Unit 8:

More on Java (Event Handling) and Features, for Android Apps

Mobile Application Development (MAD)
CCIT4059, 2025-2026

Outline

- Classes and Objects
  - Fields, Methods, Constructors
  - Object Memory, Reference, Garbage Collection
- Important OOP Features: Encapsulation, Inheritance, Polymorphism
- Interface, Event Handling
- Others

** This unit introduces Java and OOP features for developing Android apps in a bit more details for the course, while it is not supposed to be a "complete" Java book or tutorial.*

Classes and Objects

- Developing Android Apps with Java requires further understanding and using Object-Oriented (OO) programming approach.
  - Object-Oriented (OO) programming involves defining classes of objects, and applying OO features such as Encapsulation, Inheritance, and Polymorphism.
- Classes and their Objects
  - Objects represent entities (*e.g.* `aApple`, `bApple`, `cCircle`) with attributes (`color`, `radius`) and behaviors (`getColor()`, `setRadius()`)
  - Objects of same type defined using same class
    - Objects `aApple` and `bApple` of class `Apple`
  - A class is a template or blueprint that defines what an object's attributes/properties (fields , or data members) and behaviors (methods) will be
  - An object is an *instance* of a class, and many objects/ instances can be created from a class

Classes and Objects
(Declare Class – Syntax)

1) A class *without* specified "direct" superclass:

```
<modifier(s)> class <class name> {
    <class body>
}
```

* *A class without specified superclass will inherit from a root class (`Object` in Java)*

```
public class HiWorld {  // class (named HiWorld) declared
// ... Fields, Constructors, Methods
```

2) A class inherits from a specified superclass:

```
<modifier(s)> class <class name> extends <superclass name> {
    <class body>
}
```

```
public class MainActivity extends AppCompatActivity {
// ... Fields, Constructors, Methods
```

3) A class may also inherit from a specified superclass AND implements interface(s):

```
<modifier(s)> class <class name> extends <superclass name>
implements <interface(s)> { <class body>  }
```

Classes and Objects (Fields, Methods, Constructors)

- Classes (or objects) typically combine specific attributes with behaviors
  - Attributes are called ***fields*** in Java that could be accessed by reading and/or writing when an application runs
    - *E.g.* grade of a student, or balance of an account
  - Behaviors are called (implemented as) ***methods*** in Java that are accessed by calling to execute sequences of code including reading/writing attributes and performing other tasks
    - *E.g.* assigning a grade, or making a deposit into a saving account

Classes and Objects (**Fields**, Methods, Constructors)

- Declare a **field**
  - within the class declaration but outside all methods or constructors, with general syntax similar to declare local variables within methods or constructors:

<data type> <field name>;

  - A field may be of a primitive, array, or a class type (similar to local variables), *e.g.*

```
public class HiWorld   {
   String hiStr; // a field, declared outside method/constructor
   int aNum;    // another field
    HiWorld() {  // a constructor …
       int localVar;  // a local variable, declared inside constructor
         // ...
```

Classes and Objects (Fields, **Methods**, Constructors)

- Declare a **method**
  - within the class declaration, similar to declare a function in C
  - General Syntax

```
<modifier(s)> <return type> <method name> ( <parameter(s)> ){
    <method body>
}
```

E.g.

```
public class HiWorld   {
  String hiStr;   // a field
// ...
   void sayHi(String byWhom) {  // a method
     System.out.println(hiStr + " By " + byWhom);
   }
// ...
```

Classes and Objects (Fields, Methods, Constructors)

- Access fields and methods of a specific class/object in two different ways:
  - Access *inside* its own class
    - Dot-notation form is often optional, in most situation
    - In general, keyword `this` is used for dot-notation form
  - Access *outside* its own class
    - Use a dot-notation form (similar to accessing C's `struct` element)
    - In general, member to be accessed has to be associated to a specific object, *e.g.* in a form of

```
<object name>.<field name>
<object name>.<method name>(<argument(s)>)
```

Classes and Objects (Keyword **`this`** - Self-Referencing)

- Accessing fields and methods of its own (the object itself) ***inside*** its class, using keyword **`this`**
- Keyword **`this`** is used for *self-referencing:*
  - referring members of the object itself
  - General Syntax (with dot-notation):

```
this.<field name>
this.<method name>(<argument(s)>)
```

** Keyword **`this`** is also used for calling another constructor from a constructor*

Classes and Objects (Keyword **`this`** - Self-Referencing)

- Often, dot-notation with **`this.`** is optional, as long as it does not cause any confusion
  - *E.g.* in case of same name used for both local variable and field, dot-notation with **`this.`** must be used for the field

- In our `HiWorld` class example, the following two statements are essentially identical, when accessing a field `hiStr` inside a constructor or method

```
hiStr = new String("Hi World!");
this.hiStr = new String("Hi World!");
```

Classes and Objects (Fields, Methods, Constructors)

- Accessing fields and methods of an object, which is ***outside*** its own class
- General Syntax (with dot-notation):

  <object name>.<field name>

  <object name>.<method name>(<argument(s)>)

  – *E.g.* In "PureJava" case, accessing `Button` object in another class `PureJavaActivity`

```
        Button aBut;          // declare a Button field
// ...
        aBut.setText( ... );
```

***Object* Name, `aBut`**

Object Method Call, `setText()`

Classes and Objects (Class vs. Instance Fields and Methods)

- Although typical fields and methods in most applications are associated to specific objects, there are basically two types:
  - ***Instance/Object*** field and *instance/Object* method, also known as *non-static field/method* (* the type we have been dealing with so far)
  - ***Class*** field and *class* methods, also known as *static field/method*, with keyword `static`

** Very often, fields and methods are generally referred to instance field or instance method, if they are not specified to be associated to which type (class or instance)*

Classes and Objects (Class vs. Instance Fields and Methods)

- An *instance / object* field stores an attribute that is associated with each object
- Each object has a separate copy of attribute
  - *E.g.* the field `age` of `peter` (an object of class `Student`) has a value `18`, while that of `amy` has `16`
  - Declaring: As discuss earlier, *e.g.*

    ```
    < data type> <field name > ;
    ```

    E.g. `String aStr;  int age;`
  - Accessing: As discuss earlier, *e.g.*

    ```
    <object name>.<method name>(<argument(s)>)
    ```

    E.g. `aBut.setText( ... );`

Classes and Objects (Class vs. Instance Fields and Methods)

- A *class* field generally stores an attribute that is associated with the whole class
  - All objects created from that class share this
  - Declare class field/method with **`static`**

```
static int classNum; // declare a static (class) field

// declare a static (class) method, below

public static int [] getM6(int totNum, int min, int max);
```

  - Class field / method is often accessed directly by the class name, *e.g.*

```
//e.g. convert an array to string, with class method of class Arrays

String aStr = Arrays.toString(intArray);
```

***Class* Name, `Arrays`** **_Class_ Method Call, `toString()`**

Classes and Objects (Fields, Methods, **Constructors**)

- When object is created/constructed, specific ***constructor*** is invoked with **`new`** operator:

```
new <class name>(<arguments>)
```

E.g. `new String("This is MAD")`

- Constructor (similar to a method) is a block of code that is declared in a class for constructing an object when it is newly created (esp. for initializing or setting up)
  - Must use the class name, without return type
  - Mainly for initializing objects
  - Called when creating object with keyword `new`
- Each class may have more than one constructor, known as overloading feature
  - Different constructors of a class have no return type, with *different arguments*

Classes and Objects
(Overloading Methods and Constructors)

- **Overloading** methods / Constructors: two or more methods / constructors have the same name inside a class, based on rules (below) to make them distinguishable
  - They have different numbers of parameters

```
MADClass(); // overloading constructors

MADClass(int abc);
```

  - Their parameters are of different data types when the number of parameters is the same

```
void setFe(float abc); // overloading methods

void setFe(String abc);
```

** This is closely related to the concept of polymorphism*

Classes and Objects (Overriding Methods)

- **Overriding** methods: An identical method in superclass and subclass
  - Both superclass and subclass has the name method (same return type, name & parameters)
  - To override a superclass method, a subclass declares a method with the same method return type and signature (name and parameters) as the superclass method
  - The superclass method can be accessed from the subclass by keyword **`super`**, *e.g.*

```
super.setDM(123);
```

Classes and Objects (Object Memory, Reference, Garbage Collection)

- Objects are created with allocated memories (via **`new`** operator)
  - Without proper memory management to destroy “useless” objects, they will fill up the available memory space eventually
- Java handles this task by a mechanism called *Garbage Collection*, which will remove unreferenced objects automatically
  - Unlike other programming language (like C) that it is developer’s job to remove “useless” objects

Classes and Objects
(Object Memory, Reference, Garbage Collection)

- Class is a reference type; and its objects in general are assigned, passed, and manipulated with its reference memory, *e.g.*

```
String a = new String("ABC");
String b = a;
```

- Code above would not copy or construct a new object for object `b` from object `a`
  - This = assignment only lets `b` reference to the same object of `a`; After execution, both `a` and `b` reference to the same `String` with `"ABC"`

** This is similar to the operation of assigning pointer in C, which only copies the reference memory address of `a` to that of `b`.*

Encapsulation, Inheritance, Polymorphism

- Encapsulation is also called information hiding
- The major benefits include:
  - components can be replaced/ revised easily
  - protect the integrity (prevent users from setting the internal data into an invalid or inconsistent state)
  - reduces complexity and thus increases robustness (limit the interdependencies between software components)

Encapsulation, Inheritance, Polymorphism

- There are different access (visibility) types associated to encapsulate implementation details
- Java includes keywords `public`, `private`, `protected` for four different access types (and a default one without access modifier)
  - **`public`** fields and methods are accessible to everyone
  - **`private`** fields and methods are accessible only within the class itself
  - **`protected`** access is an intermediate level of access between public and private
    - its ***subclasses*** (***descendant*** classes), if any
    - all classes in the ***same package***, including the class itself
  - The fourth type of access level which is without access modifier is also known as *package-private*, which let all classes in the *same package* access.

Encapsulation, Inheritance, Polymorphism (Example)

```
...
Service obj = new Service();

obj.memberOne = 10;

obj.memberTwo = 20;

obj.doOne();

obj.doTwo();

...
```

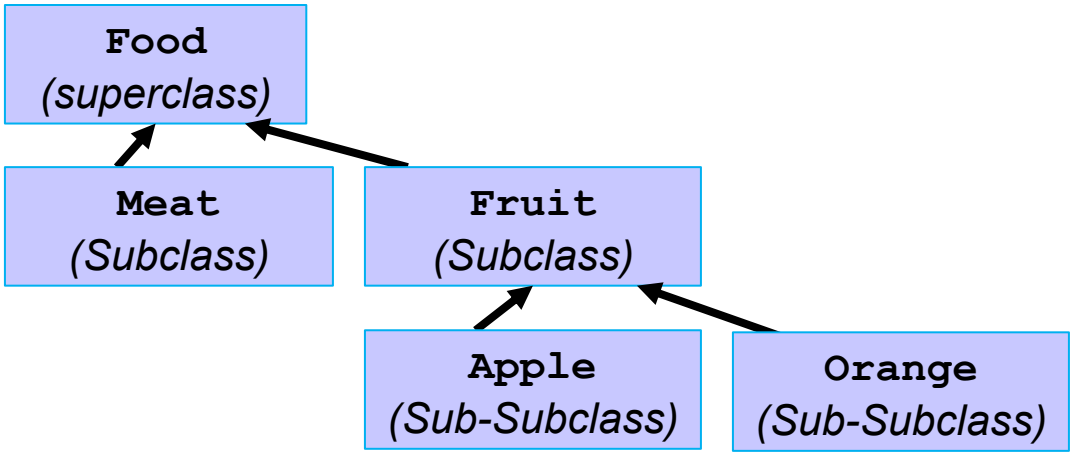
Client

```
class Service {
    public  int memberOne;
    private int memberTwo;

    public void doOne() {

    ...

    }
    private void doTwo() {

    ...

    }

}
```

Service

Encapsulation, **Inheritance**, Polymorphism

- Inheritance
  - Subclass inherits (`extends`) Superclass
    - Like a child (descendant) inherits a parent (ancestor)
      - All children are different, with some common features from the parent
    - Similarly, a parent may further inherits a grandparent, etc., to form an inheritance hierarchy
    - In Java, each subclass can have only one superclass; but each superclass may have many subclasses



Encapsulation, **Inheritance**, Polymorphism

- When creating a class, rather than declaring completely new members, we may designate that the *new specialized class* should inherit the members of an *existing general class*
  - Existing general class is called **superclass**
  - New specialized classes are called **subclasses**
  - In Java, each class is allowed to have ONLY ONE direct superclass, and each superclass may have many direct subclasses

- Every class has a superclass (except root class)
  - If the class declaration does not explicitly designate the superclass with the `extends` clause, then the class's superclass is the `Object` (the root) class in Java

Encapsulation, **Inheritance**, Polymorphism (Keywords `this` vs. `super`)

- Keyword **`this`** is for *self-referencing*
  - it is used to 1) refer to members of the object itself, and 2) call another constructor within a constructor
- Keyword **`super`** references to superclass
  - it is used to 1) refer to members of superclass, and 2) call a specific constructor of superclass,
  - *e.g.* the following statement in the method body calls the `onCreate()` method of its superclass

```
protected void onCreate(Bundle savedInstanceState) {//app starts here

    super.onCreate(savedInstanceState); // always call superclass
```

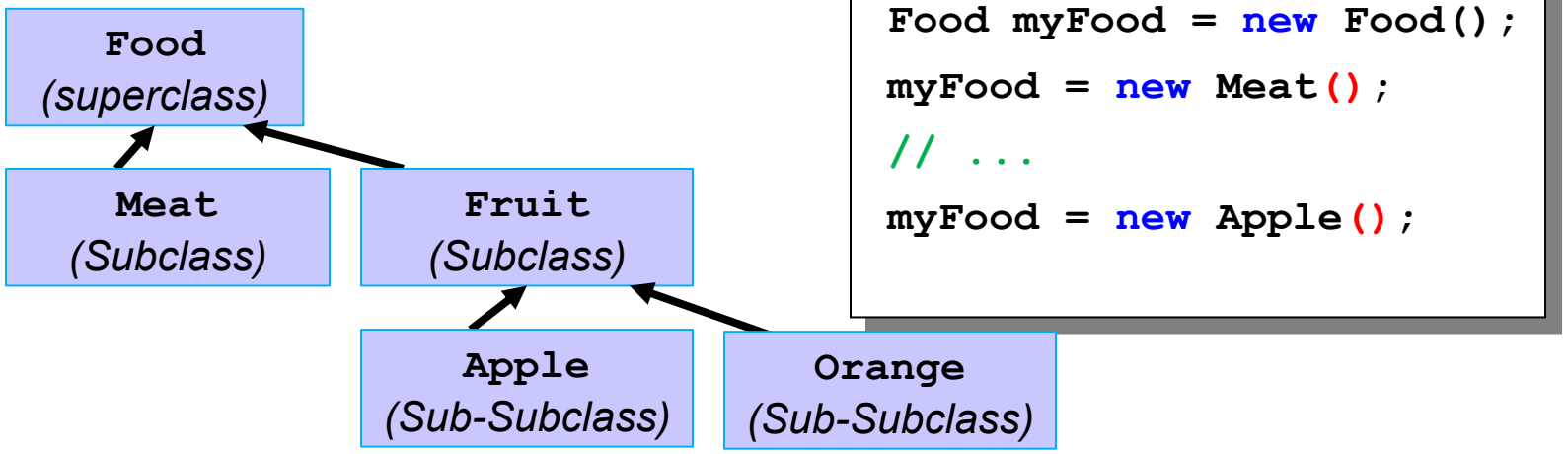
Encapsulation, Inheritance, **Polymorphism**

- Polymorphism allows one name to be used differently, and different OO programming languages support this in various ways, *e.g.*
  - Two or more methods / constructors have the same name, based on some rules to make them distinguishable: this is called method / constructor overloading, *e.g.*

```
void setFe(float abc);  // overloading methods
void setFe(String abc); // overloading methods
```

Encapsulation, Inheritance, **Polymorphism**

- Polymorphism also supports a variable (with a specific name) referring to objects from different subclasses in the same inheritance hierarchy, called Subtype polymorphism (or subtyping). *E.g.*
  - if **`Fruit`** and **`Meat`** are subclasses of **`Food`**, and **`Apple`** is a subclass of **`Fruit`**




```
Food myFood = new Food();
myFood = new Meat();
// ...
myFood = new Apple();
```

Interface, Event Handling

- A Java interface is a reference type, similar to a class, that can contain only constants, method signatures (abstract methods)
  - There are no method bodies
- No instances can be created, and they can only be implemented by classes or extended by other interfaces
  - Each class can implement many interfaces
- To use an interface, you write a class that implements the interface

Interface, Event Handling

- General Syntax of using (implementing) interface(s):

```
<modifiers> class <classname> implements <interface-names> {
    // other fields, constructors, and methods
    // implement the methods declared in the interface
}
```

- Example of using (implementing) interface(s):

```
class MyClass implements MyInterfA, MyInterfB {
// implement the methods declared in the interfaces A,B
    public void methForA() {... }
    public void methForB() { ... }
}
```

Interface, Event Handling

- Interface is especially important in Graphical User Interface (GUI), as most GUI components are often interacted via interfaces in Java for Event Handling, *e.g.* “Listener” interfaces in Android

```
public class PureJavaActivity extends AppCompatActivity
implements View.OnClickListener {
//implements interface for button Click Event Handling
// ...
    RelativeLayout aRLayout = null;      // field for UI layout
    Bicycle bikeA;                       // a field, Bicycle
    Button aButA;                   // a field, Button

    @Override // for interface: View.OnClickListener
// BELOW method, called if button is clicked
    public void onClick(View v) {
        aButA.setX(aButA.getX() + bikeA.getSpeed()); // move
    }
// ...
```

Interface, Event Handling

- An action involving a GUI object, such as clicking a button, is called an event
- The mechanism to process events is called event handling
- The event-handling model of Java is based on the concept known as the delegation-based event model
  - With this model, event handling is implemented by two types of objects: 1) event source objects, and 2) event listener objects

Interface, **Event Handling**

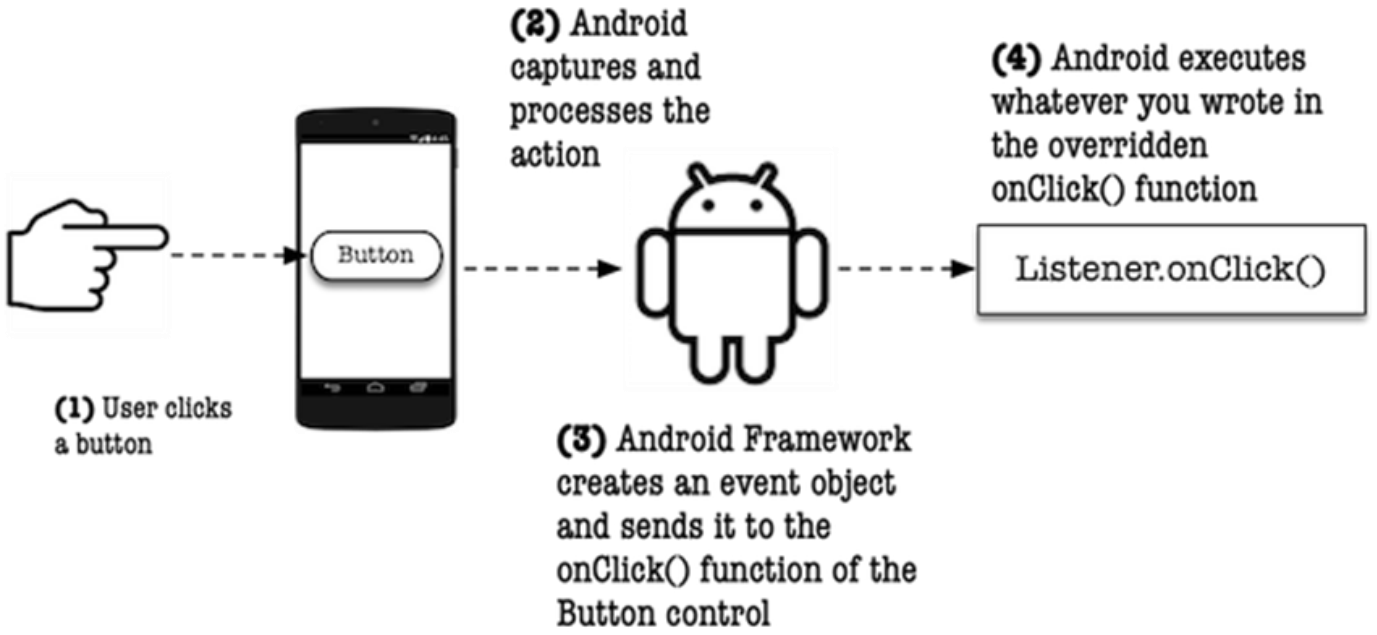
- An *event source* is a GUI object where an event occurs, and generates events
  - Buttons, text boxes, list boxes, and menus are common event sources in GUI-based applications

- Event listeners and event handlers are fundamental concepts in programming, especially in the context of user interfaces and interactive applications in Android Studio.

- An *event listener* object is an object that includes a method that gets executed in response to the generated events
  - A listener must be associated, or registered, to a source, so it can be notified when the source generates events

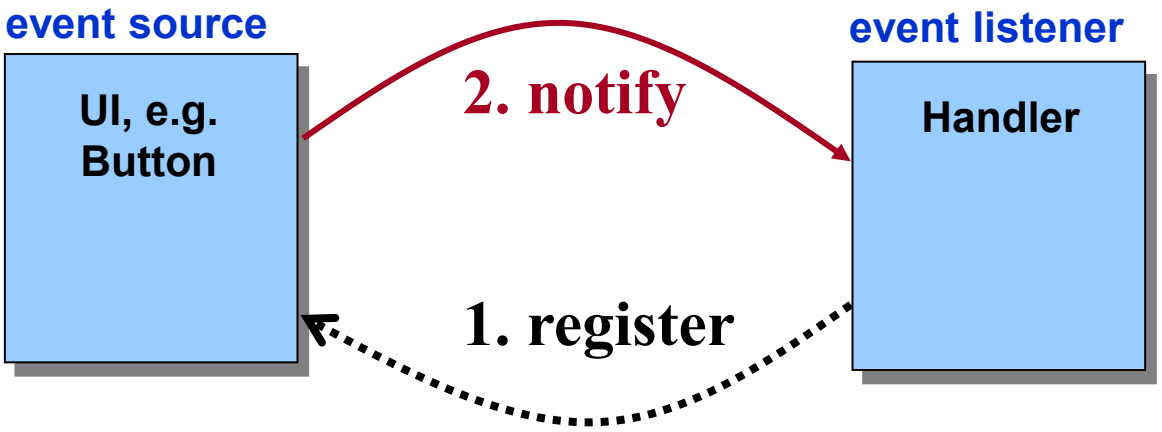
Event Listeners and Event Handlers

- Event Handlers are the functions that contain the code to be executed when an event occurs.
- They are often associated with event listeners. When an event listener detects an event, it calls the corresponding event handler to perform the necessary actions.



Interface, **Event Handling**

- A listener must first be registered to an event source, in order to handle a source event.
  - Once registered, it will get notified when the event source generates events
- If a button is pressed then this action or event gets registered by the event listener and then the task to be performed by that button press is handled by the event handler, it can be anything like changing the colour of the text on a button press or changing the text itself, etc.



Interface, **Event Handling**

- Registration and notification are specific to event types, *e.g.* the Java interfaces below in Android
  - `OnClickListener()`
  - `OnTouchListener()`
  - `SensorEventListener()`
- Among different types of events, the `OnClick` Button event is common where an `OnClickListener` must be created and registered in order to process the event

Interface, **Event Handling**

- To illustrate this, let us reference to the earlier "Pure Java" `Activity`

```
package com.example.mad.madm31purejava;    // package statement
import android.content.Context;             // import statements
// ... and more import statements here

public class PureJavaActivity extends AppCompatActivity // a new Java class
      implements View.OnClickListener {//implements interface for button Click
    Button aButA;                    // a field, Button: UI component
// ...

    @Override // for interface: View.OnClickListener
    public void onClick(View v) {  // a method, notified by registered button
        aButA.setX(aButA.getX() + bikeA.getSpeed()); // move
    }

// ...
    aButA = new Button(this);      // new (create) button to be added to layout
    aButA.setOnClickListener(this); // register "this" as button's listener
```

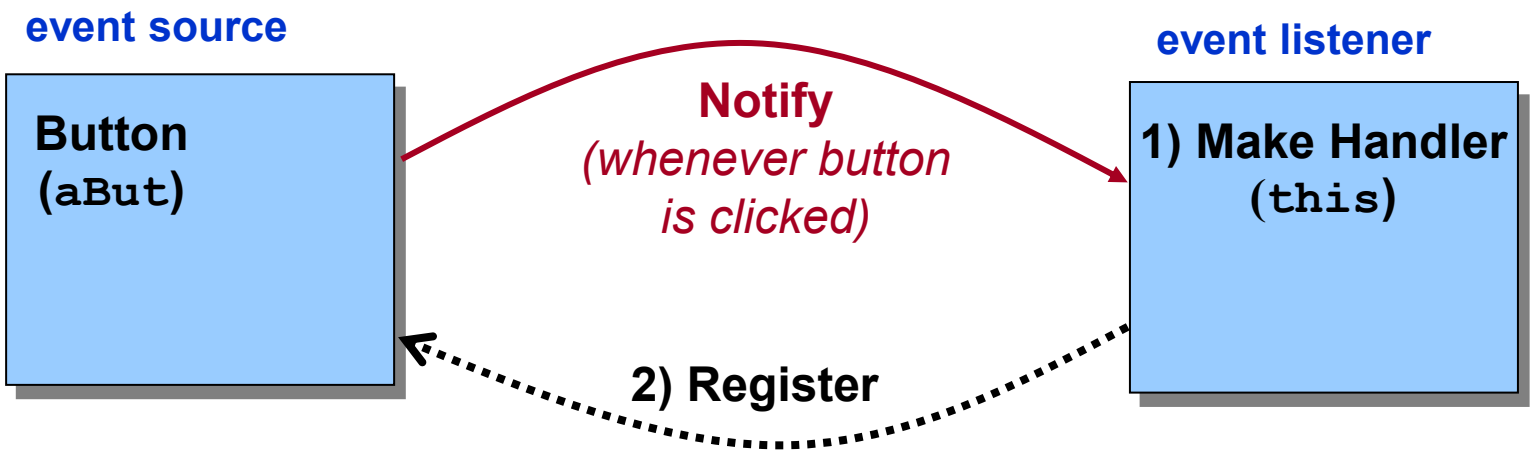
Interface, **Event Handling**

1) Make the object `this` as a Handler, the action listener (`OnClickListener`)

```
implements View.OnClickListener {//implements interface
 // ...
 public void onClick(View v) {// implement interface's method
 // ...
```

2) Register an instance of Handler as the action listener of the button:

```
 aBut = new Button(this);   // new button
 aBut.setOnClickListener(this); // set this as listener
```



1. Introduction to implements in Java (Android)

- In Java, the keyword **implements** is used by a class to implement one or more interfaces.
  - An **interface** defines a set of method signatures (what must be done), while the class that implements it provides the actual method implementations (how it is done).
- In Android app development using Android Studio, implements is commonly used because many Android components rely on event-driven programming and callback mechanisms, such as:
  - Button click handling
  - Sensor data updates
  - Location or network callbacks
  - Lifecycle or system event listeners
- Using implements allows an Android class (e.g. Activity) to respond to system or user events in a structured and reusable way.

Purpose of Using implements in MAD (Contract Between Class and System)

- When a Java class uses implements, it must provide all methods declared in the interface.
  - This ensures the class follows a strict contract, which improves:
    - Code correctness
    - Predictability of behaviour
    - Compatibility with Android framework components
- In Android, many system features expect a class to implement a specific interface before it can interact with the system correctly.

Purpose of Using implements in MAD (Handling User Interaction and Events)

- Android applications are event-driven.
- Interfaces are widely used to define event listeners, such as:
  - User clicks
  - Touch gestures
  - Background task completion
- By using implements, an Activity or class can directly handle these events, making the app responsive to user actions.
- Example
  - A button click event is defined by an interface
  - The Activity implements that interface
  - The Activity provides the response logic
- This design improves separation of concerns between UI components and logic.

Allowing Multiple Interface Inheritance

- Java does not allow multiple inheritance of classes, but it does allow a class to Implement multiple interfaces.
- is especially useful in Android when a class needs to:
  - Respond to multiple event types
  - Act as a listener for different system services
  - Integrate several independent behaviours
- Using implements avoids ambiguity while keeping the code flexible and modular.

implements vs extends

In practice, Android classes often use both together:

- extends are for inheriting core Android behaviour
- implements are for handling events and callbacks

Keyword	Usage in Android	Key Purpose
`extends`	Inherit from a parent class (e.g. `Activity`, `AppCompatActivity`)	Reuse existing functionality
`implements`	Implement interface(s) (e.g. listeners, callbacks)	Respond to events / enforce behaviour

Others

- There are many advanced features supported by Java, including:
  - Exception Handling
    - Handling error condition that can occur during program execution
  - Threads
    - Multithreaded execution supports concurrent program
  - Annotation
    - A form of syntactic metadata that can be applied to Java source code
    - A common annotation `@Override` is for checking if the method is an overriding method. It causes compile error if method is not found. *E.g.*

```
@Override // Override onClick method below expected
public void onClick(View v) {  // ...
}
```

References

- Part of this slide set is prepared and extracted from the followings:
  - Friesen, J. (2014) *'Learn Java for Android Development'*, Apress Media
  - Gargenta, M and Nakamura, M. (2014) *'Learning Android: Develop Mobile Apps Using Java and Eclipse'*, 2ed, O'Reilly Media, Inc.
  - Gok, N. and Khanna, N. (2013) *'Building Hybrid Android Apps with Java and JavaScript'*, O'Reilly Media, Inc.
  - Nolan, G., Cinar, O. and Truxall, D. (2014) *'Android Best Practices'*, Apress Media
  - Wolfson, M. (2013) *'Android Developer Tools Essentials'*, O'Reilly Media, Inc.
  - Google Inc. 'Website for Android Developers', http://developer.android.com/
  - Google Inc. 'Website for Android Sources', https://source.android.com/
  - Apple Inc. 'Website for Apple Developers', Website: https://developer.apple.com/
  - Microsoft Corp. 'Website for Windows Developers', https://developer.microsoft.com/en-us/windows
  - Oracle "Website for Java" http://docs.oracle.com/javase
  - Wikipedia Website: http://en.wikipedia.org
- This set of slides is for teaching purpose only