JAVA FOR ANDROID : A BRIEF

ROMI FADILLAH RAHMAT, B.COMP.SC, M.SC

WHAT WE CONCERNED..

- Moore's law affects processors and memory (doubling their power approximately every two years), no such law affects battery life.
- PC - > When processors were slow, developers used to be concerned about CPU speed and efficiency.
- Mobile -> Mobile developers, on the other hand, need to be concerned about energy efficiency.

JAVA PRIMITIVE TYPE

- There are 2 data types in Java : Primitive data Types and Object Data Types
- Primitive Data Type ≠ Object
- Modification only with predefined operators = “+”, “-”, “&”, “|”, “=”, and so on.
  - **boolean** The values true or false
  - **byte** An 8-bit 2’s-complement integer
  - **short** A 16-bit 2’s-complement integer
  - **int** A 32-bit 2’s-complement integer
  - **long** A 64-bit 2’s-complement integer
  - **char** A 16-bit unsigned integer representing a UTF-16 code unit
  - **float** A 32-bit IEEE 754 floating-point number
  - **double** A 64-bit IEEE 754 floating-point number

OBJECT AND CLASSES

- Java is an object-oriented language and focuses not on its primitives but on objects—combinations of data, and procedures for operating on that data.
- A *class* defines the fields (data) and methods (procedures) that comprise an object.

```
public class Trivial {
        /** a field: its scope is the entire class */
        private long ctr;
        /** Modify the field. */
        public void incr() {
        ctr++;
        } }
```

OBJECT CREATION

- A new object, an instance of some class, is created by using the new keyword

```
Trivial trivial = new Trivial();
```

- On the left side of the assignment operator "=", this statement defines a variable, named trivial.
- The variable has a type, Trivial, so only objects of type Trivial can be assigned to it.
- The right side of the assignment allocates memory for a new instance of the Trivial class and initializes the instance.
- The assignment operator assigns a reference to the newly created object to the variable.

- Java guarantees that all fields are automatically initialized at object creation: boolean is initialized to false, numeric primitive types to 0, and all object types (including Strings) to null.
- Local variables must be initialized before they are referenced!

OBJECT CREATION - CONSTRUCTOR

- Purpose of constructor to its class definition : take greater control over the initialization of an object
- A constructor definition looks like a method except that it doesn't specify a return type.
- Its name must be exactly the name of the class that it constructs

```
public class LessTrivial {
  private long ctr; /** a field: its scope is the entire class */
  public LessTrivial(long initCtr) {
    ctr = initCtr;
  } // Constructor: initialize the fields
  public void incr() { ctr++; } /** Modify the field. */
}
```

OBJECT CREATION - CONSTRUCTOR

- The definition of Trivial given earlier (which specifies no explicit constructor), actually has a constructor that looks like this:

```
public Trivial() { super(); }
```

- Since the LessTrivial class explicitly defines a constructor, Java does *not* implicitly add a default. That means that trying to create a LessTrivial object, with no arguments, will cause an error:

```
LessTrivial fail = new LessTrivial(); // ERROR!!
LessTrivial ok = new LessTrivial(18); // ... works
```

OBJECT CREATION - CONSTRUCTOR

- There are two concepts that it is important to keep separate: *no-arg constructor* and *default constructor*.
- A default constructor is the constructor that Java adds to your class, implicitly, if you don't define any other constructors. It happens to be a no-arg constructor.
- A no-arg constructor, on the other hand, is simply a constructor with no parameters. There is no requirement that a class have a no-arg constructor. There is no obligation to define one, unless you have a specific need for it.

OBJECT CREATION - CONSTRUCTOR

```
public class LessTrivial {
        /** a field: its scope is the entire class */
        private long ctr;
        /** Constructor: init counter to 0 */
        public LessTrivial() { this(0); }
        /** Constructor: initialize the fields */
        public LessTrivial(long initCtr) { ctr = initCtr; }
        /** Modify the field. */
        public void incr() { ctr++; } }
```

OBJECT CLASS AND ITS METHOD

- The Java class Object—java.lang.Object—is the root ancestor of every class. Every Java object is an Object.
- If the definition of a class does not explicitly specify a superclass, it is a direct subclass of Object.
- The Object class defines the default implementations for several key behaviors that are common to every object.
- Unless they are overridden by the subclass, the behaviors are inherited directly from Object.

OBJECT, INHERITANCE AND POLYMORPHISM

- A language is said to be polymorphic if objects of a single type can have different behavior.
- This happens when subtypes of a given class can be assigned to a variable of the base class type.

OBJECT, INHERITANCE AND POLYMORPHISM

Subtypes in Java are declared through use of the extends keyword. Here is an example of inheritance in Java:

```
public class Car {
    public void drive() {
        System.out.println("Going down the road!");     } }
public class Ragtop extends Car { // override the parent's definition.
public void drive() {
    System.out.println("Top down!");       // optionally use a superclass method
    super.drive();
    System.out.println("Got the radio on!");  }
```

OBJECT, INHERITANCE AND POLYMORPHISM

```
Car auto = new Car();
auto.drive();
auto = new Ragtop();
auto.drive();
```

Result :

```
Going down the road!
Top down!
Going down the road!
Got the radio on!
```

- Like many other object-oriented languages, Java supports type casting to allow coercion of the declared type of a variable to be any of the types with which the variable is polymorphic:

```
Ragtop funCar;
Car auto = new Car();
funCar = (Ragtop) auto; //ERROR! auto is a Car, not a Ragtop!
auto.drive();
auto = new Ragtop();
Ragtop funCar = (Ragtop) auto; //Works! auto is a Ragtop
auto.drive();
```

FINAL AND STATIC DECLARATIONS

- There are 11 modifier keywords that can be applied to a declaration in Java
- A **final** declaration is one that cannot be changed. Classes, methods, fields, parameters, and local variables can all be final
- When applied to a class, final means that any attempt to define a subclass will cause an error.
- When applied to a method, final means that the method cannot be overridden in a subclass.
- When applied to a variable—a field, a parameter, or a local variable—final means that the value of the variable, once assigned, may not change.
- For parameters, final means that, within the method, the parameter value always has the value passed in the call. An attempt to assign to a final parameter will cause an error.

FINAL AND STATIC DECLARATIONS

- A static declaration belongs to the class in which it is described, not to an instance of that class. The opposite of static is *dynamic*. Any entity that is not declared static is implicitly dynamic.

```
public class QuietStatic {
    public static int classMember;
    public int instanceMember; }

public class StaticClient {
    public void test() {
        QuietStatic.classMember++;
        QuietStatic.instanceMember++; // ERROR!!
        QuietStatic ex = new QuietStatic();
        ex.classMember++; // WARNING!
        ex.instanceMember++;
    } }
```

FINAL AND STATIC DECLARATIONS

- The static member classMember is an attribute of the class; you can refer to it simply by qualifying it with the class name. On the other hand, instanceMember is a member of an *instance* of the class. An attempt to refer to it through the class reference causes an error. That makes sense. There are many different variables called instance Member, one belonging to each instance of QuietStatic.

FINAL AND STATIC DECLARATIONS

- Static class members allow you to maintain information that is held in common by all members of a class.

```
public class LoudStatic {
    private static int classMember; private int instanceMember;
public void incr() { classMember++;
    instanceMember++;}
@Override public String toString() {
    return "classMember: " + classMember + ", instanceMember: " + instanceMember;}
public static void main(String[] args) {
    LoudStatic ex1 = new LoudStatic(); LoudStatic ex2 = new LoudStatic();
    ex1.incr();  ex2.incr();
    System.out.println(ex1);  System.out.println(ex2); }}
```

FINAL AND STATIC DECLARATIONS

Output:

classMember: 2, instanceMember: 1

classMember: 2, instanceMember: 1

- The initial value of the variable classMember in the preceding example is 0. It is incremented by each of the two different instances. Both instances now see a new value, 2.
- The value of the variable instanceMember also starts at 0, in each instance. On the other hand, though, each instance increments its own copy and sees the value of its own variable, 1.

- Static class and method definitions are similar in that, in both cases, a static object is visible using its qualified name, while a dynamic object is visible only through an instance reference.
- One significant difference in behavior between statically and dynamically declared methods is that statically declared methods cannot be overridden in a subclass. The following, for instance, fails to compile:

```
public class Star {
        public static void twinkle() { } }
public class Arcturus extends Star {
        public void twinkle() { } // ERROR!! }
public class Rigel {  // this one works
        public void twinkle() { Star.twinkle(); } }
```

- A dynamic class has access to instance members of the enclosing class (since it belongs to the instance). A static class does not. Here's some code to demonstrate:

```
public class Outer {
        public int x;
        public class InnerOne { public int fn() { return x; } }
        public static class InnerTube {
                public int fn() {          return x; // ERROR!!! }}}
public class OuterTest {
        public void test() {
                new Outer.InnerOne(); // ERROR!!!
                new Outer.InnerTube(); }}
```

- The class InnerTube is a part of the class Outer, but not of any *instances* of Outer. It has no way of identifying an x.
- The class InnerOne, on the other hand, because it is dynamic, belongs to an instance of Outer