Java OOP for Android Platform

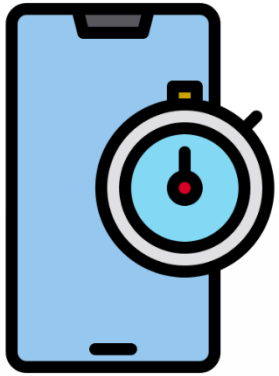
Dmytro Zubov, PhD
dmytro.zubov@ucentralasia.org

Naryn, Kyrgyzstan, October 23, 2022

Special thanks to Meerbek Akimzhanov for sharing his pics

Naryn, Kyrgyzstan, 9:25 am, November 5, 2021

Lessons learnt last time



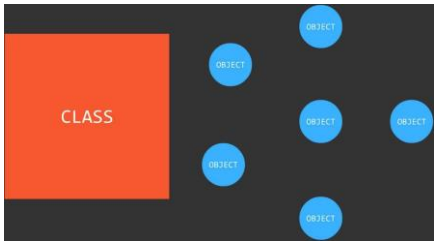
- Course project proposal

Note: A kind reminder to show the course project's progress till this Friday. Thank you.

What we gonna discuss today?

- An Example of the Java Class
- Java Inheritance
- Java Interfaces
- Java Polymorphism
- Java Abstract Class
- Java Encapsulation
- final Keyword
- static Class Members
- Java OOP Constructor

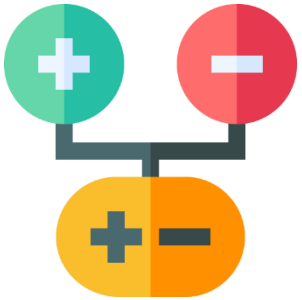
An Example of the Java Class




```
class clsName extends ParentClass
{
  // instance variable declaration
  type1 varName1 = value1;
  type2 varName2 = value2;
  :
  :
  typeN varNameN = valueN;

  //  Constructors
  clsName(cparam1)
  {
    // body of constructor
  }
  :
  :
  clsName(cparamN)
  {
    // body of constructor
  }

  // Methods
  rType1 methodName1(mParams1)
  {
    // body of method
  }
  :
  :
  rTypeN methodNameN(mParamsN)
  {
    // body of method
  }
}
```

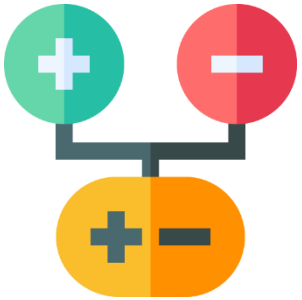
- ***Variables*** represent class/object state. Class can have static and instance variables. ***Methods*** provide the logic that constitutes the behavior defined by a class. Class can have static and instance methods. ***Constructors*** initialize the state of a new instance of a class.
- ***class*** indicates that a class named ***clsName*** is being declared. It must follow the Java naming conventions for identifiers.
- ***Instance variables*** named varName1 through varNameN have normal variable declaration syntax. Each variable must be assigned a type shown as type1 through typeN and may be initialized to a value shown as valueN.
- ***Constructors*** always have the same name as the class. They do not have return values. cparam1 through cparamN are optional parameter lists.
- ***Methods*** named mthName1 through mthNameN can be included. The return type of the methods are rtype1 through rTypeN and mParamN are an optional parameter lists.

Java Inheritance



- **Inheritance** can be defined as the process where one object acquires the properties of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- When we talk about inheritance, the most used keywords are **extends** (IS-A type, e.g., bus is a vehicle) and **implements** (HAS-A relationship, e.g., bus has an engine). By using these keywords, we can make one object acquire the properties of another object.
- Java only supports only single inheritance. However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

Java Inheritance. IS-A Relationship



- "IS-A" says "This object is a type of that object", e.g.:

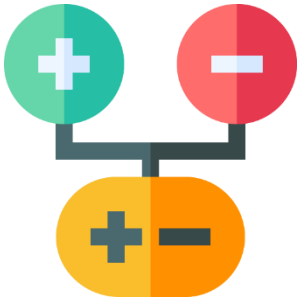
```
public class Computer{ }
public class Small_Computer_Device extends Computer{ }
public class Quantum_Computer extends Computer{ }
public class Laptop extends Small_Computer_Device{ }
```

- Computer is the superclass of Small_Computer_Device class
- Computer is the superclass of Quantum_Computer class
- Quantum_Computer and Small_Computer_Device are subclasses of Computer class
- Laptop is the subclass of both Small_Computer_Device and Computer classes

- If we consider the IS-A relationship, we can say:

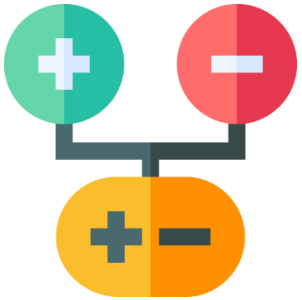
- Small_Computer_Device IS-A Computer
- Quantum_Computer IS-A Computer
- Laptop IS-A Small_Computer
- Hence : Laptop IS-A Computer as well.

Java Inheritance. IS-A Relationship (cont.)

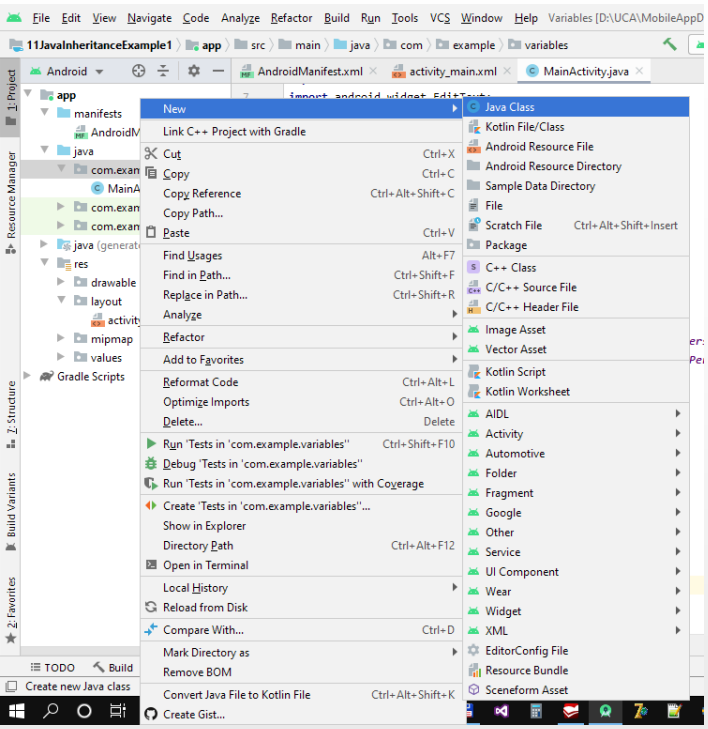


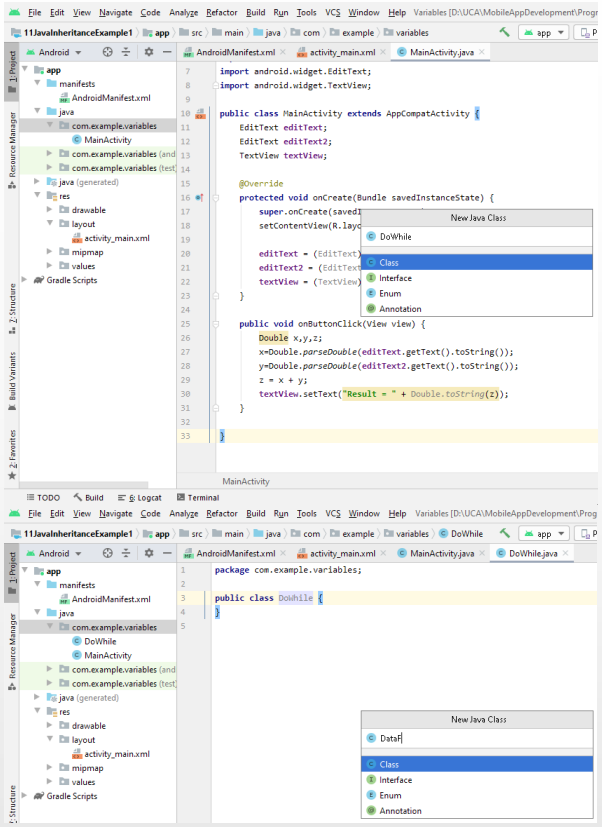
- With use of the **extends** keyword the subclasses will be able to inherit all the properties of the superclass except private properties of the superclass

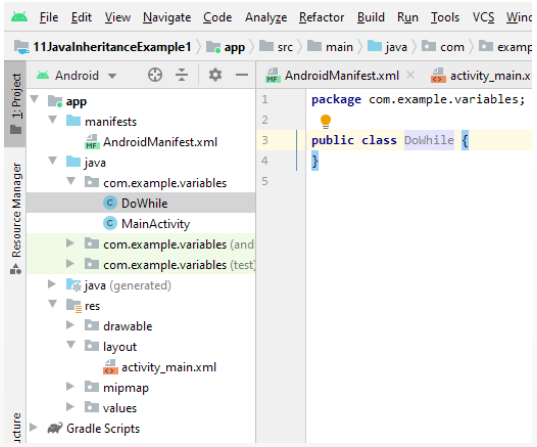
Java Inheritance: An Example

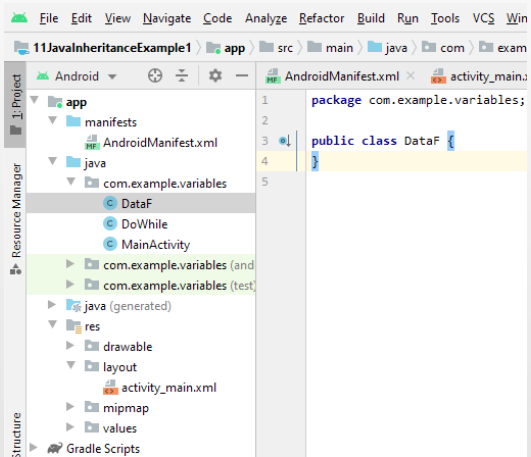


- Here, we create two new class files `DoWhile` and `DataF` in the branch with `MainActivity.java` (or `FullscreenActivity.java`) file:

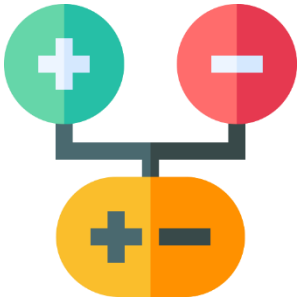




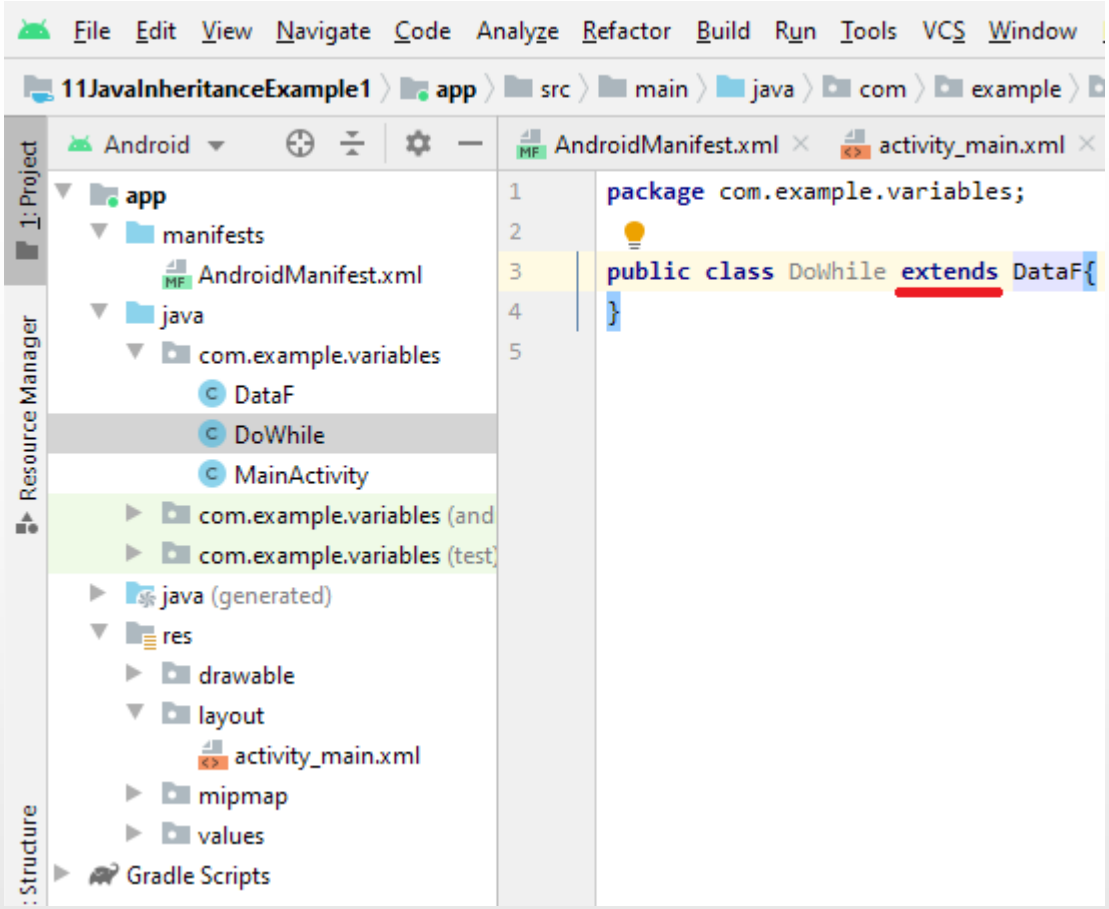




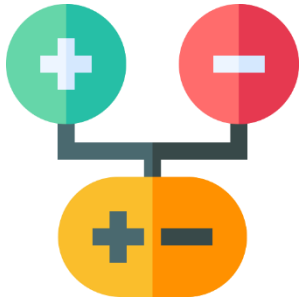
Java Inheritance: An Example (cont.)



- `DataF` is a superclass:



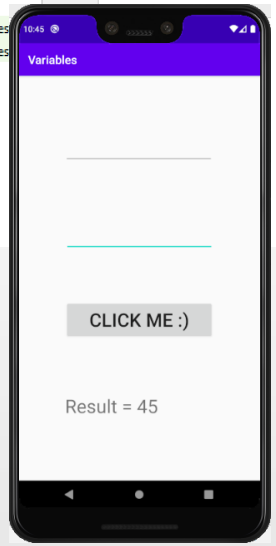
Java Inheritance: An Example (cont.)



- Then, we write the codes in the `MainActivity.java` and in the class files `DoWhile` and `DataF` as follows:

```
package com.example.variables;

public class DataF {
    public int i, x; //
}
```

```
package com.example.variables;

public class DoWhile extends DataF{
    public int Calc()
    {
        i=0; x=0;
        do
        {
            x = x + i; i++;
        }
        while (i<10);
        return x;
    }
}
```




```
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    public void onButtonClick(View view) {
        DoWhile d1 = new DoWhile();
        int x = d1.Calc();
        textView.setText("Result = " + x);
    }

}
```

Java Inheritance: HAS-A relationship

- **HAS-A relationship** determines whether a certain class **HAS-A** certain thing. It helps to reduce duplication of code as well as bugs.

DoWhileHasA

```
package com.example.zubov.myappl;

/**
 * Created by Zubov on 11/11/2017.
 */

public class DoWhileHasA {
    public DoWhile dw = new DoWhile();
}
```

activity_fullscreen.xml × | strings.xml × | FullscreenActivit | hileHasA.java ×

InheritanceHAS-Arelationsl
- manifests
- java
  - com.example.zubov.r
    - DataF
    - DoWhile
    - DoWhileHasA
    - FullscreenActi
  - com.example.zubov.r
  - com.example.zubov.r
- res
- Gradle Scripts

FullscreenActivity | onButtonClick()

```
122
123     public void onButtonClick(View view) {
124         DoWhileHasA dl = new DoWhileHasA();
125         textView.setText("Result = " + Integer.toString(dl.dw.Calc()));
126     }
127
128     @Override
129     protected void onPostCreate(Bundle savedInstanceState) {
130         super.onPostCreate(savedInstanceState);
131
132         // Trigger the initial hide() shortly after the activity has be
133         // created, to briefly hint to the user that UI controls
134         // are available.
```

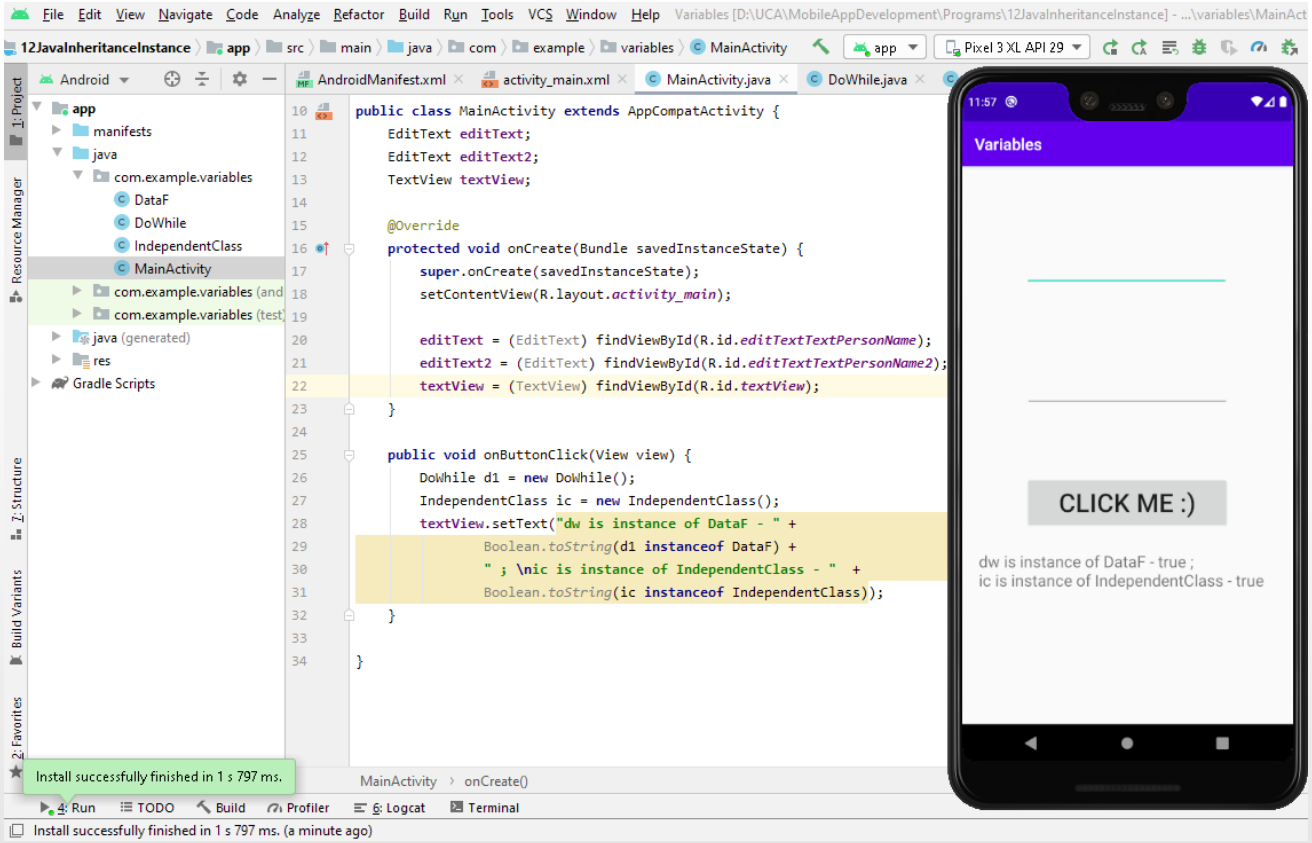
QVGA_API_24:55...

Result = 45

Java Inheritance: instanceof Operator

- `instanceof` operator allows to identify that some class is a subclass of the appropriate superclass:

```
public class IndependentClass {
}
```



Java Inheritance: instanceof Operator (cont.)

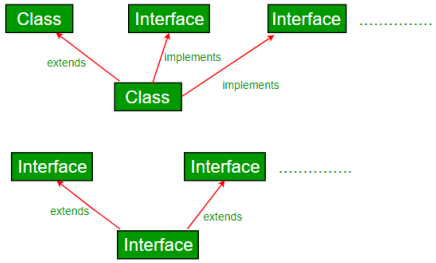
- What will we have if we modify code above? In other words, we are trying to identify if "ic is instanceof DataF"...

```
public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    public void onButtonClick(View view) {
        DoWhile d1 = new DoWhile();
        IndependentClass ic = new IndependentClass();
        textView.setText("dw is instance of DataF - " +
                Boolean.toString(d1 instanceof DataF) +
                " ; \nic is instance of IndependentClass - "  +
                Boolean.toString(ic instanceof IndependentClass)
                + " ; \nic is instance of DataF - "  +
                Boolean.toString(ic instanceof DataF)
                );
    }

}
```

Inconvertible types; cannot cast 'com.example.variables.IndependentClass' to 'com.example.variables.DataF'

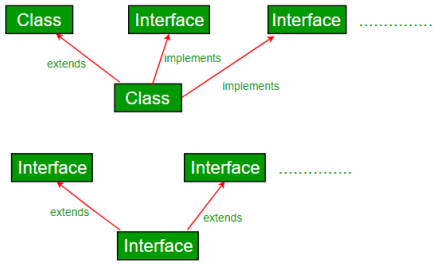
Install successfully finished in 1 s 797 ms. (4 minutes ago)

Java Interfaces



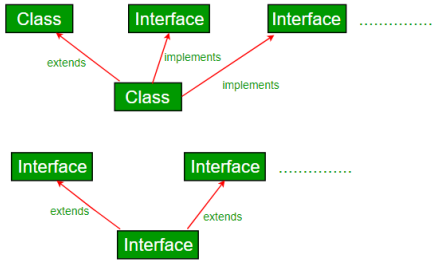
- An **<u>interface</u>** is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
  - **An interface is not a class**. Writing an interface is like writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. **An interface contains behaviors that the class implements.**
  - Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Java Interfaces (cont.)



- An interface is different from a class in several ways:
  - we cannot instantiate an interface
  - an interface does not contain any constructors
  - all methods in the interface are abstract
  - an interface cannot contain instance fields – the only fields that can appear in an interface must be declared both static and final
  - an interface is not extended by a class, it is implemented by a class
  - an interface can extend multiple interfaces
- The **interface** keyword is used to declare an interface

Java Interfaces: implements Operator



- The **implements** keyword is used by classes to inherit from interfaces. *Interfaces can never be extended by the classes*.

```
public interface Computer{ }
public interface PowerGrid { }
public interface LoopDoWhile extends Computer, PowerGrid{
    public int Calc();
}
public class DataF {
    public int i, x;
}
public class DoWhile extends DataF implements LoopDoWhile{
    public int Calc()
    {
…
        return x;
    }
}
```

Java Interfaces: implements Operator (cont.)

Android | activity_fullscreen.xml | strings.xml | FullscreenActivity.java | PowerGrid.java | LoopDoWhile.java

- Interface
  - manifests
  - java
    - com.example.zubov.mya
      - Computer
      - DataF
      - DoWhile
      - FullscreenActivity
      - LoopDoWhile
      - PowerGrid
    - com.example.zubov.mya
    - com.example.z
  - res
- Gradle Scripts

LoopDoWhile

```java
package com.example.zubov.myapp1;

/*
 * Created by Zubov on 11/12/2017.
 */

public interface LoopDoWhile extends Computer, PowerGrid {
    public int Calc();
}
```

Android Emulator - 3.3_WQVGA_API_24:55...

Result = 45

PowerGrid

```java
package com.example.zubov.myapp1;

/**
 * Created by Zubov on 11/12/2017.
 */

public interface PowerGrid {
}
```

Computer

```java
package com.example.zubov.myapp1;

/**
 * Created by Zubov on 11/12/2017.
 */

public interface Computer {
}
```

```java
package com.example.zubov.myapp1;

/**
 * Created by Zubov on 11/12/2017.
 */

public class DoWhile extends DataF implements LoopDoWhile {
    public int Calc()
    {
        i=0;
        x=0;
        //Sum of integers from 0 to 9
        //If i>9, we have incorrect resul
        do
        {
            x = x + i;
            i++;
        }
        while (i<10);
        return x;
    }
}
```

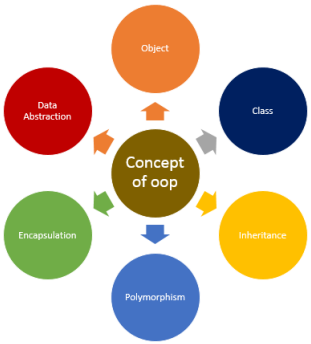
New Java Class

- Name
- Class
- Interface
- Enum
- Annotation

DataF

```java
package com.example.zubov.myapp1;

/**
 * Created by Zubov on 11/12/2017.
 */

public class DataF {
    public int i, x;  // This line with public access is equal to int i, x;
}
```

Java Polymorphism

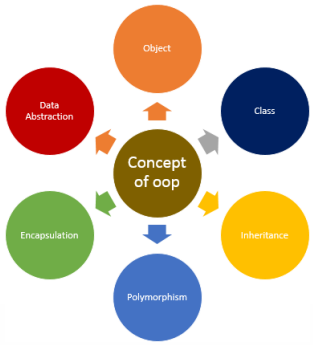


- **Polymorphism** is the ability of an object to take on many forms

- Polymorphism in Java has two types: **compile time polymorphism (static binding)** and **runtime polymorphism (dynamic binding)**. Method overloading is an example of the static polymorphism, while method overriding is an example of the dynamic polymorphism.

http://www.sitepoint.com/quick-guide-to-polymorphism-in-java/

Java Polymorphism: Static Binding Based on Overloading



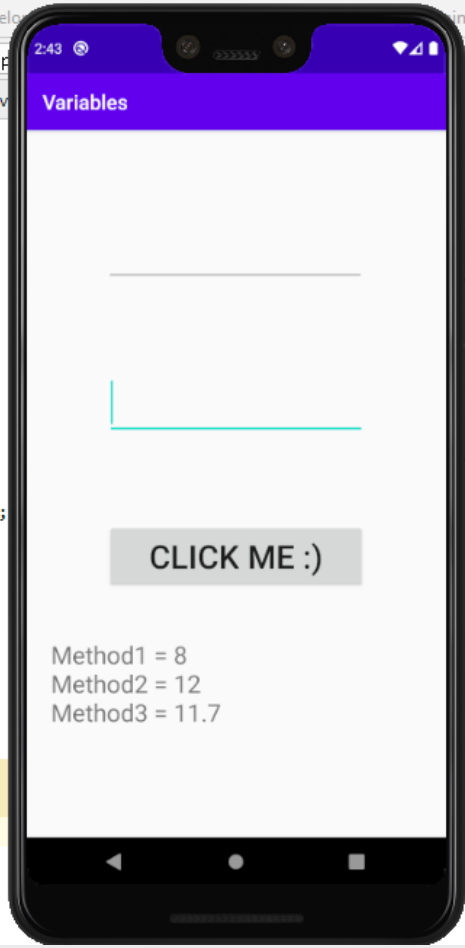
- Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters

- At compile time, Java knows which method to invoke by checking the method signatures. So, this is called **compile time polymorphism** or **static binding**.

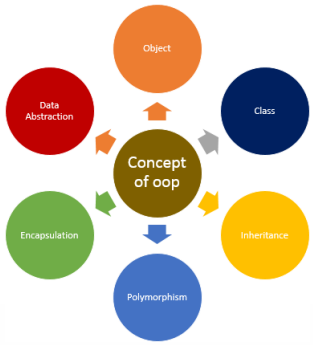
Java Polymorphism: An Example of Static Binding Based on Overloading

```java
package com.example.variables;

public class JavaOverloading {
    public int add(int x, int y)          //method 1
    {
        return x+y+1;
    }
    public int add(int x, int y, int z)   //method 2
    {
        return x+y+z;
    }
    public double add( double x, int y)   //method 3
    {
        return x+y+2.0;
    }
}
```

```java
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    public void onButtonClick(View view) {
        JavaOverloading d1=new JavaOverloading();
        int x = d1.add( x: 3, y: 4);          //method 1 called
        int y = d1.add( x: 3, y: 4, z: 5);    //method 2 called
        double z = d1.add( x: 6.7, y: 3);          //method 3 called
        textView.setText("Method1 = " + Integer.toString(x) + "\n"
                + "Method2 = " + Integer.toString(y) + "\n"
                + "Method3 = " + Double.toString(z));
    }
```



Java Polymorphism: Dynamic Polymorphism Based on Overriding



- In Java, static polymorphism may be achieved through method overriding as well. Let's say, in the program we create an object of the subclass and assign it to the superclass reference. Now, if we call the overridden method on the superclass reference then the subclass version of the method will be called.

Java Polymorphism: An Example of Dynamic Polymorphism Based on Overriding

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window

15PolymorphismOverriding 〉 app 〉 src 〉 main 〉 java 〉 com 〉 example

AndroidManifest.xml | Computer.java | Laptop.java | activity_main.x

```java
package com.example.variables;

public class Computer {
    String s1;
    public String info() {
        s1 = "Computer is a general purpose device that can...";
        return s1;
    }
}
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Windo

15PolymorphismOverriding 〉 app 〉 src 〉 main 〉 java 〉 com 〉 example

AndroidManifest.xml | Computer.java | Laptop.java | activity_main.

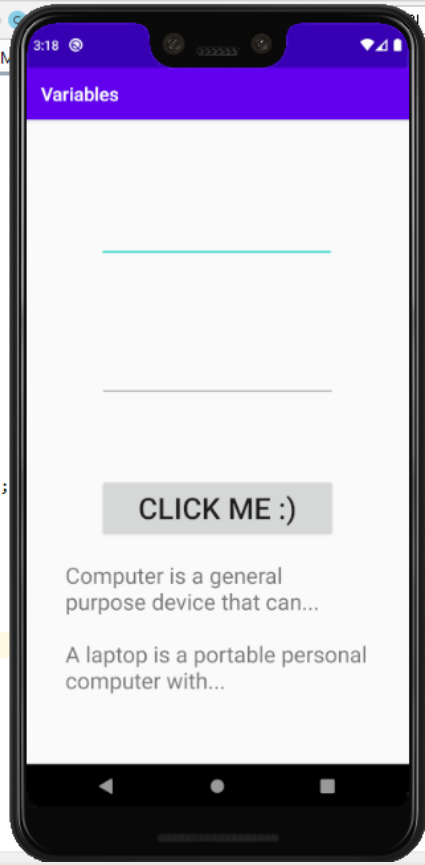
```java
package com.example.variables;

public class Laptop extends Computer{
    public String info() {
        s1 = "A laptop is a portable personal computer with...";
        return s1;
    }
}
```

lit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help Variables [D:\UCA\MobileAppDevelopment\Programs\15Polyr

hismOverriding 〉 app 〉 src 〉 main 〉 java 〉 com 〉 example 〉 variables

oidManifest.xml | Computer.java | Laptop.java | activity_main.xml

```java
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    public void onButtonClick(View view) {
        Computer c1 = new Computer();
        Laptop l1 = new Laptop();
        textView.setText(c1.info() + "\n\n" + l1.info());
    }

}
```



Java Polymorphism: Overriding. Using super Keyword

- The **super** keyword is used when we would like to invoke a superclass version of an overridden method.

activity_fullscreen.xml × | strings.xml × | FullscreenActivity.java × | Laptop.java × | Computer.java ×

Laptop | info()

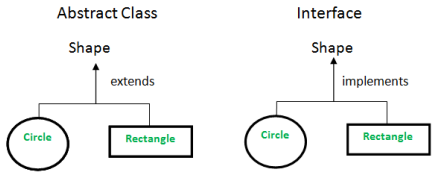
```java
package com.example.zubov.myapp1;

/**
 * Created by Zubov on 11/12/2017.
 */

public class Laptop extends Computer {
    public String info() {
        s1 = "A laptop is a portable personal computer with..." + "\r\n" + super.info();
        return s1;
    }
}
```

Android Emulator - 3.3_WQVGA_API_24:55...

A laptop is a portable personal computer with...
Computer is a general purpose device that can...

```java
public void onButtonClick(View view) {
    Laptop ll = new Laptop();
    textView.setText(ll.info());
}
```

Java Abstract Class



- **Abstraction** refers to the ability to make a class abstract in OOP
- **An abstract class is one that cannot be instantiated**. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. However, it is not possible to create an instance of the abstract class.
- **abstract** keyword is used for declaration of a class abstract. The keyword appears in the class declaration somewhere before the class keyword.

Java Abstract Class: An Example

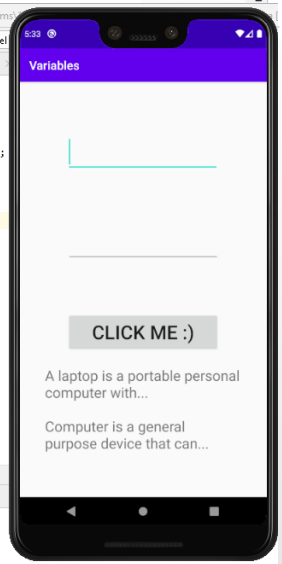
```java
package com.example.variables;

public abstract class Computer {
    String s1;
    public String info() {
        s1 = "Computer is a general purpose device that can...";
        return s1;
    }
}
```

```java
package com.example.variables;

public abstract class Laptop extends Computer{
    public String info() {
        s1 = "A laptop is a portable personal computer with..." + "\n\n" + super.info();
        return s1;
    }
}
```

WRONG

```java
package com.example.variables;

public class Laptop extends Computer{
    public String info() {
        s1 = "A laptop is a portable personal computer with..." + "\n\n" + super.info();
        return s1;
    }
}
```

CORRECT




```java
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    public void onButtonClick(View view) {
        Laptop l1 = new Laptop();
        textView.setText
    }
```

'Laptop' is abstract; cannot be instantiated

Make 'Laptop' not abstract More actions...

Build: Sync Build Output

:app:compileDebugJavaWithJavac 1 error 211 ms

app/src/main/java/com/example/variables/MainA

Laptop is abstract; cannot be instantiated

Compilation failed; see the compiler error output for d

D:\UCA\MobileAppDevelopment\Programs\17OOPAbstractClass\app\src\main\

Laptop l1 = new Laptop();

^

Build: finished at 11/9/2020 5:32 PM 1 s 582 ms

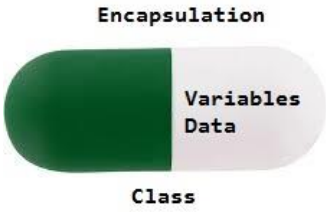
BUILD SUCCESSFUL in 1s

21 actionable tasks: 4 executed, 17 up-to-date

Build Analyzer results available

Install successfully finished in 1 s 279 ms.

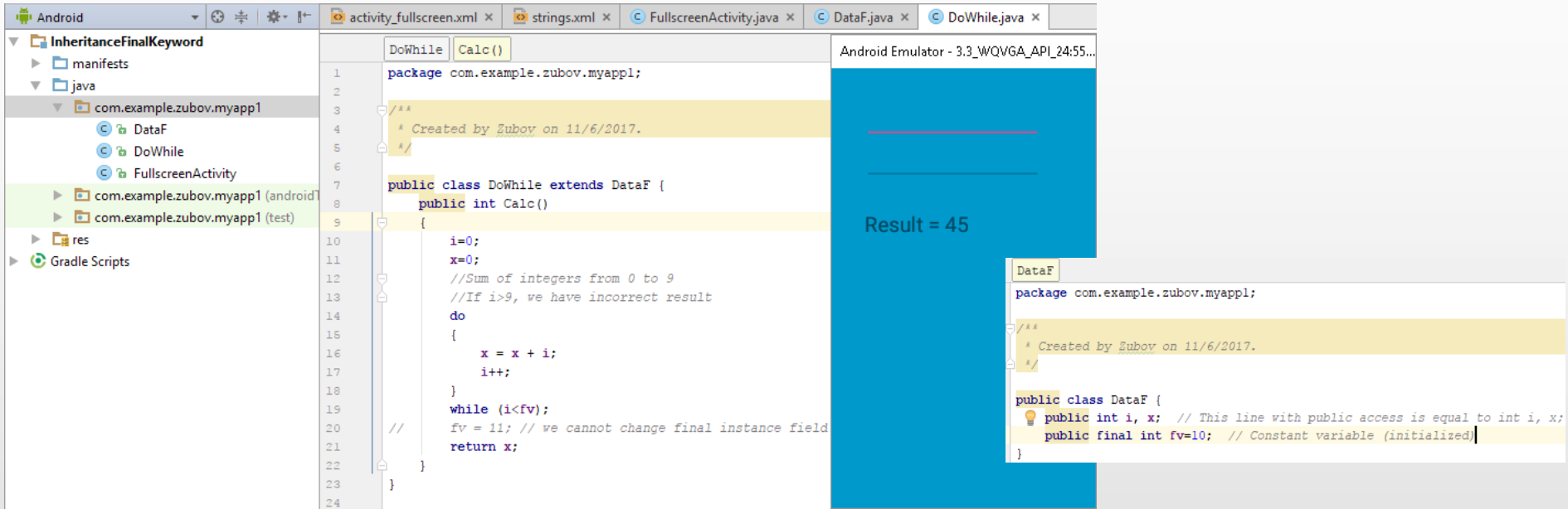
Java Encapsulation



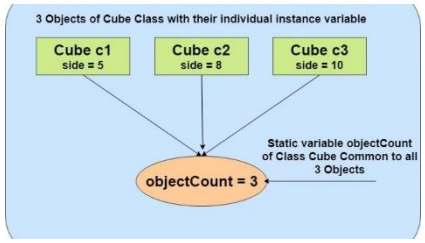
- **Encapsulation** is the technique of making the fields in a class ***private*** and providing access to the fields via ***public*** methods.
- If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, **<u>encapsulation is also referred to as data hiding</u>**.

final keyword. final Instance Variables

- Keyword **final** specifies that a variable is not modifiable (i.e., it is a constant)
- **final** instance variables can be initialized at their declaration. If they are not initialized in their declarations, they must be initialized in a constructor.



static Class Members



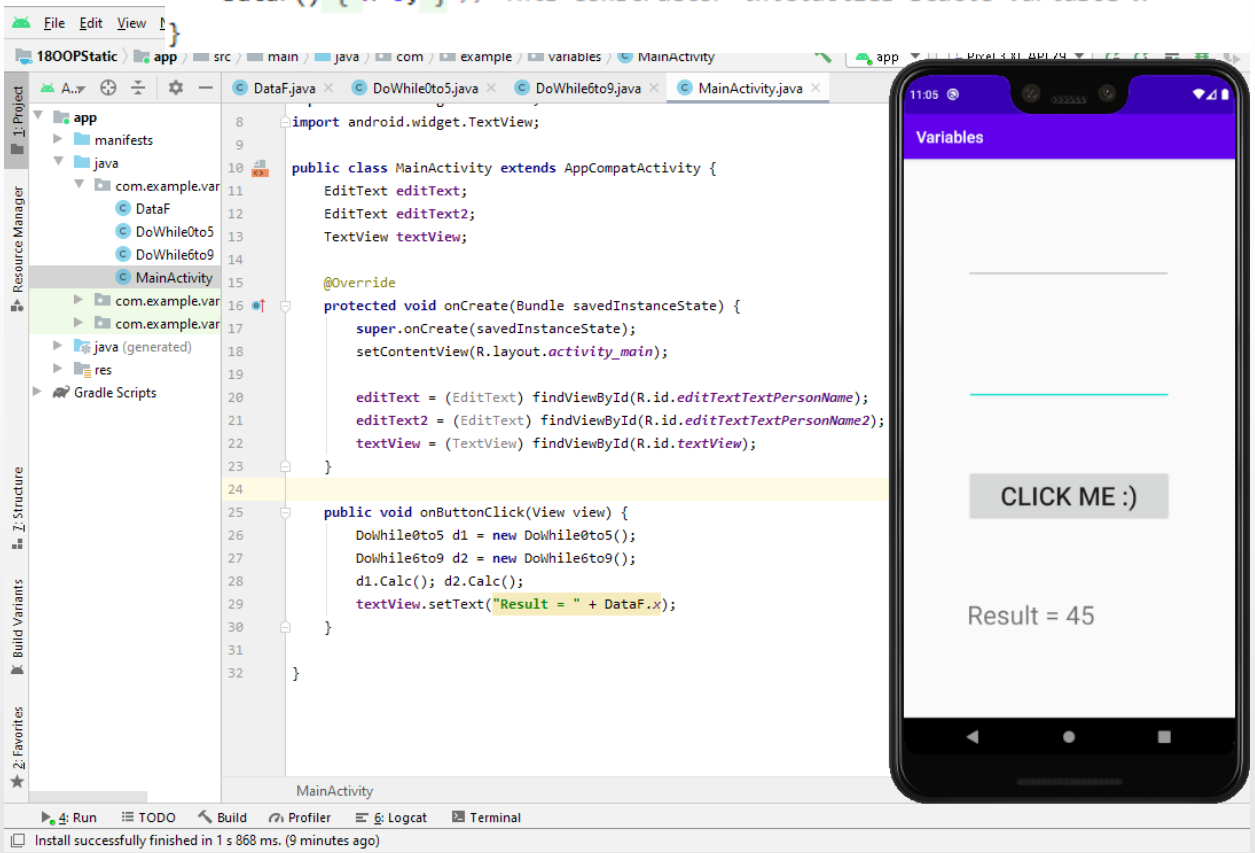
- **static** fields are also known as the class variables. They are used when:
  - all objects of the class should share the same copy of this instance variable
  - this instance variable should be accessible even when no objects of the class exist
- **static** fields can be accessed with the class name or an object name and a dot (.)
- Must be initialized in their declarations, or else the compiler will initialize it with a default value (e.g., 0 for ints)

static Class Members: An Example

```java
public class DoWhile6to9 extends DataF {
    public void Calc()
    {
        i=6;
        do
        {
            x = x + i; i++;
        }
        while (i<10);
    }
}
```

```java
public class DoWhile0to5 extends DataF{
    public void Calc()
    {
        i=1;
        do
        {
            x = x + i; i++;
        }
        while (i<6);
    }
}
```

```java
package com.example.variables;

public class DataF {
    public int i; // In this line, we declare public variable i
    public static int x; // In this line, we declare public static variable x
    DataF() { x=0; } // This constructor initializes static variable x
}
```



Java OOP Constructor

- A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations – except that they use the name of the class and have no return type.
- An example (pls see previous slide):

```java
package com.example.variables;

public class DataF {
    public int i; // In this line, we declare public variable i
    public static int x; // In this line, we declare public static variable x
    DataF() { x=0; } // This constructor initializes static variable x
}
```

Java, OOP, and Destructor Method

- Because Java is a garbage collected language, we cannot predict when (or even if) an object will be destroyed. Hence there is no direct equivalent of a destructor.

MyOnClickListener.java | DataF.java | DoWhile.java

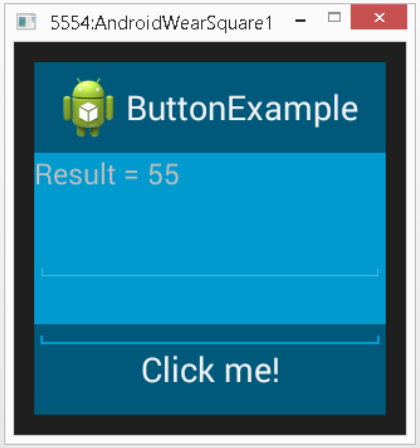
```java
package com.example.buttonexample;

public class DoWhile
{   // finalize() is called automatically when object is destroyed
    protected void finalize () {
        DataF.x = DataF.x + 5;
    }
}
```

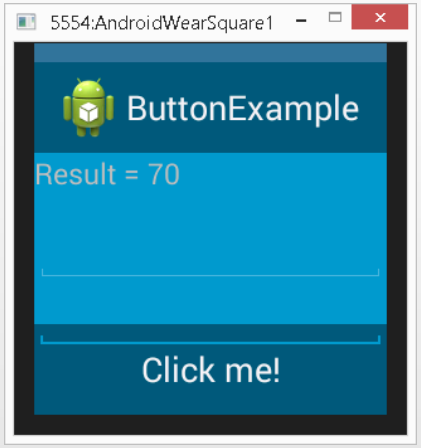
MyOnClickListener.java | DataF.java | DoWhile.java

```java
package com.example.buttonexample;
public class DataF
{
    public static int x=55; // Public static variable x
}
```

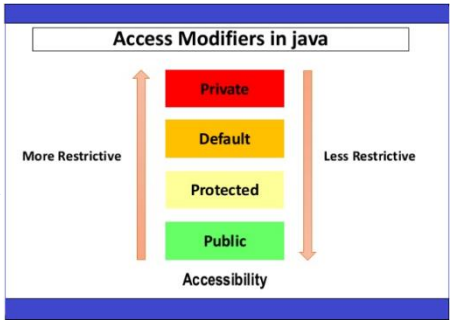
*MyOnClickListener.java | DataF.java | DoWhile.java

```java
package com.example.buttonexample;

import android.view.View;
import android.view.View.OnClickListener;

public class MyOnClickListener
    implements OnClickListener {
        FullscreenActivity caller;
        public MyOnClickListener(FullscreenActivity activity) {
            this.caller = activity;
        }
        public void onClick(View view) {
            DoWhile d1 = new DoWhile();
            DoWhile d2;
            d1 = null;
            d2 = null;
            System.gc(); // Garbage collection
            caller.textView.setText("Result = " + Integer.toString(DataF.x));
        }
    }
```





Java Access Modifiers



- A **private** member is only accessible within the same class as it is declared
- A member with no access modifier is only accessible within classes within classes in the same package
- A **protected** member is accessible within all classes in the same package and within subclasses in other packages
- A **public** member is accessible to all classes (unless it resides in a module that does not export the package it is declared in)

Do you have any questions or comments?

Thank you
for your attention !

In this presentation:

- Some icons were downloaded from flaticon.com and iconscout.com