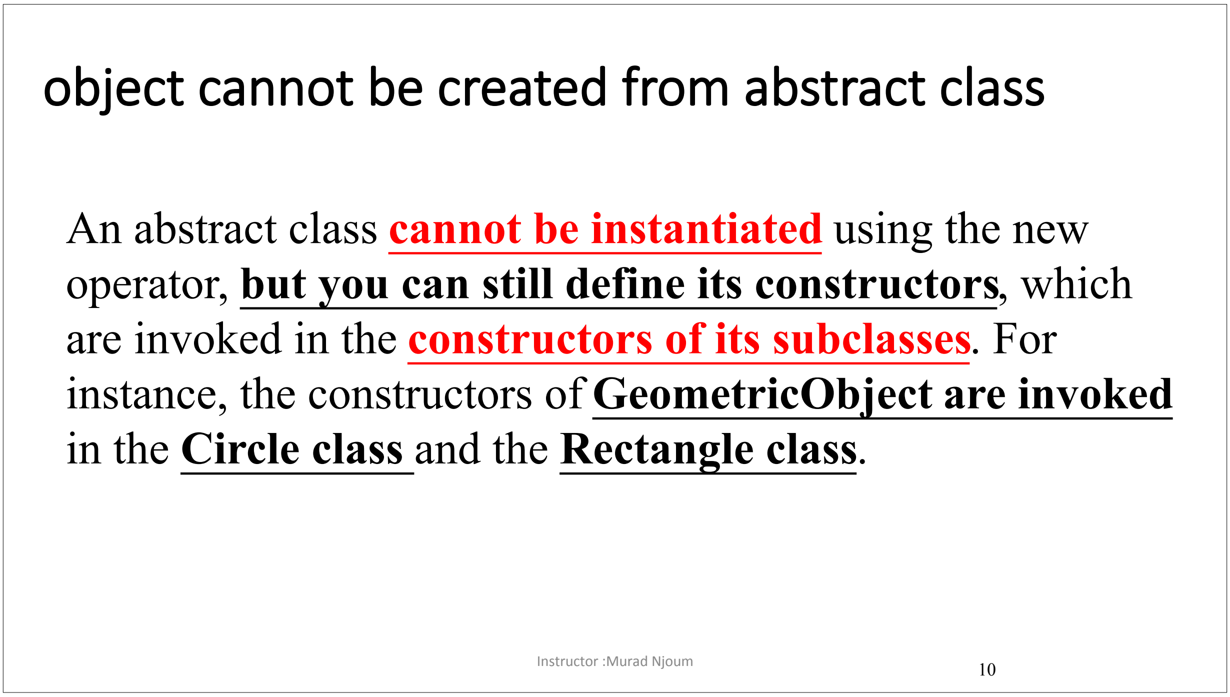
Advantages of Abstraction

1.It reduces the complexity of viewing the things.
2.Avoids code duplication and increases reusability.
3.Helps to increase security of an application or program as only important details are provided to the user.



abstract method in abstract class

- ✓ An abstract method **cannot be contained in a non abstract class**.
- ✓ If a subclass of an abstract superclass does not implement all the abstract methods, **the subclass must be defined abstract**.
- ✓ In other words, in a **non abstract subclass extended from an abstract class**, all the **abstract methods** must be implemented, **even if they are not used** in the subclass.



object cannot be created from abstract class

An abstract class **cannot be instantiated** using the new operator, **but you can still define its constructors**, which are invoked in the **constructors of its subclasses**. For instance, the constructors of **GeometricObject are invoked** in the **Circle class** and the **Rectangle class**.

abstract class without abstract method

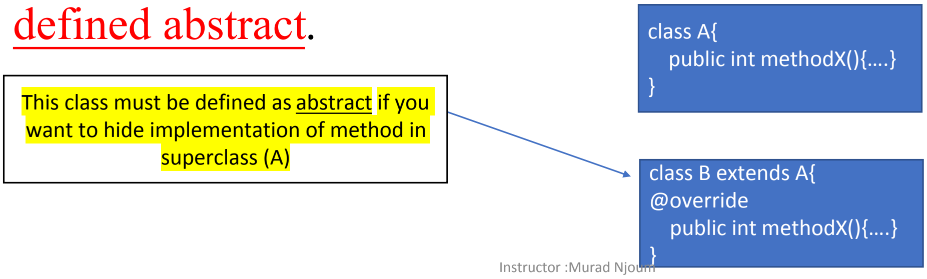
- □ A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods.
- □ In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

A subclass **can be abstract** even if its **superclass is concrete**. For example, the **Object class is concrete**, but its subclasses, such as GeometricObject, may be abstract.

concrete method overridden to be abstract

A subclass can **override a method** from its superclass to define it abstract. This is rare, but useful when the implementation of the method **in the superclass becomes invalid in the subclass**. In this case, the subclass must be defined abstract.



This class must be defined as abstract if you want to hide implementation of method in superclass (A)

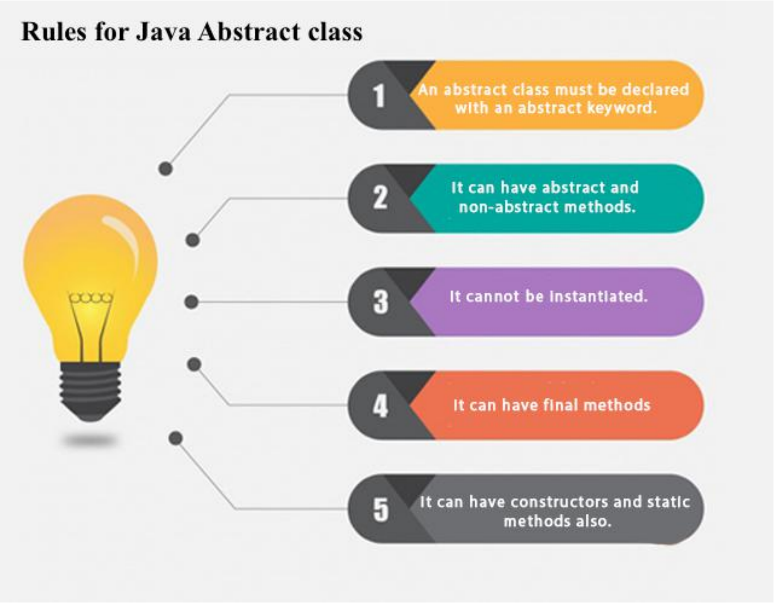
```
class A{
   public int methodX(){….}
}
```

```
class B extends A{
@override
   public int methodX(){….}
}
```

abstract class as type

You cannot create an instance from an abstract class using the new operator, but an abstract class can be **used as a data type**. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new    GeometricObject[10];
```



Rules for Java Abstract class

1 An abstract class must be declared with an abstract keyword.

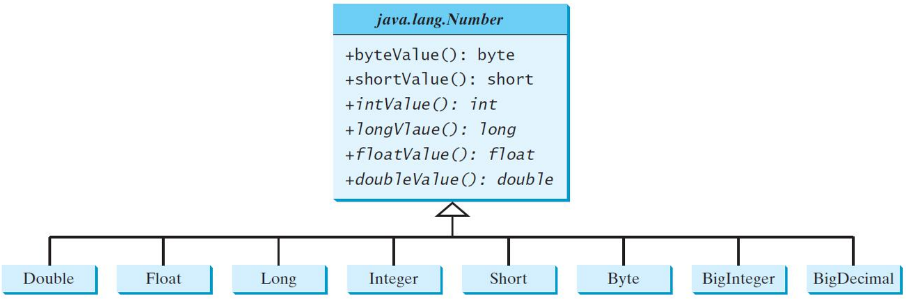
2 It can have abstract and non-abstract methods.

3 It cannot be instantiated.

4 It can have final methods

5 It can have constructors and static methods also.

Case Study: **the Abstract Number Class**

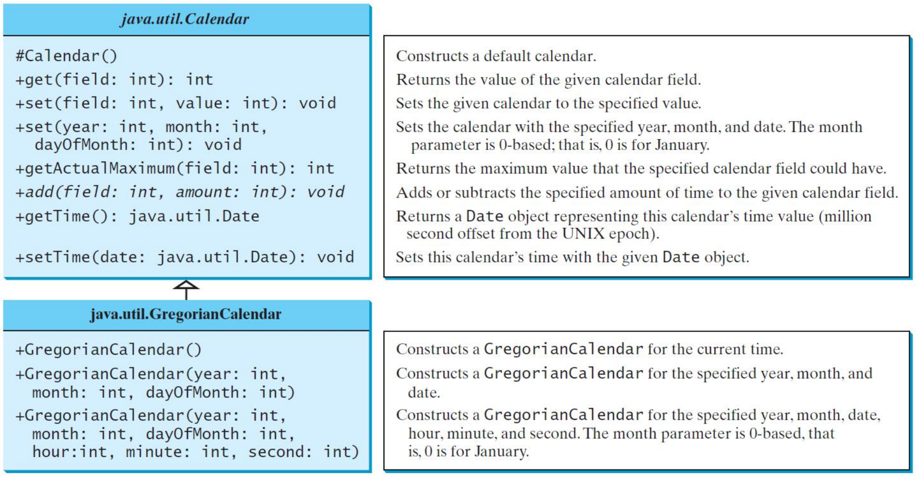


java.lang.Number

```
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue(): double
```

LargestNumbers Run

The Abstract Calendar Class and Its GregorianCalendar subclass



java.util.Calendar

`#Calendar()`	Constructs a default calendar.
`+get(field: int): int`	Returns the value of the given calendar field.
`+set(field: int, value: int): void`	Sets the given calendar to the specified value.
`+set(year: int, month: int, dayOfMonth: int): void`	Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.
`+getActualMaximum(field: int): int`	Returns the maximum value that the specified calendar field could have.
`+add(field: int, amount: int): void`	Adds or subtracts the specified amount of time to the given calendar field.
`+getTime(): java.util.Date`	Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).
`+setTime(date: java.util.Date): void`	Sets this calendar's time with the given `Date` object.

java.util.GregorianCalendar

`+GregorianCalendar()`	Constructs a `GregorianCalendar` for the current time.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int)`	Constructs a `GregorianCalendar` for the specified year, month, and date.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)`	Constructs a `GregorianCalendar` for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

The Abstract Calendar Class and Its GregorianCalendar subclass

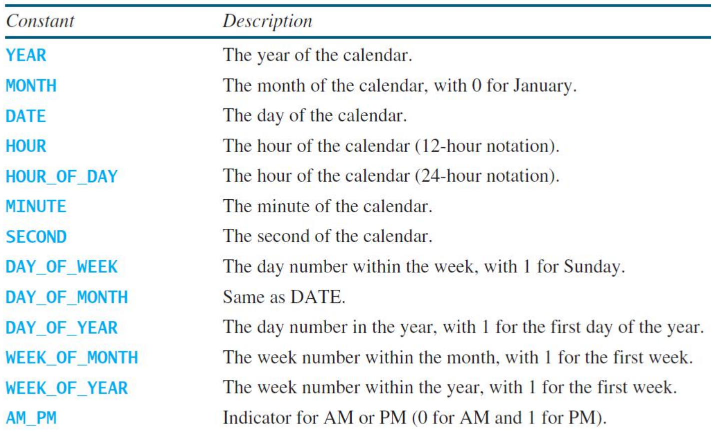
- ❖An instance of java.util.Date represents a specific instant in time with millisecond precision.
- ❖ **java.util.Calendar** is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object.
- ❖ Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.
- ❖Currently, java.util.GregorianCalendar for the Gregorian calendar is supported in the Java API.

The GregorianCalendar Class

- ✓You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time
- ✓use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date.
- ✓The month parameter is **0-based**, i.e., 0 is for January.

The get Method in Calendar Class

The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object. The fields are defined as constants, as shown in the following.



Constant	*Description*
YEAR	The year of the calendar.
MONTH	The month of the calendar, with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week, with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year, with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month, with 1 for the first week.
WEEK_OF_YEAR	The week number within the year, with 1 for the first week.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).

```
import java.util.*;

public class TestCalendar {
  public static void main(String[] args) {
    // Construct a Gregorian calendar for the current date and time
    Calendar calendar = new GregorianCalendar();

    System.out.println("Current time is " + new Date());
    System.out.println("YEAR: " + calendar.get(calendar.YEAR));
    System.out.println("MONTH: " + calendar.get(calendar.MONTH));
    System.out.println("DATE: " + calendar.get(calendar.DATE));
    System.out.println("HOUR: " + calendar.get(calendar.HOUR));
    System.out.println("HOUR_OF_DAY: " +
      calendar.get(Calendar.HOUR_OF_DAY));
    System.out.println("MINUTE: " + calendar.get(calendar.MINUTE));
    System.out.println("SECOND: " + calendar.get(calendar.SECOND));
    System.out.println("DAY_OF_WEEK: " +
      calendar.get(calendar.DAY_OF_WEEK));
```

```
System.out.println("DAY_OF_MONTH: " +    calendar.get(calendar.DAY_OF_MONTH));
    System.out.println("DAY_OF_YEAR: " +    calendar.get(calendar.DAY_OF_YEAR));
    System.out.println("WEEK_OF_MONTH: " +  calendar.get(calendar.WEEK_OF_MONTH));
    System.out.println("WEEK_OF_YEAR: " +    calendar.get(calendar.WEEK_OF_YEAR));
    System.out.println("AM_PM: " + calendar.get(calendar.AM_PM));

    // Construct a calendar for December 25, 1997
    Calendar calendar1 = new GregorianCalendar(1997, 11, 25);
    String[] dayNameOfWeek = {"Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"};
    System.out.println("December 25, 1997 is a " +
      dayNameOfWeek[calendar1.get(calendar1.DAY_OF_WEEK) - 1]);
  }
}
```

```
Current time is Thu Nov 12 15:56:21 IST 2020
YEAR: 2020
MONTH: 10
DATE: 12
HOUR: 3
HOUR_OF_DAY: 15
MINUTE: 56
SECOND: 21
DAY_OF_WEEK: 5
DAY_OF_MONTH: 12
DAY_OF_YEAR: 317
WEEK_OF_MONTH: 2
WEEK_OF_YEAR: 46
AM_PM: 1
December 25, 1997 is a Thursday
```

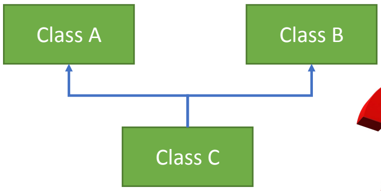
- Why we need an abstract class?
- Why can’t we create the object of an abstract class?
- What is the advantage of using an abstract class?

Does Java support Multiple Inheritance？

```
// First Parent class
class Parent1
{
        void fun()
        {
                System.out.println("Parent1");
        }
}

// Second Parent Class
class Parent2
{
        void fun()
        {
                System.out.println("Parent2");
        }
}
```

Compiler Error

```
// Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
public static void main(String args[])
{
        Test t = new Test();
        t.fun();
   }
}
```



class C extends A, B { …. }

The Diamond Problem:

```
      GrandParent
        /   \
       /     \
 Parent1     Parent2
       \     /
        \   /
         Test
```

```
// A Grand parent class in diamond
class GrandParent
{
        void fun()
        {
                System.out.println("Grandparent");
        }
}

// First Parent class
class Parent1 extends GrandParent
{
        void fun()
        {
                System.out.println("Parent1");
        }
}
```

```
// Second Parent Class
class Parent2 extends GrandParent
{
        void fun()
        {
                System.out.println("Parent2");
        }
}


// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
public static void main(String args[])
{
        Test t = new Test();
        t.fun();
}
}
```

Java and Multiple Inheritance:

- Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class.
- **The problem** occurs when there exist methods with same signature in both the **super classes and subclass**.
- On calling the method, the **compiler cannot determine** which class method to be called and even on calling which class method gets the priority

Simplicity –

- Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex.

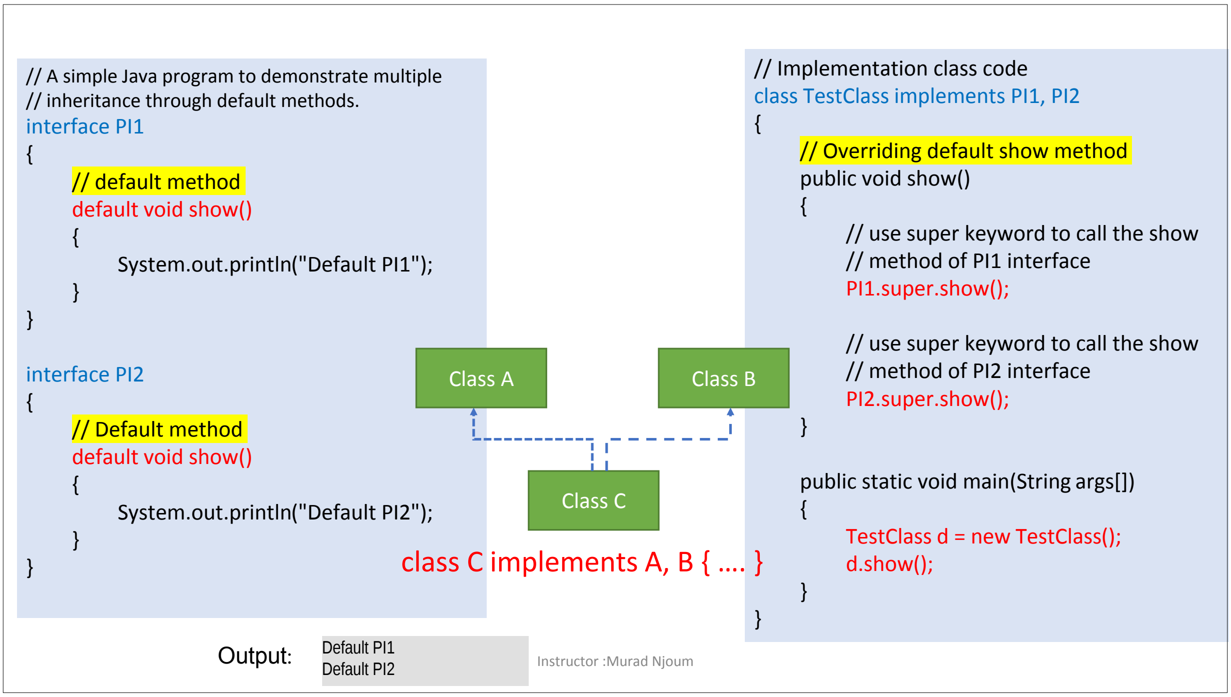
- It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

How are above problems handled for Default Methods and Interfaces ?

- Java 8 supports **default methods** where interfaces can provide default implementation of methods.

- And a class **can implement two or more interfaces**.

- In case both the implemented interfaces contain **default methods with same method signature**, the implementing class should explicitly specify which default method is to be used or it should override the default method.



```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface PI1
{
        // default method
        default void show()
        {
                System.out.println("Default PI1");
        }
}

interface PI2
{
        // Default method
        default void show()
        {
                System.out.println("Default PI2");
        }
}
```

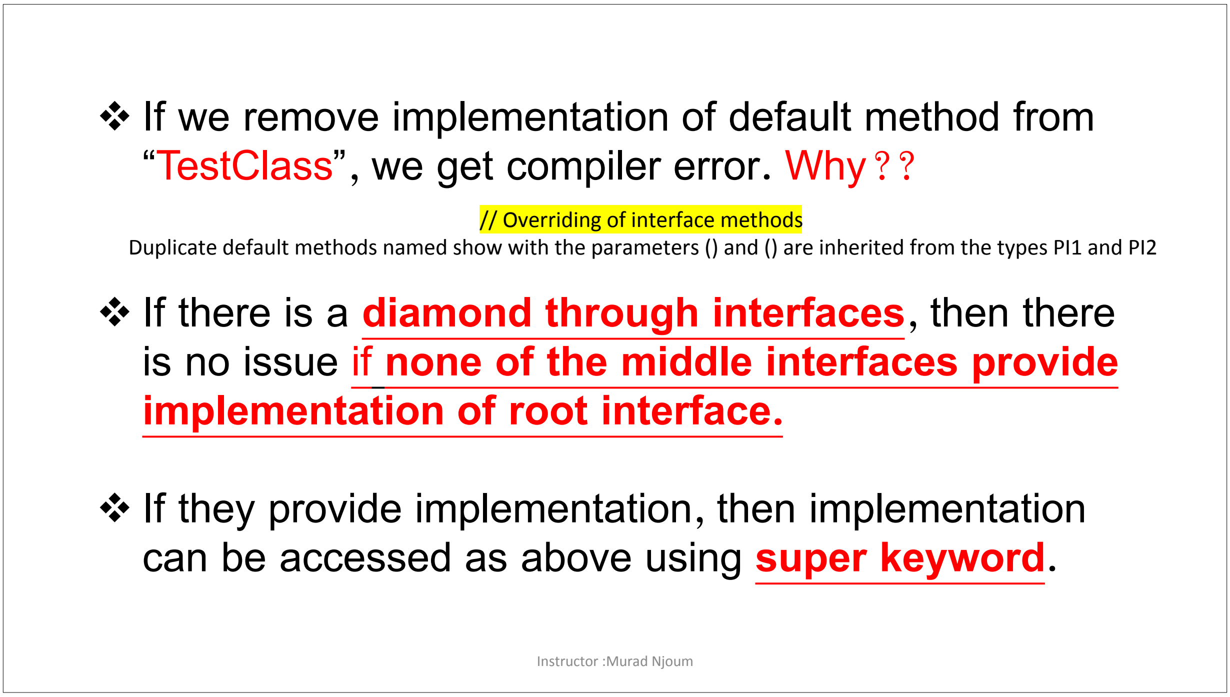
Class A
Class B
Class C

class C implements A, B { …. }

```
// Implementation class code
class TestClass implements PI1, PI2
{
        // Overriding default show method
        public void show()
        {
                // use super keyword to call the show
                // method of PI1 interface
                PI1.super.show();

                // use super keyword to call the show
                // method of PI2 interface
                PI2.super.show();
        }

        public static void main(String args[])
        {
                TestClass d = new TestClass();
                d.show();
        }
}
```

Output:

```
Default PI1
Default PI2
```



- If we remove implementation of default method from “TestClass”, we get compiler error. Why ? ?

// Overriding of interface methods

Duplicate default methods named show with the parameters () and () are inherited from the types PI1 and PI2

- If there is a **diamond through interfaces**, then there is no issue if **none of the middle interfaces provide implementation of root interface.**

- If they provide implementation, then implementation can be accessed as above using **super keyword**.

```
// A simple Java program to demonstrate how diamond
// problem is handled in case of default methods
interface GPI
{
	// default method
	default void show()
	{
		System.out.println("Default GPI");
	}
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

// Implementation class code
class TestClass implements PI1, PI2
{
	public static void main(String args[])
	{
		TestClass d = new TestClass();
		d.show();
	}
}
```

Definition and Usage

The implements keyword is used to implement an interface.
The interface keyword is used to declare a special type of class that **only contains** **abstract methods**.
To access the interface methods, the interface must be "**implemented**" (ike inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.
Notes on Interfaces:

It c**annot be used to create objects** (it is not possible to create an "Animal" object in the MyMainClass)

- Interface methods **does not have a body** - the body is provided by the "implement" class
- On implementation of an interface, **you must override all of its methods**
- Interface methods are by **default abstract and public**
- Interface attributes are by **default public, static and final**
- An interface **cannot contain a constructor** (as it cannot be used to create objects)

Why And When To Use Interfaces?
To achieve security - hide certain details and only show the important details of an object (interface).
Java does not support "**multiple inheritance"** (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note:

To implement multiple interfaces, separate them with a comma (see example below).

Interfaces

- An interface is a way to describe what classes should do, without specifying how they should do it.
- It is **not a class** but a set of requirements for classes that want to conform to the interface.

What is an **interface**? Why is an **interface** useful?

An interface is a class like construct that contains only constants and abstract methods.

In many ways, an interface is similar to an abstract class, but the intent نوايا of an interface is to specify common behavior for objects.

For example, you can specify that the objects are comparable, edible, cloneable using appropriate **interfaces**.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

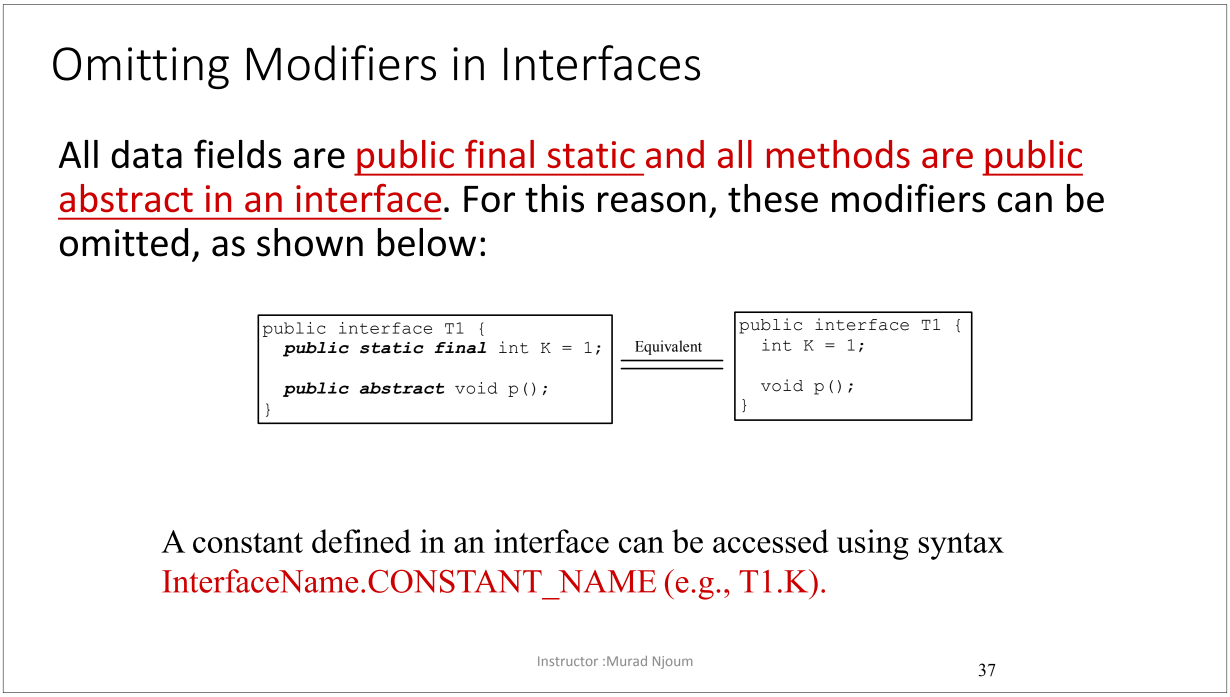
```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interface is a Special Class

- An interface is treated like a special class in Java.
- Each interface is compiled into a separate bytecode file, just like a regular class.
- Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class.
- For example, you can use an interface as a **data type** for a variable, as the result of casting, and so on.



Omitting Modifiers in Interfaces

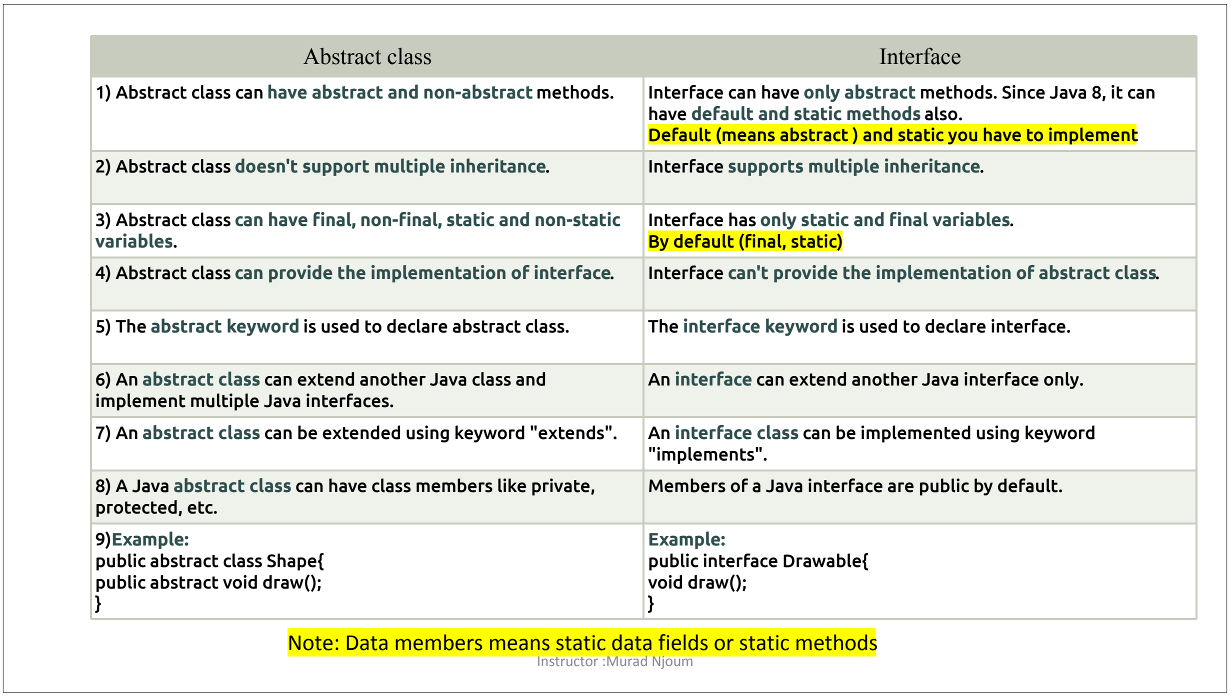
All data fields are public final static and all methods are public abstract in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).



Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also. Default (means abstract) and static you have to implement
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables. By default (final, static)
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface class can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Note: Data members means static data fields or static methods

```
public interface testInterface {
 int x=5;  //by default it's public static final
public static int methodX() {return 0;}
int X(); //by default it's abstracted method
}
```

```
public class testinter implements testInterface {

   public static void main(String[] args) {

System.out.print(testInterface.methodX());
  //output is zero
}

public int X() {
// just test override method of interface
return 0;
}
}
```

Example: The Comparable Interface

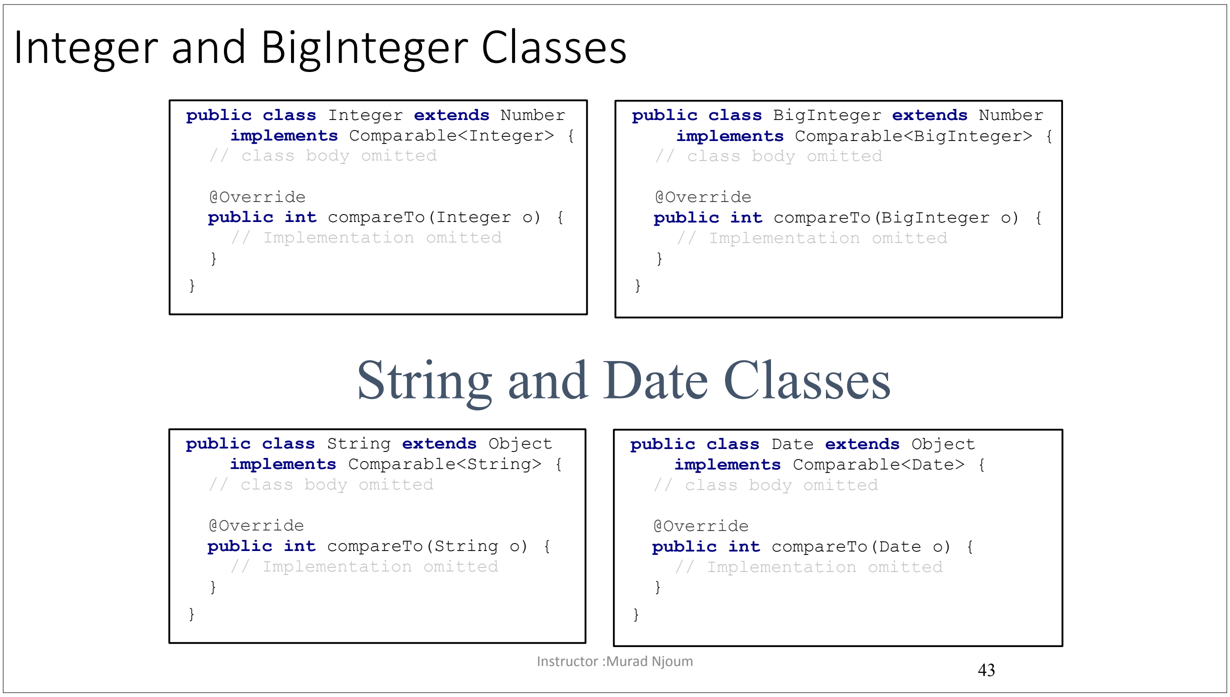
```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<T> {
  public int compareTo(T obj); // generic
}
```

Example

```
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

The toString, equals, and hashCode Methods

- ✓Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class.
- ✓Since all the numeric wrapper classes and the Character class **implement** the Comparable interface, the compareTo method is implemented in these classes.



Integer and BigInteger Classes

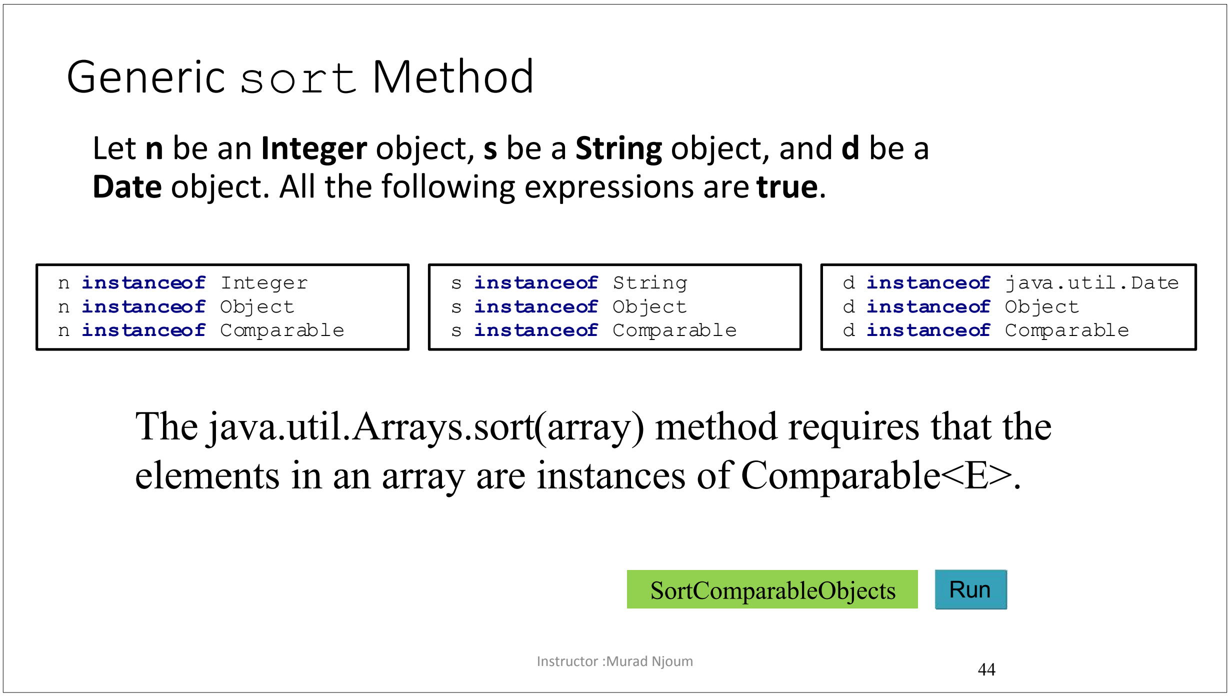
```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```



Generic `sort` Method

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**.

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

The java.util.Arrays.sort(array) method requires that the elements in an array are instances of Comparable<E>.

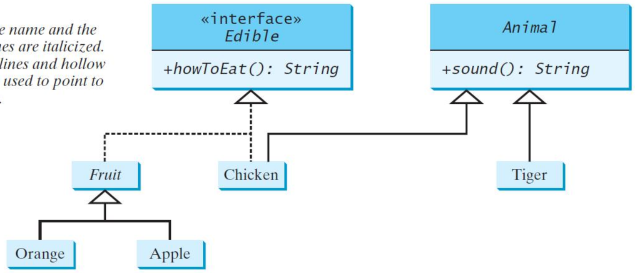
SortComparableObjects Run

Example

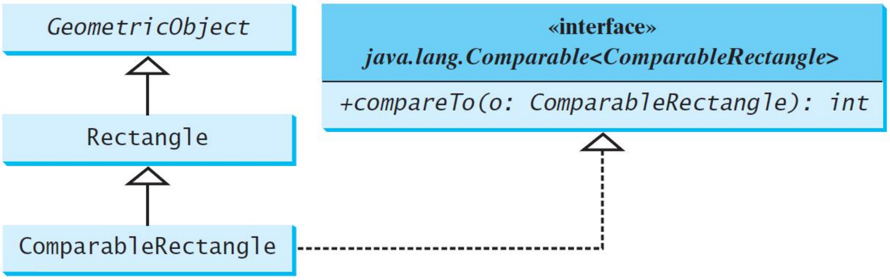
You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the **implements** keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).

Edible TestEdible Run

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



Defining Classes to Implement Comparable


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

ComparableRectangle SortRectangles Run

```
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return "Width: " + getWidth() + " Height: " + getHeight() +
      " Area: " + getArea();
  }
}
```

```
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

The **Cloneable** Interfaces

- Marker Interface: An empty interface.
- A marker interface **does not contain constants or methods**.
- It is used to denote that a **class possesses** certain desirable properties.
- A class that implements the **Cloneable interface** is marked cloneable, and its objects can be cloned using the **clone()** method defined in the **Object** class.

```
package java.lang;
public interface Cloneable {
}
```

Why Clonable Interface Needed ? Why we need clone()?

- Clone() returned type is object .
- Clone() is clasified under **Object class**.
- Clone() used with reference objects in memory
- Clone used with (Date, Calenders, Array, ArrayList, Any Classes created by a programmer) It doesn't use with primitive data types (float, int, double, char,...) and also we can't use it with immutable classes (objects content can't be changed after created)

You usually create copies (clones) of an object if you want to make changes in the state of the copy without changing the state of the original object.

Since the state of objects of immutable classes cannot be changed, you can use the original object without any risk of changing its state.

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();

System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));

System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

displays

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.

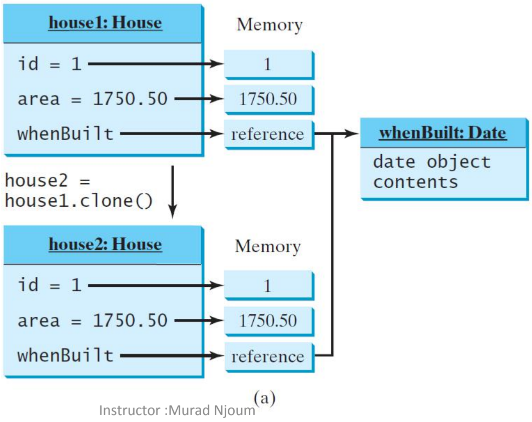
```
public class House implements Cloneable,
Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public double getArea() {
    return area;
  }

  public java.util.Date getWhenBuilt() {
    return whenBuilt;
  }

@Override /** Override the protected clone
method defined in the Object class, and
strengthen its accessibility */
  public Object clone() {
    try {
      return super.clone();
    }
    catch (CloneNotSupportedException ex) {
      return null;
    }
  }

  @Override // Implement the compareTo
method defined in Comparable
  public int compareTo(House o) {
    if (area > o.area)
      return 1;
    else if (area < o.area)
      return -1;
    else
      return 0;
  }
}
```

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();

Shallow Copy

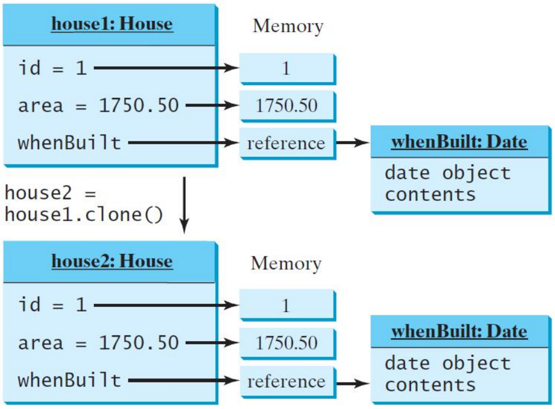


Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();

Deep Copy



house1: House

Memory

id = 1

area = 1750.50

whenBuilt

1

1750.50

reference

whenBuilt: Date

date object contents

house2 = house1.clone()

house2: House

Memory

id = 1

area = 1750.50

whenBuilt

1

1750.50

reference

whenBuilt: Date

date object contents

(b)

The default version of clone() method creates the shallow copy of an object.

The shallow copy of an object will have exact copy of all the fields of original object

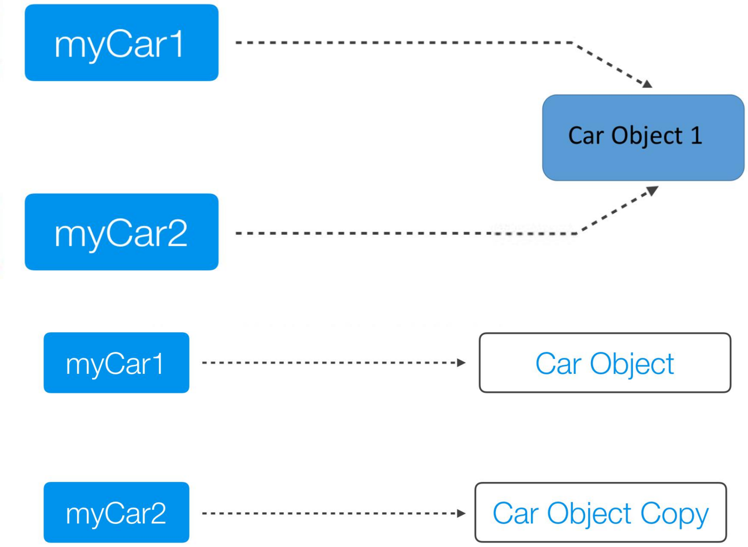
If original object has any references to other objects as fields, then only references of those objects are copied into clone object, copy of those objects are not created.

That means any changes made to those objects through clone object will be reflected in original object or vice-versa. Shallow copy is not 100% disjoint from original object. Shallow copy is not 100% independent of original object.

Deep copy of an object will have exact copy of all the fields of original object just like shallow copy.

But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone() method on them. That means clone object and original object will be 100% disjoint.

They will be 100% independent of each other. Any changes made to clone object will not be reflected in original object or vice-versa.



myCar1

Car Object 1

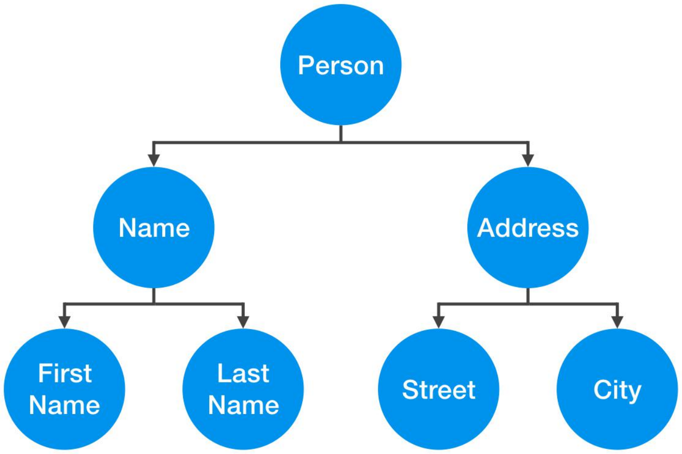
myCar2

myCar1

Car Object

myCar2

Car Object Copy



Person

Name

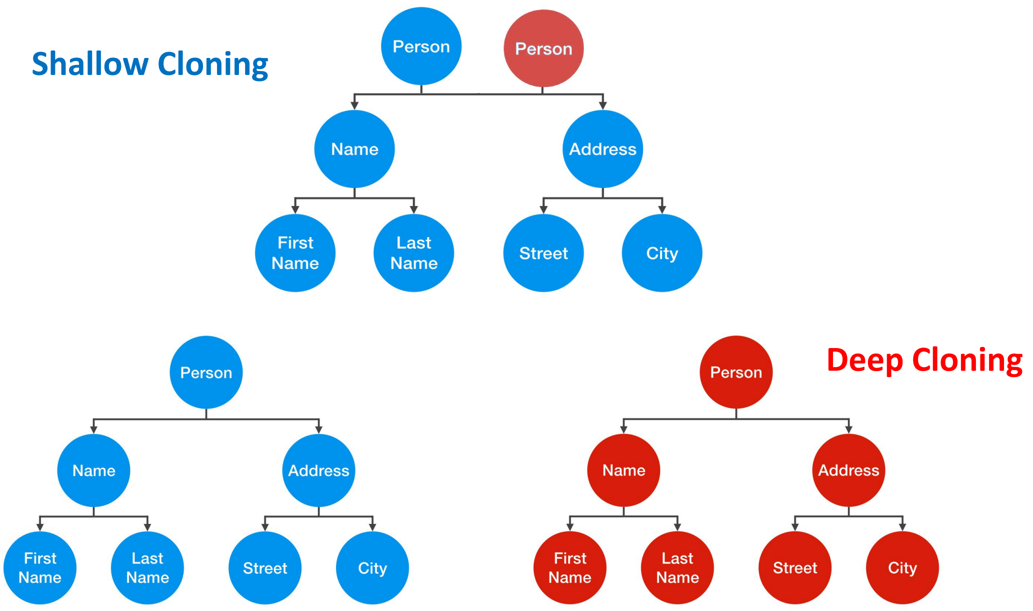
Address

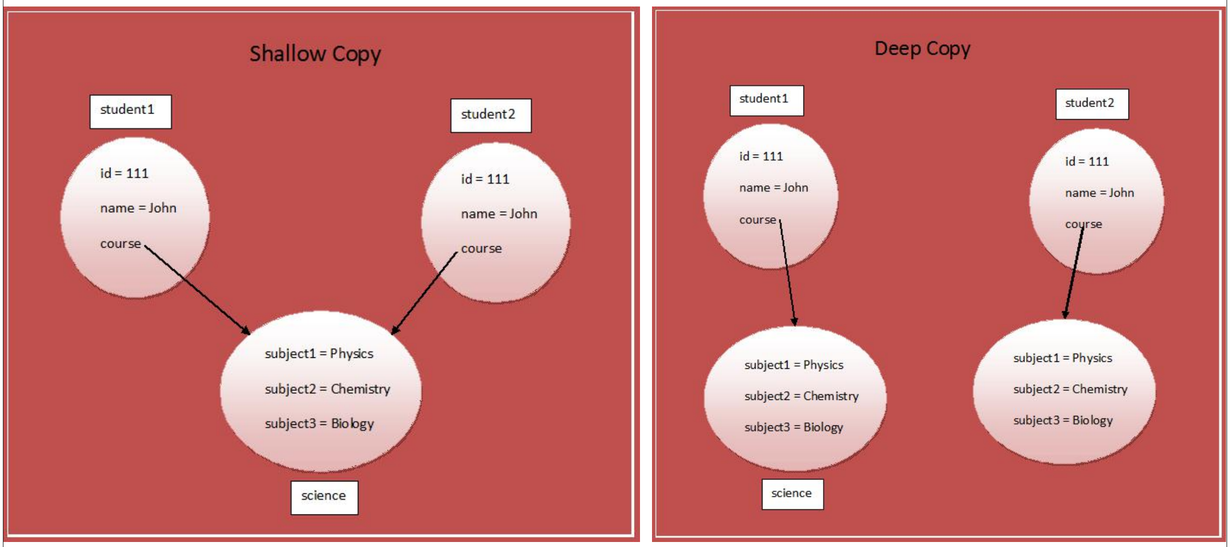
First Name

Last Name

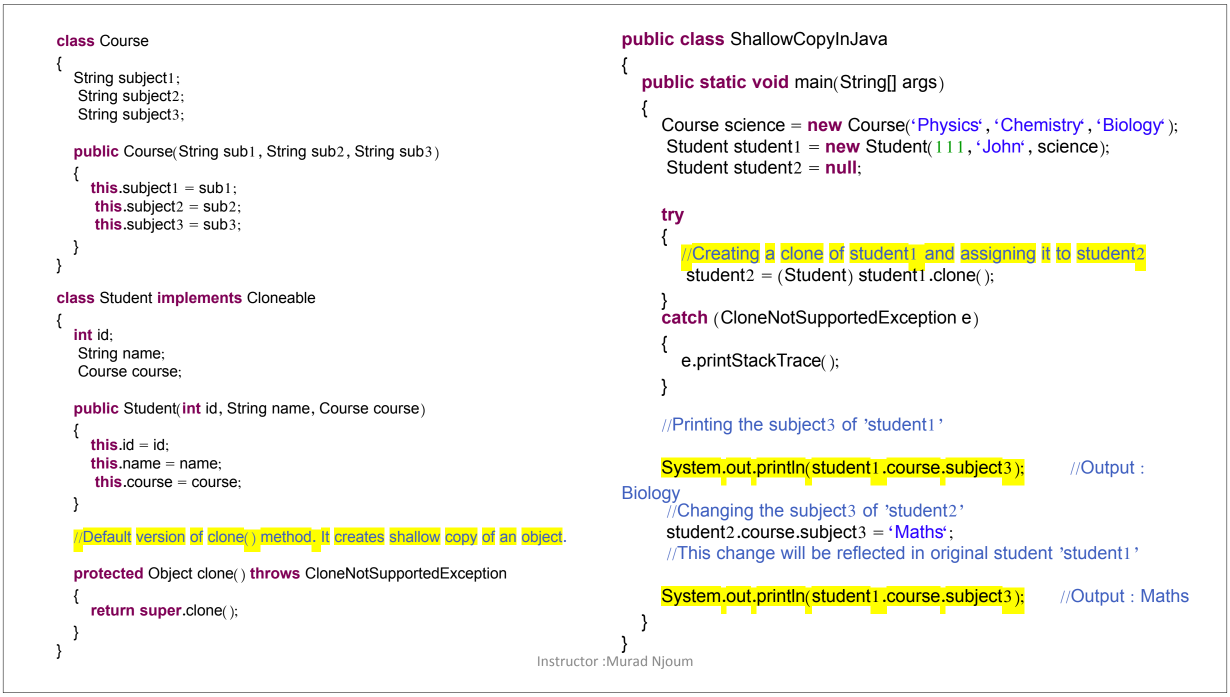
Street

City





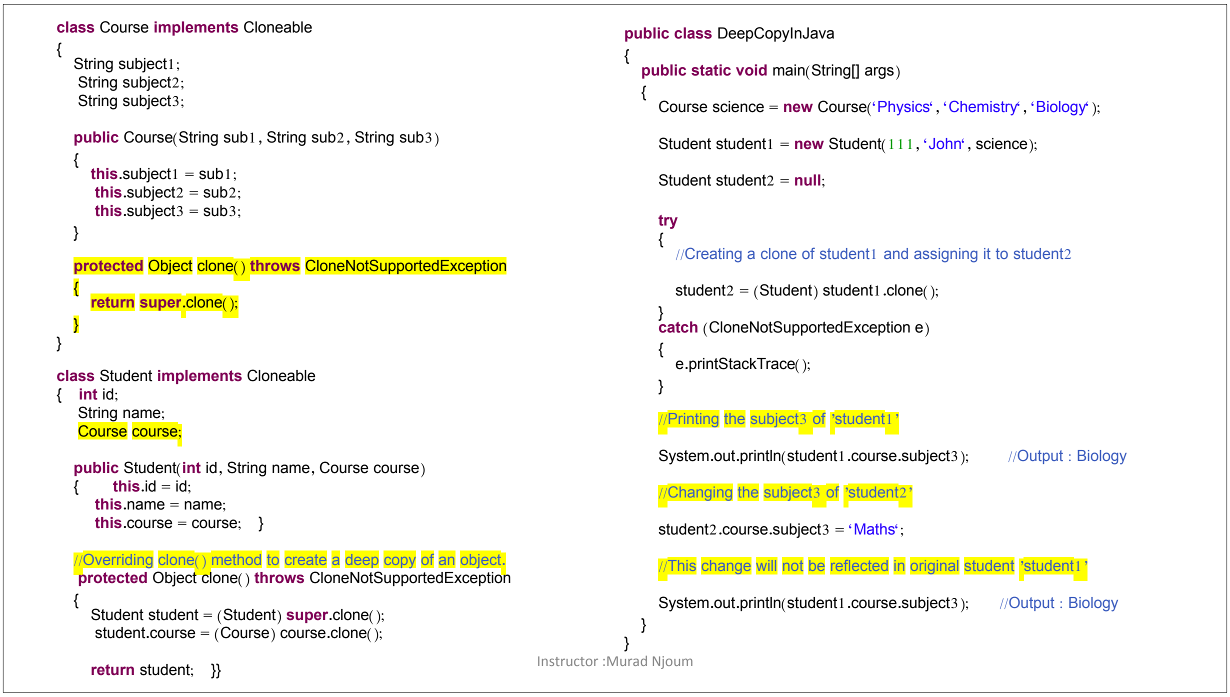
Instructor :Murad Njoum



```
class Course
{
    String subject1;
     String subject2;
     String subject3;

    public Course(String sub1, String sub2, String sub3)
    {
        this.subject1 = sub1;
         this.subject2 = sub2;
         this.subject3 = sub3;
    }
}

class Student implements Cloneable
{
    int id;
     String name;
     Course course;

    public Student(int id, String name, Course course)
    {
        this.id = id;
        this.name = name;
         this.course = course;
    }

    //Default version of clone() method. It creates shallow copy of an object.

    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

```
public class ShallowCopyInJava
{
   public static void main(String[] args)
   {
      Course science = new Course(‘Physics‘, ‘Chemistry‘, ‘Biology‘);
       Student student1 = new Student(111, ‘John‘, science);
       Student student2 = null;

      try
      {
         //Creating a clone of student1 and assigning it to student2
          student2 = (Student) student1.clone();
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }

      //Printing the subject3 of 'student1'

      System.out.println(student1.course.subject3);        //Output :
Biology
       //Changing the subject3 of 'student2'
       student2.course.subject3 = ‘Maths‘;
       //This change will be reflected in original student 'student1'

      System.out.println(student1.course.subject3);       //Output : Maths
   }
}
```

Instructor :Murad Njoum



```
class Course implements Cloneable
{
    String subject1;
     String subject2;
     String subject3;

    public Course(String sub1, String sub2, String sub3)
    {
        this.subject1 = sub1;
         this.subject2 = sub2;
         this.subject3 = sub3;
    }

    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

class Student implements Cloneable
{   int id;
     String name;
     Course course;

    public Student(int id, String name, Course course)
    {       this.id = id;
         this.name = name;
         this.course = course;    }

    //Overriding clone() method to create a deep copy of an object.
     protected Object clone() throws CloneNotSupportedException
    {
        Student student = (Student) super.clone();
         student.course = (Course) course.clone();

        return student;    }}
```

```
public class DeepCopyInJava
{
   public static void main(String[] args)
   {
      Course science = new Course(‘Physics‘, ‘Chemistry‘, ‘Biology‘);

      Student student1 = new Student(111, ‘John‘, science);

      Student student2 = null;

      try
      {
         //Creating a clone of student1 and assigning it to student2

         student2 = (Student) student1.clone();
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }

      //Printing the subject3 of 'student1'

      System.out.println(student1.course.subject3);        //Output : Biology

      //Changing the subject3 of 'student2'

      student2.course.subject3 = ‘Maths‘;

      //This change will not be reflected in original student 'student1'

      System.out.println(student1.course.subject3);       //Output : Biology
   }
}
```

Instructor :Murad Njoum

Interfaces vs. Abstract Classes

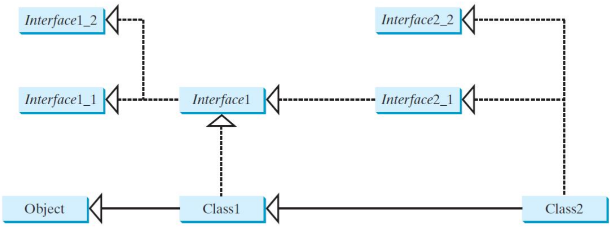
In an interface, the **data must be constants**; an abstract class can **have all types of data.**

Each method in an **interface** has only a signature without **implementation**; an abstract class can **have concrete methods.**

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

- **All classes share a single root**, the Object class, but there is **no single root** for interfaces. **Like a class, an interface also defines a type**.
- A variable of an interface type can reference any instance of the class that implements the interface.
- If a class extends an interface, this interface plays the same role as a superclass. **You can use an interface as a data type and cast a variable** of an interface type to its subclass, and vice versa.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Instructor :Murad Njoum