Tutorial: Programming in Java for Android Development

Outline

- **Getting Started**
- Java: The Basics
- Java: Object–Oriented Programming
- Android Programming

Getting Started (1)

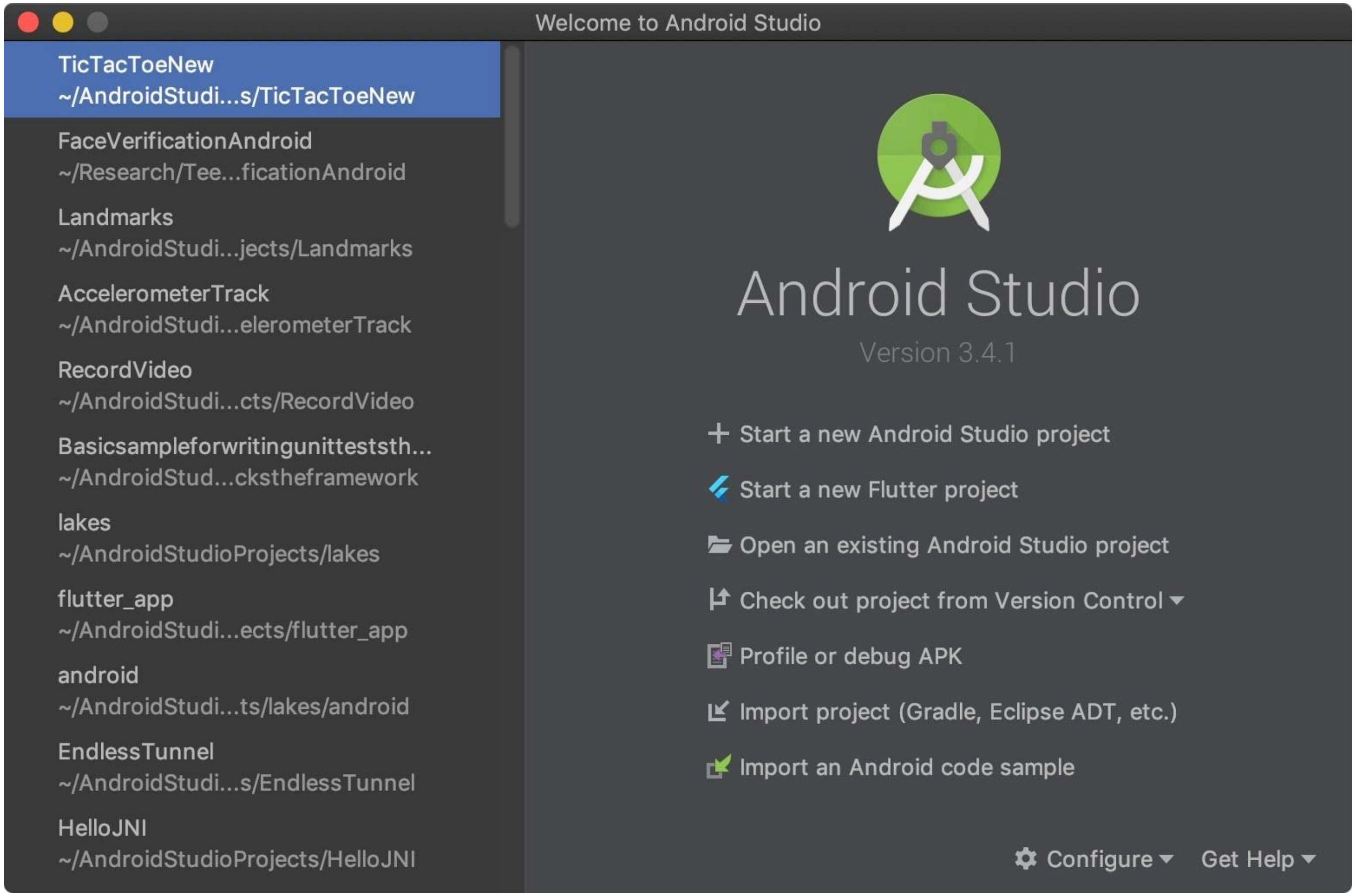
- Need to install Java Dev. Kit (JDK) *version 8* to write Java (Android) programs
  - **Don’t** install Java Runtime Env. (JRE); JDK is different!
  - Newer versions of JDK can cause issues with Android
- Can download JDK (free): https://adoptopenjdk.net/
  - Oracle’s JDK (http://java.oracle.com) free for *dev. only*; payment for commercial use
- Alternatively, for macOS, Linux:
  - macOS: Install Homebrew (http://brew.sh), then type `brew cask info adoptopenjdk8` at command line
  - Linux: Type `sudo apt install default-jdk` at command line (Debian, Ubuntu)

Getting Started (2)

- After installing JDK, download Android SDK from http://developer.android.com
- Simplest: download and install Android Studio bundle (including Android SDK) for your OS
- We’ll use Android Studio with SDK included (easy)

Getting Started (3)

- Install Android Studio directly (Windows, Mac); unzip to directory `android-studio`, then run `./android-studio/bin/studio.sh` (Linux)
- You should see this:



Welcome to Android Studio

TicTacToeNew
~/AndroidStudi...s/TicTacToeNew

FaceVerificationAndroid
~/Research/Tee...ficationAndroid

Landmarks
~/AndroidStudi...jects/Landmarks

AccelerometerTrack
~/AndroidStudi...elerometerTrack

RecordVideo
~/AndroidStudi...cts/RecordVideo

Basicsampleforwritingunitteststh...
~/AndroidStud...cksthefarmework

lakes
~/AndroidStudioProjects/lakes

flutter_app
~/AndroidStudi...ects/flutter_app

android
~/AndroidStudi...ts/lakes/android

EndlessTunnel
~/AndroidStudi...s/EndlessTunnel

HelloJNI
~/AndroidStudioProjects/HelloJNI

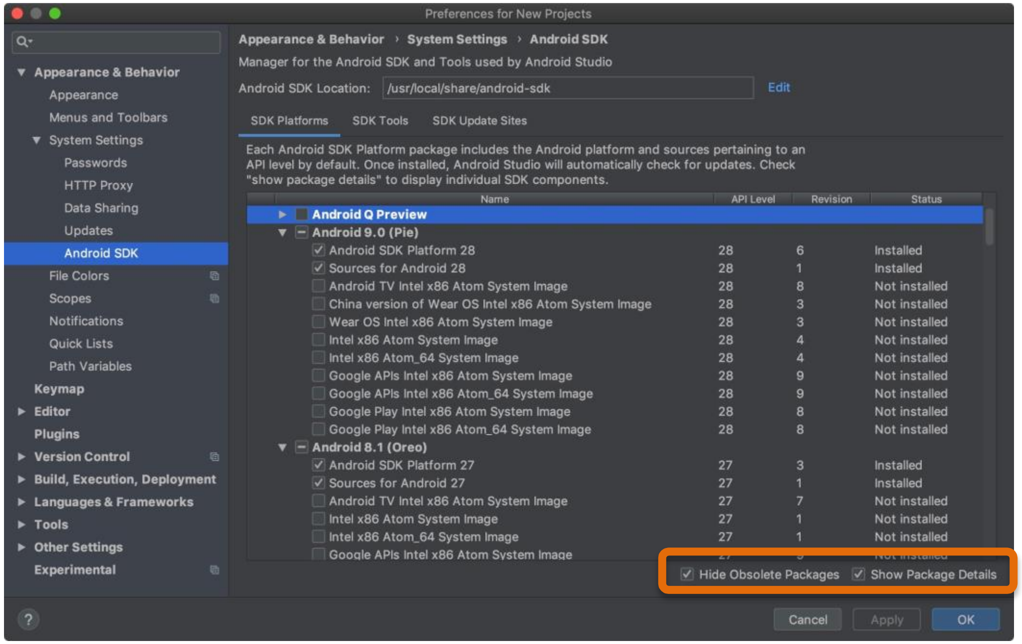
Android Studio

Version 3.4.1

+ Start a new Android Studio project
Start a new Flutter project
Open an existing Android Studio project
Check out project from Version Control ▾
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Configure ▾
Get Help ▾

Getting Started (4)

- Strongly recommend testing with real Android device
  - Android emulator slow; Genymotion faster [14], [15]
  - Install USB drivers for your Android device!
- Go to File
  - Recommended: Install Android 5–8 APIs
  - Don’t worry about system images for non-x86 arch.



Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: /usr/local/share/android-sdk

Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, Android Studio will automatically check for updates. Check "show package details" to display individual SDK components.

Name	API Level	Revision	Status
▶ □ Android Q Preview			
▼ ⊟ Android 9.0 (Pie)			
☑ Android SDK Platform 28	28	6	Installed
☑ Sources for Android 28	28	1	Installed
□ Android TV Intel x86 Atom System Image	28	8	Not installed
□ China version of Wear OS Intel x86 Atom System Image	28	3	Not installed
□ Wear OS Intel x86 Atom System Image	28	3	Not installed
□ Intel x86 Atom System Image	28	4	Not installed
□ Intel x86 Atom_64 System Image	28	4	Not installed
□ Google APIs Intel x86 Atom System Image	28	9	Not installed
□ Google APIs Intel x86 Atom_64 System Image	28	9	Not installed
□ Google Play Intel x86 Atom System Image	28	8	Not installed
□ Google Play Intel x86 Atom_64 System Image	28	8	Not installed
▼ ⊟ Android 8.1 (Oreo)			
☑ Android SDK Platform 27	27	3	Installed
☑ Sources for Android 27	27	1	Installed
□ Android TV Intel x86 Atom System Image	27	7	Not installed
□ Intel x86 Atom System Image	27	1	Not installed
□ Intel x86 Atom_64 System Image	27	1	Not installed
□ Google APIs Intel x86 Atom System Image	27	[illegible]	Not installed

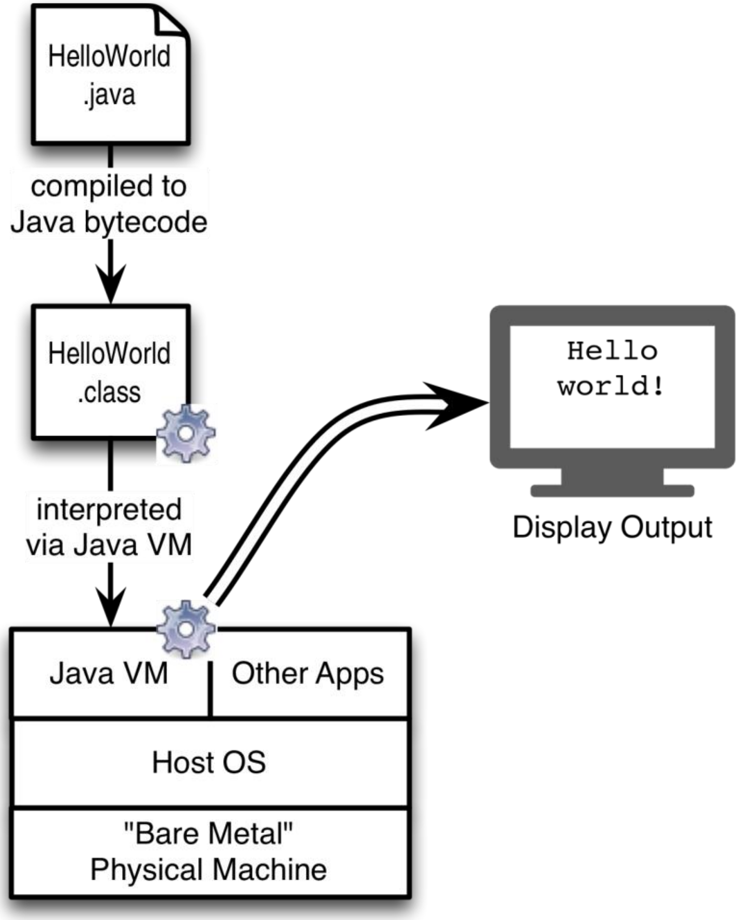
☑ Hide Obsolete Packages ☑ Show Package Details

Outline

- Getting Started
- **Java: The Basics**
- Java: Object–Oriented Programming
- Android Programming

Java Programming Language

- Java: general-purpose language: “write code once, run anywhere”
- The key: Java Virtual Machine (JVM)
  - Program code compiled to JVM bytecode
  - JVM bytecode interpreted on JVM
- We’ll focus on Java; see Chaps. 1–7 in [1].



Our First Java Program

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println(“Hello world!”);
   }
}
```

- Don’t forget to match curly braces `{ , }` or semicolon at the end!
- Recommended IDEs:
  - IntelliJ IDEA CE (free; http://www.jetbrains.com/student)
  - Eclipse (free; http://www.eclipse.org)
  - Text editor of choice (with Java programming plugin)

Explaining the Program

- Every `.java` source file contains one class
  - We create a class `HelloWorld` that greets user
  - The class `HelloWorld` must have the same name as the source file `HelloWorld.java`
  - Our class has `public` scope, so other classes can "see" it
  - We'll talk more about classes and objects later
- Every Java program has a *method* `main()` that executes the program
  - Method "signature" must be exactly `public static void main(String[] args) {}`
  - This means: (1) `main()` is "visible" to other methods; (2) there is "only one" `main()` method in the class; and (3) `main()` has one argument (`args`, an array of `String` variables)
  - Java "thinks" `main()`, `Main()`, `miAN()` are different methods
- Every Java method has curly braces `{,}` surrounding its code
- Every statement in Java ends with a semicolon, e.g., `System.out.println("Hello world!");`
- Program prints "`Hello world!`" to the console, then quits

Basic Data Types

- Java variables are instances of mathematical “types”
  - Variables can store (almost) any value their type can have
  - Example: the value of a `boolean` variable can be either `true` or `false` because any (mathematical) `boolean` value is *true* or *false*
  - Caveats for integer, floating–point variables: their values are subsets of values of mathematical integers, real numbers. Cannot assign *mathematical* 2^{500} to integer variable (limited range) or *mathematical* √2 to a floating–point variable (limited precision; irrational number).
  - Variable names must start with lowercase letter, contain only letters, numbers, `_`
- Variable *declaration*: `boolean b = true;`
- Later in the program, we might *assign* `false` to `b`: `b = false;`
- Java strongly suggests that variables be initialized at the time of declaration, e.g., `boolean b;` gives a compiler warning (`null` pointer)
- Constants defined using `final` keyword, e.g.,
  `final boolean falseBool = FALSE;`

Basic Data Types (2)

- Java’s primitive data types: [5]

Primitive type	Size	Minimum	Maximum	Wrapper type
`boolean`	1–bit	N/A	N/A	`Boolean`
`char`	16–bit	Unicode 0	Unicode $2^{16} – 1$	`Character`
`byte`	8–bit	–128	+127	`Byte`
`short`	16–bit	$–2^{15}$	$+2^{15} – 1$	`Short`
`int`	32–bit	$–2^{31}$	$+2^{31} – 1$	`Integer`
`long`	64–bit	$–2^{63}$	$+2^{63} – 1$	`Long`
`float`	32–bit	IEEE 754	IEEE 754	`Float`
`double`	64–bit	IEEE 754	IEEE 754	`Double`

Note: All these types are signed, except `char`.

Basic Data Types

- Sometimes variables need to be *cast* to another type, e.g., if finding average of integers:

```
int intOne = 1, intTwo = 2, intThree = 3, numInts = 2;
double doubOne = (double)intOne, doubTwo = (double)myIntTwo, doubThree =
(double)intThree;
double avg = (doubOne + doubTwo + doubThree)/(double)numInts;
```

- `Math` library has math operations like `sqrt()`, `pow()`,etc.
- `String`: immutable type for sequence of characters
  - Every Java variable can be converted to `String` via `toString()`
  - The `+` operation concatenates `Strings` with other variables
  - Let `str` be a `String`. We can find `str`'s length (`str.length()`), substrings of `str` (`str.substring()`), and so on [6]

Basic Data Types (4)

- A literal is a “fixed” value of a variable type
  - `TRUE`, `FALSE` are `boolean` literals
  - `‘A’`, `‘\t’`, `‘\”’`, and `‘\u03c0’` are `char` literals (escaped tab, quote characters, Unicode value for *π*)
  - `–1`, `0`, `035`, `0x1a` are `int` literals (last two are octal and hexadecimal)
  - `0.5`, `1.0`, `1E6`, `6.023E23` are `double` literals
  - “`At OSU`”, “`Hello world!`” are `String` literals
- Comments:
  - Single-line: `// some comment to end of line`
  - Multi-line: `/* comments span multiple lines */`

Common Operators in Java

`String`	`boolean`	`char`	`int`	`double`
	`!`		`++ --`	
`+`	`||`		`+   -`	`+   -`
	`&&`		`*   /   %`	`*   /`

		`<   >`
		`<= >=`
		`== !=`
			`<   >`
			`<= >=`
			`== !=`
				`<   >`

Notes:

- Compare `String` objects using the `equals()` method, not `==` or `!=`
- `&&` and `||` use *short-circuit evaluation*. Example: `boolean canPigsFly = FALSE;` we evaluate (`canPigsFly && <some Boolean expression>`). Since `canPigsFly`is `FALSE`, the second part of the expression won't be evaluated.
- The second operand of `%` (integer modulus) must be positive.
- Don't compare `double`s for equality. Instead, define a constant like so:

  ```
  final double EPSILON = 1E-6; // or some other threshold
  …     // check if Math.abs(double1 – double2) < EPSILON
  ```

Control Structures: Decision (1)

- Programs don’t always follow “straight line” execution; they “branch” based on certain conditions
- Java decision idioms: if-then-else, switch
- if-then-else idiom:

```
if (<some Boolean expression>) {
    // take some action
}
else if (<some other Boolean expression) {
    // take some other action
}
else {
    // do something else
}
```

Control Structures: Decision (2)

- Example:

```
final double OLD_DROID = 5.0, final double NEW_DROID = 9.0;
double myDroid = 8.1;
if (myDroid < OLD_DROID)
{
    System.out.println(“Antique!”);
}
else if (myDroid > NEW_DROID)
{
    System.out.println(“Very modern!”);
}
else
{
    System.out.println(“Your device: barely supported.”);
}
```

- Code prints “`Very modern!`” to the screen.
- What if `myDroid == 4.1`? `myDroid == 10.0`?

Control Structures: Decision

- Example two:

```
final double JELLY_BEAN = 4.1, final double ICE_CREAM = 4.0;
final double EPSILON = 1E-6;
double myDroid = 4.1;
if (myDroid > ICE_CREAM) {
    if (Math.abs(myDroid – ICE_CREAM) < EPSILON) {
        System.out.println(“Ice Cream Sandwich”);
    }
    else {
        System.out.println(“Jelly Bean”);
    }
}
else {
    System.out.println(“Old version”);
}
```

- Code prints “`Jelly Bean`” to screen. Note nested if-then-else, `EPSILON` usage.

Control Structures: Decision (4)

- Other idiom: switch
- Only works when comparing an `int` or `boolean` variable against a fixed set of alternatives
- Example:

```
int api = 10;
switch (api) {
    case 3:  System.out.println(“Cupcake”); break;
    case 4:  System.out.println(“Donut”); break;
    case 7:  System.out.println(“Éclair”); break;
    case 8:  System.out.println(“Froyo”); break;
    case 10: System.out.println(“Gingerbread”); break;
    case 11: System.out.println(“Honeycomb”); break;
    case 15: System.out.println(“Ice Cream Sandwich”); break;
    case 16: System.out.println(“Jelly Bean”); break;
    default: System.out.println(“Other”); break;
}
```

Control Structures: Iteration (1)

- Often, blocks of code loop while a condition holds (or fixed # of times)
- Java iteration idioms: while, do-while, for
- While loop: execute loop as long as condition is true (checked each iteration)
- Example:

```
String str = “aaaaa”;
int minLength = 10;

while (str.length() < minLength)
{
    str = str + “a”;
}

System.out.println(str);
```

- Loop executes 5 times; code terminates when `str = “aaaaaaaaaa”`
- Notice: if the length of `str` was `minLength`, the while loop would not execute

Control Structures: Iteration (2)

While Loop

```
String str = “aaaaaaaaaa”;
int minLength = 10;

while (str.length() <
minLength) {
    str = str + “a”;
}

System.out.println(str);
```

Do-While Loop

```
String str = “aaaaaaaaaa”;
int minLength = 10;

do {
    str = str + “a”;
} while (str.length() <
minLength)

System.out.println(str);
```

Unlike the while loop, the do-while loop executes at least once so long as condition is true.
The while loop prints “aaaaaaaaaa” whereas the do-while loop prints “aaaaaaaaaaa” (11 as)

Control Structures: Iteration (3)

- The for loop has the following structure:

```
for (<expression1>; <expression2>; <expression3>) {
    . . .
}
```

- Semantics:
  - `<expression1>` is loop initialization (run once)
  - `<expression2>` is loop execution condition (checked every iteration)
  - `<expression3>` is loop update (run every iteration)
- Example:

```
int i;
for (i = 0; i < 10; i++) {
    System.out.println(“i = ” + i);
}
System.out.println(“i = ” + i);
```

- What do you think this code does?

Methods and Design-by-Contract (1)

- Design your own methods to perform specific, well-defined tasks
- Each method has a *signature*:

```
public static ReturnType method(paramType1 param1, … paramTypeN
paramN) {
    // perform certain task
}
```

- Example: a method to compute area of rectangle:

```
public static double findRectArea(double length, double
width) {
     return length * width;
}
```

- Each method has a precondition and a postcondition
  - Precondition: constraints method’s caller must satisfy to call method
  - Postcondition: guarantees method provides if preconditions are met
- For our example:
  - Precondition: `length > 0.0`, `width > 0.0`
  - Postcondition: returns `length × width` (area of rectangle)

Methods and Design-by-Contract (2)

- In practice, methods are annotated via JavaDoc, e.g.,

```
/**
    Compute area of rectangle.

    @param length Length of rectangle
    @param width Width of rectangle
    @return Area of rectangle
*/
```

- Methods called from `main()` (which is `static`) need to be defined `static` too
- Some methods may not return anything (`void`)

Array Data Structure

- Array: fixed-length sequence of variable types; cannot change length at run-time

  Examples:

  ```
  final int NUMSTUDENTS = 10;
  String[] students; // Declaration
  String[] students = new String[NUMSTUDENTS];
      // Declaration and initialization
  String[] moreStudents = { “Alice”, “Bob”, “Rohit”, “Wei”};
      // Declaration and explicit initialization
  System.out.println(moreStudents.length) // Prints 4
  ```

- Enhanced `for` loop: executed for each element in array

  Example:

  ```
  for (String student: moreStudents) {
      System.out.println(student + “, ”);
  }
  ```

- Prints “`Alice, Bob, Rohit, Wei,`” to screen
- Array indices are numbered 0, …, *N*–1; watch for off-by-one errors!

  `moreStudents[0]` is “`Alice`”; `moreStudents[3]` is “`Wei`”

Two-Dimensional Arrays

- We can have two-dimensional arrays. Example:

```
final int ROWS = 3; final int COLUMNS = 3;
char[][] ticTacToe = new char[ROWS][COLUMNS]; //
declare
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLUMNS; j++)     {
        ticTacToe[i][j] = ‘_’; // Initialize to ‘blank’
    }
}
// Tic-tac-toe logic goes here (with ‘X’s, ‘O’s)
```

- `ticTacToe.length` returns number of rows; `ticTacToe[0].length` returns number of columns
- Higher-dimensional arrays are possible too

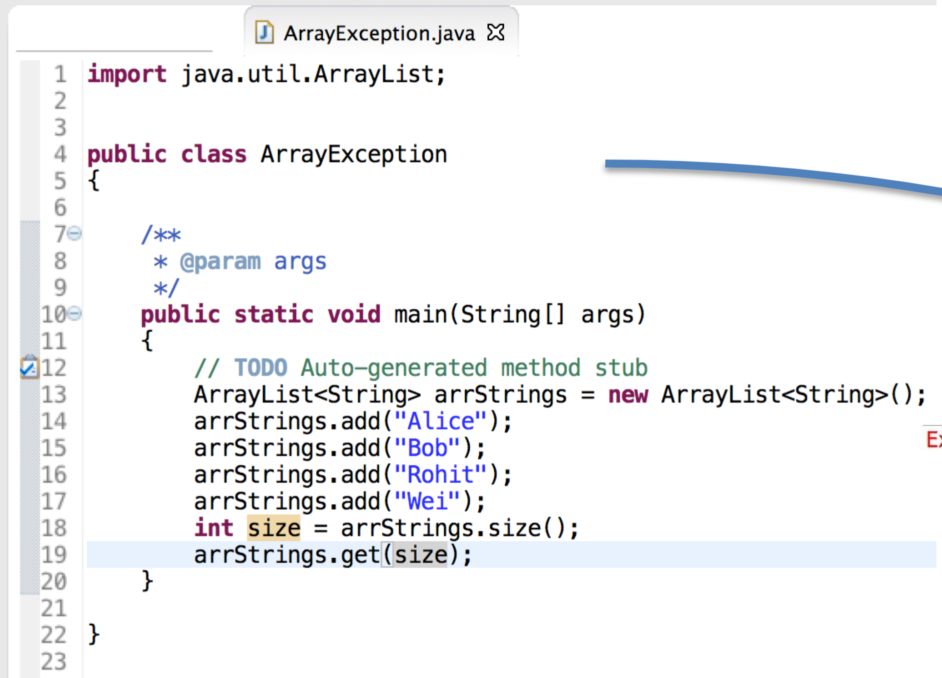
Parameterized Data Structures

- We can define data structures in terms of an arbitrary variable type (call it Item).
- `ArrayList<Item>`, a variable-length array that can be modified at run-time. Examples:

  ```
  ArrayList<String> arrStrings = new ArrayList<String>();
  ArrayList<Double> arrDoubles = new ArrayList<Double>();
  arrStrings.add(“Alice”); arrStrings.add(“Bob”); arrStrings.add(“Rohit”);
  arrStrings.add(“Wei”);
  String str = arrStrings.get(1); // str becomes “Bob”
  arrStrings.set(2, “Raj”); // “Raj” replaces “Rohit”
  System.out.println(arrStrings.size()); // prints 4
  ```
- Notice:
  - Need to call `import java.util.ArrayList;` at beginning of program
  - Off-by-one indexing: cannot call `arrStrings.get(4);`
  - *Auto-boxing:* we cannot create an `ArrayList` of `doubles`. We need to replace `double` with *wrapper class* `Double`. (Recall the “primitive data types” table)
- Other parameterized data types include `Lists`, `Sets`, `Maps`, `Stacks`, `Queues`, `Trees` (see chapters 14–16 in [1])

Exception Handling (1)

- If we had called `arrStrings.get(4)`, we would have an error condition
  - The JVM throws an `IndexOutOfBounds` exception, halts execution



ArrayException.java

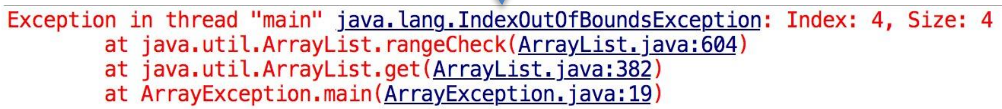
```java
import java.util.ArrayList;


public class ArrayException
{

    /**
     * @param args
     */
    public static void main(String[] args)
    {
        // TODO Auto-generated method stub
        ArrayList<String> arrStrings = new ArrayList<String>();
        arrStrings.add("Alice");
        arrStrings.add("Bob");
        arrStrings.add("Rohit");
        arrStrings.add("Wei");
        int size = arrStrings.size();
        arrStrings.get(size);
    }

}

```



```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 4, Size: 4
	at java.util.ArrayList.rangeCheck(ArrayList.java:604)
	at java.util.ArrayList.get(ArrayList.java:382)
	at ArrayException.main(ArrayException.java:19)
```

Exception Handling (2)

- We handle exceptions using the try-catch-finally structure:

```
try {
    // Code that could trigger an exception
}
catch (IndexOutOfBoundsException e) { // Or another Exception
    // Code that “responds” to exception, e.g.,
    e.printStackTrace();
}
finally {
    // Code executes regardless of whether exception occurs
}
```

- There can be many `catch` blocks for different `Exceptions`, but there is only one `try` block and one (optional) `finally` block. (See Section 7.4 in [1] for the full hierarchy of `Exceptions`)
- Exceptions always need to be caught and “reported”, especially in Android

Outline

- Getting Started
- Java: The Basics
- **Java: Object–Oriented Programming**
- Android Programming

Objects and Classes (1)

- *Classes* serve as "blueprints" that describe the states and behaviors of *objects*, which are actual "instances" of classes
- For example, a `Vehicle` class describes a motor vehicle's blueprint:
  - States: "on/off", driver in seat, fuel in tank, speed, etc.
  - Behaviors: startup, shutdown, drive "forward", shift transmission, etc.
- There are many possible `Vehicle`s, e.g., Honda Accord, Mack truck, etc. These are *instances* of the `Vehicle` blueprint
- Many Vehicle states are specific to each `Vehicle` object, e.g., on/off, driver in seat, fuel remaining. Other states are specific to the class of `Vehicle`s, not any particular `Vehicle` (e.g., keeping track of the "last" `Vehicle` ID # assigned). These correspond to *instance fields* and *static fields* in a class.
- Notice: we can operate a vehicle without knowing its implementation "under the hood". Similarly, a class makes public *instance methods* by which objects of this class can be manipulated. Other methods apply to the set of all `Vehicle`s (e.g., set min. fuel economy). These correspond to *static methods* in a class

Objects and Classes

```
public class Vehicle {
    // Instance fields (some omitted for brevity)
    private boolean isOn = false;
    private boolean isDriverInSeat = false;
    private double fuelInTank = 10.0;
    private double speed = 0.0;

    // Static fields
    private static String lastVin = “4A4AP3AU*DE999998”;

    // Instance methods (some omitted for brevity)
    public Vehicle() { … } // Constructor
    public void startUp() { … }
    public void shutOff() { … }
    public void getIsDriverInSeat() { … }  // getter, setter methods
    public void setIsDriverInSeat() { … }
    private void manageMotor() { … }    // More private methods …

    // Static methods
    public static void setVin(String newVin) { … }
}
```

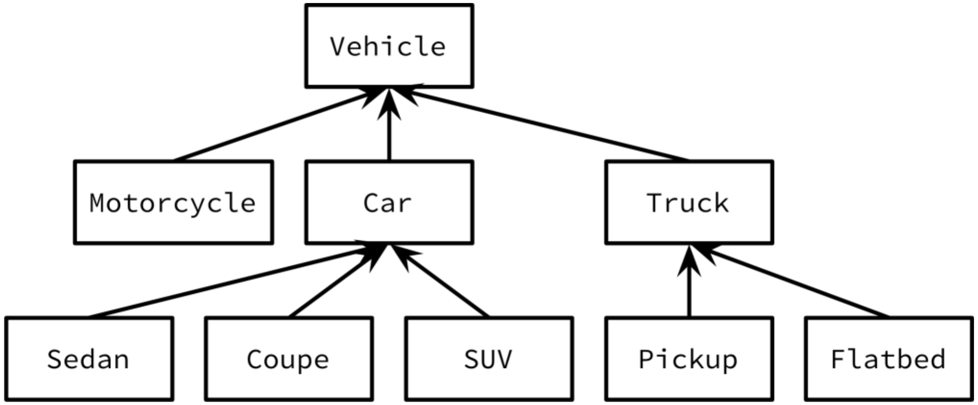
Objects and Classes (3)

- How to use the `Vehicle` class:
  - First, create a new object via constructor `Vehicle()`, e.g., `Vehicle myCar = new Vehicle();`
  - Change Vehicle states, e.g., `startUp()` or `shutOff()` the `Vehicle`
  - You can imagine other use cases
  - Mark a new `Vehicle`'s ID number (VIN) as "taken" by calling `Vehicle.setVin(…)`
  - Caveat: VINs more complex than this (simple) implementation [7]
- Notes:
  - Aliasing: If we set `Vehicle myTruck = myCar`, both `myCar` and `myTruck` "point" to the same variable. Better to perform "deep copy" of `myCar` and store the copy in `myTruck`
  - `null` reference: refers to no object, cannot invoke methods on `null`
  - Implicit parameter and the `this` reference

– Access control: `public`, `protected`, `private`

Inheritance (1)

- Types of `Vehicles`: `Motorcycle`, `Car`, `Truck`, etc. Types of `Cars`: `Sedan`, `Coupe`, `SUV`. Types of `Trucks`: `Pickup`, `Flatbed`.
- Induces inheritance hierarchy
- Subclasses inherit fields/methods from superclasses.
- Subclasses can add new fields/methods, override those of parent classes
- For example, `Motorcycle`'s `driveForward()` method differs from `Truck`'s `driveForward()` method



Inheritance

- Inheritance denoted via `extends` keyword

```
public class Vehicle {
    …
    public void driveForward
(double speed) {
        // Base class method
    }
}
```

```
public class Motorcycle
extends Vehicle {
    …
    public void driveForward
(double speed) {
        // Apply power…
    }
}
```

Inheritance (3)

```
public class Truck extends Vehicle {
    private boolean useAwd = true;
    // . . .
    public Truck(boolean useAwd) { this.useAwd = useAwd; }
    // . . .
    public void driveForward(double speed)
    {
        if (useAwd) {
            // Apply power to all wheels…
        }
        else {
            // Apply power to only front/back wheels…
        }
    }
}
```

Polymorphism

- Suppose we create `Vehicles` and invoke the `driveForward()` method:

  ```
  Vehicle vehicle = new Vehicle();
  Vehicle motorcycle = new Motorcycle();
  Truck truck1 = new Truck(true);
  Vehicle truck2 = new Truck(false);
  // Code here to start vehicles…
  vehicle.driveForward(5.0);
  motorcycle.driveForward(10.0);
  truck1.driveForward(15.0);
  truck2.driveForward(10.0);
  ```

- For `vehicle`, `Vehicle`'s `driveForward`() method is invoked
- For `motorcycle`, `Motorcycle`'s `driveForward()` method is invoked
- With `truck1` and `truck2`, `Truck`'s `driveForward()` function is invoked (with all-wheel drive for `truck1`, not for `truck2`).
- Dynamic method lookup: Java looks at objects' actual types to find which method to invoke
- Polymorphism: feature where objects of different subclasses are treated same way. (All `Vehicles` `driveForward()` regardless of (sub)class.)

The `Object` Class

- *Every* class in Java is a subclass of `Object`
- Important methods in `Object`:
  - `toString()`: Converts `Object` to a `String` representation
  - `equals()`: Compares `Object`s' contents for equality
  - `hashCode()`: Hashes the `Object` to a fixed-length `String`, useful for data structures like `HashMap`, `HashSet`
- If you create your own class, you should override `toString()` and `hashCode()`

Interfaces

- Java interfaces abstractly specify methods to be implemented
- Intuition: decouple method definitions from implementations (clean design)
- Interfaces, implementations denoted by `interface`, `implements` keywords
- Examples:

```
public interface Driveable {
    public void driveForward(double speed);
}

public class Vehicle implements Driveable {
    public void driveForward(double speed) { /* implementation */ }
}

public class Motorcycle extends Vehicle implements Driveable {
    public void driveForward(double speed) { /* implementation */ }
}
```

The `Comparable` Interface

- Comparing `Objects` is important, e.g., sorting in data structures
- The `Comparable` interface compares two `Objects`, e.g., `a` and `b`:

  ```
  public interface Comparable
  {
      int compareTo(Object otherObject);
  }
  ```
- `a.compareTo(b)` returns negative integer if `a` “comes before” `b`, `0` if `a` is the same as `b`, and a positive integer otherwise
- In your classes, you should implement `Comparable` to facilitate `Object` comparison

Object-Oriented Design Principles

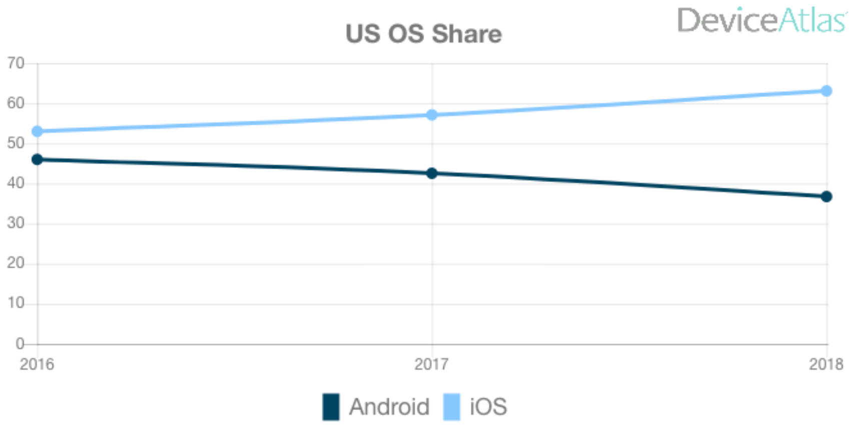
- Each class should represent a single concept
  - Don't try to fit all functionality into a single class
  - Consider a class per "noun" in problem description
  - Factor functionality into classes, interfaces, etc. that express the functionality with minimal coupling
- For software projects, start from use cases (how customers will use software: high level)
  - Then identify classes of interest
  - In each class, identify fields and methods
  - Class relationships should be identified: is-a (inheritance), has-a (aggregation), implements interface, etc.
- Packages provide class organization mechanism
  - Examples: `java.lang.*`, `java.util.*`, etc.
  - Critical for organizing large numbers of classes!
  - All classes in a package can "see" each other (scope)

Outline

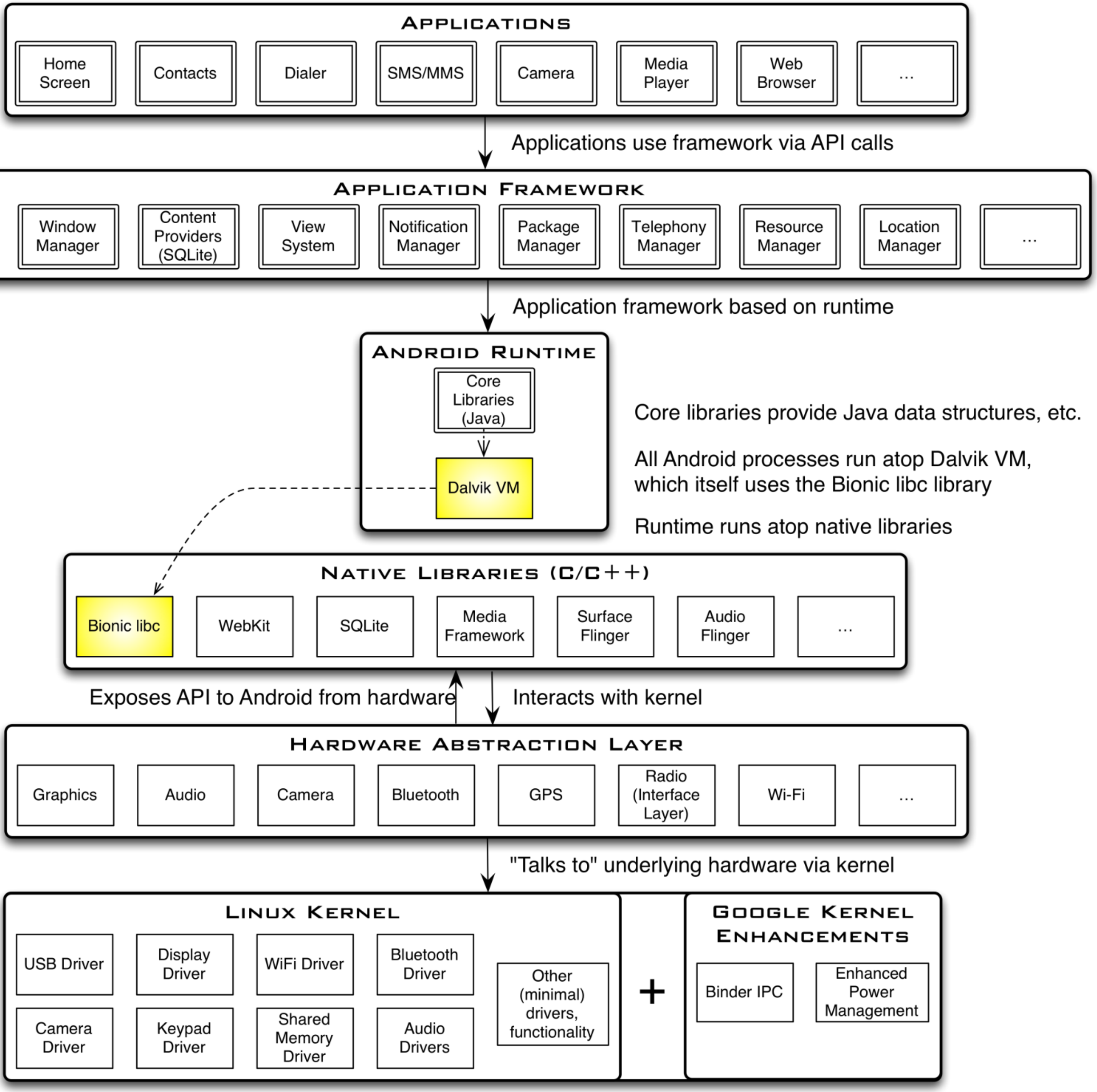
- Getting Started
- Java: The Basics
- Java: Object–Oriented Programming
- **Android Programming**

Introduction to Android

- Popular smartphone OS with Apple iOS [16]
- Developed by Open Handset Alliance, led by Google
- Over two billion Android smartphones in use worldwide [17]



Source: [16]



APPLICATIONS

Home Screen | Contacts | Dialer | SMS/MMS | Camera | Media Player | Web Browser | …

Applications use framework via API calls

APPLICATION FRAMEWORK

Activity Manager | Window Manager | Content Providers (SQLite) | View System | Notification Manager | Package Manager | Telephony Manager | Resource Manager | Location Manager | …

Application framework based on runtime

ANDROID RUNTIME

Core Libraries (Java)

Dalvik VM

Core libraries provide Java data structures, etc.

All Android processes run atop Dalvik VM, which itself uses the Bionic libc library

Runtime runs atop native libraries

NATIVE LIBRARIES (C/C++)

Bionic libc | WebKit | SQLite | Media Framework | Surface Flinger | Audio Flinger | …

Exposes API to Android from hardware

Interacts with kernel

HARDWARE ABSTRACTION LAYER

Graphics | Audio | Camera | Bluetooth | GPS | Radio (Interface Layer) | Wi-Fi | …

"Talks to" underlying hardware via kernel

LINUX KERNEL

USB Driver | Display Driver | WiFi Driver | Bluetooth Driver | Camera Driver | Keypad Driver | Shared Memory Driver | Audio Drivers | Other (minimal) drivers, functionality

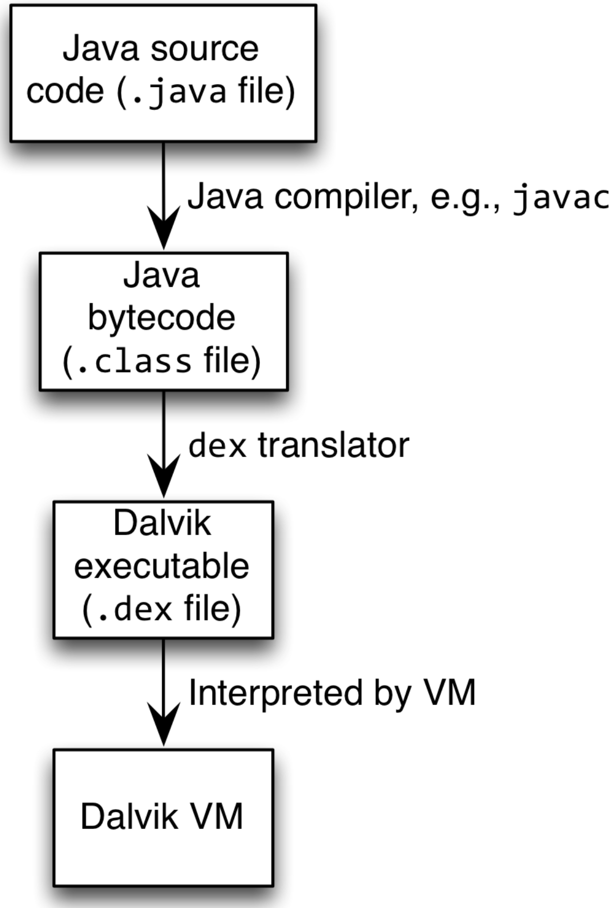
+

GOOGLE KERNEL ENHANCEMENTS

Binder IPC | Enhanced Power Management

Android Highlights (1)

- Android apps execute on Dalvik VM, a "clean-room" implementation of JVM
  - Dalvik optimized for efficient execution
  - Dalvik: register-based VM, unlike Oracle's stack-based JVM
  - Java `.class` bytecode translated to Dalvik EXecutable (DEX) bytecode, which Dalvik interprets


Java source code (.java file)
Java compiler, e.g., javac
Java bytecode (.class file)
dex translator
Dalvik executable (.dex file)
Interpreted by VM
Dalvik VM

Android Highlights (2)

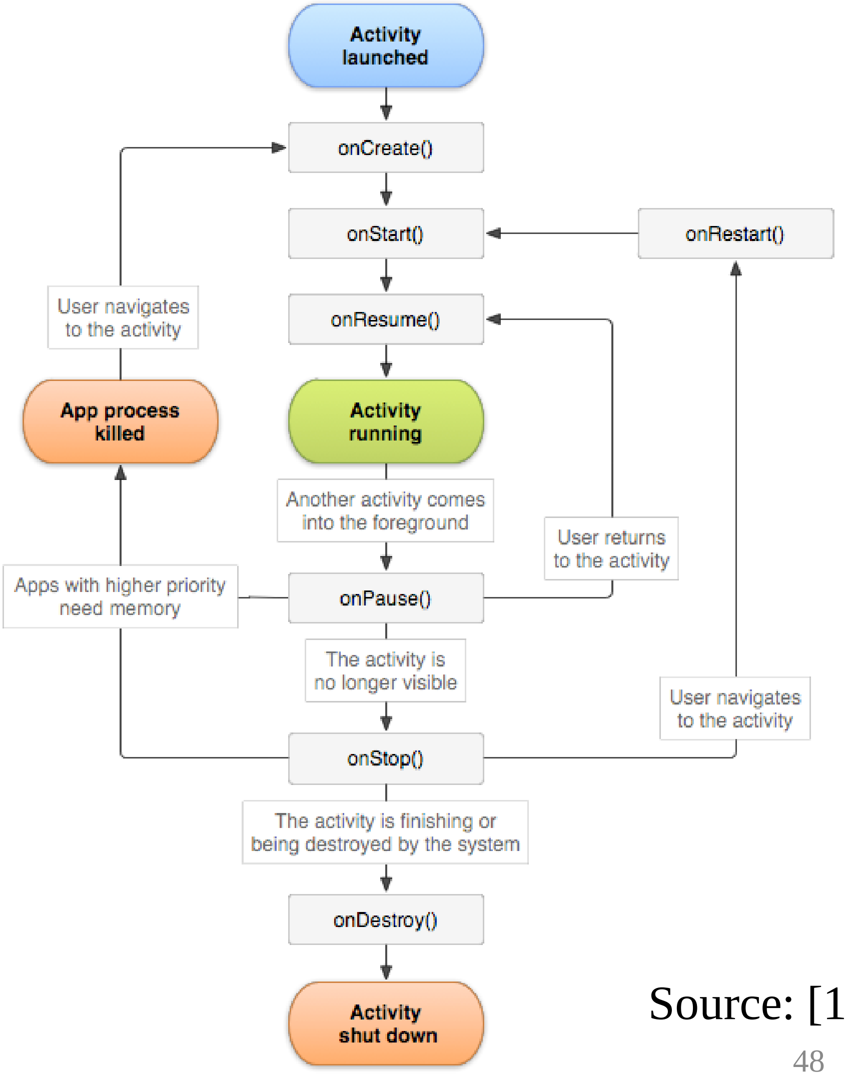
- Android apps written in Java 6+
  - Everything we’ve learned still holds
- Apps use four main components:
  - `Activity`: A “single screen” that’s visible to user
  - `Service`: Long-running background “part” of app (*not* separate process or thread)
  - `ContentProvider`: Manages app data (usually stored in database) and data access for queries
  - `BroadcastReceiver`: Component that listens for particular Android system “events”, e.g., “found wireless device”, and responds accordingly

App Manifest

- Every Android app must include an `AndroidManifest.xml` file describing functionality
- The manifest specifies:
  - App’s Activities, Services, etc.
  - Permissions requested by app
  - Minimum API required
  - Hardware features required, e.g., camera with autofocus

Activity Lifecycle

- `Activity`: key building block of Android apps
- Extend `Activity` class, override `onCreate()`, `onPause()`, `onResume()` methods
- Dalvik VM can stop any `Activity` without warning, so saving state is important!
- Activities need to be “responsive”, otherwise Android shows user “App Not Responsive” warning:
  - Place lengthy operations in `Runnable Threads`, `AsyncTasks`


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

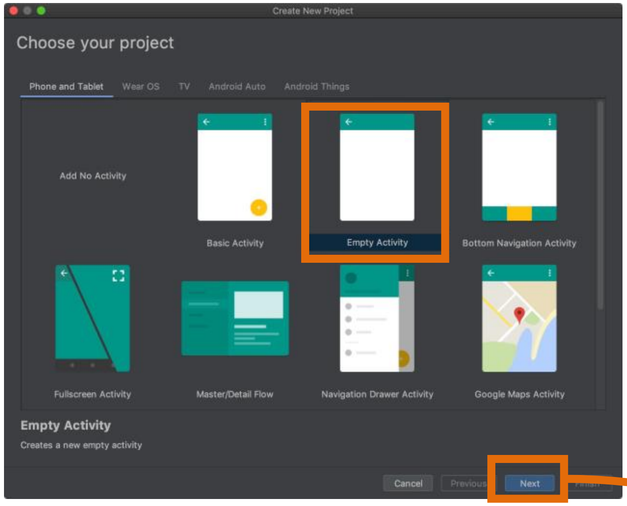
Source: [12]

App Creation Checklist

- If you own an Android device:
  - Ensure drivers are installed
  - Enable developer options on device under *Settings*, specifically *USB Debugging*
    - Android 4.2+: Go to *Settings→About phone*, press *Build number* 7 times to enable developer options
- For Android Studio:
  - Under File→*Settings→Appearance*, enable “Show tool window bars”, “Widescreen tool window layout”
  - Programs should log states via `android.util.Log`’s `Log.d(APP_TAG_STR, “debug”)`, where `APP_TAG_STR` is a `final String` tag denoting your app
  - Other commands: `Log.e()` (error); `Log.i()` (info); `Log.w()` (warning); `Log.v()` (verbose) – same parameters

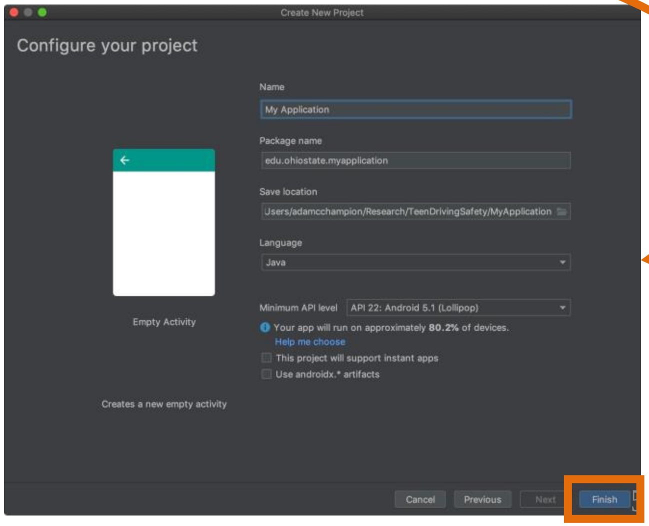
Creating Android App

- Creating Android app project (Android Studio):
  - Go to *File→New Project*
  - Select what kind of Activity to create (we'll use Empty activity)
  - Choose package name using "reverse DNS" style (*e.g.*, `edu.osu.myapp`)
  - Choose APIs for app
  - Click Finish to create "Hello World" app



Empty Activity

Creates a new empty activity



Name: My Application

Package name: edu.ohiostate.myapplication

Save location: Users/adamcchampion/Research/TeenDrivingSafety/MyApplication

Language: Java

Minimum API level: API 22: Android 5.1 (Lollipop)

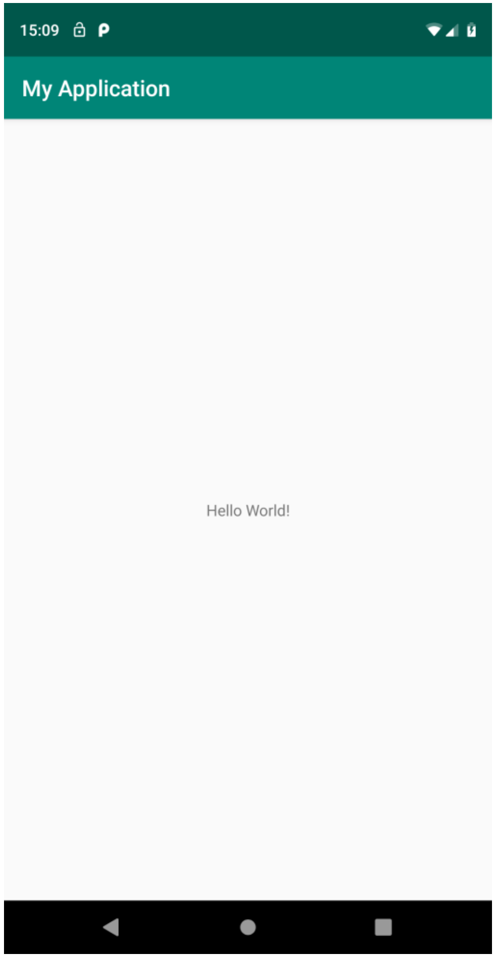
Your app will run on approximately **80.2%** of devices.

□ This project will support instant apps

□ Use androidx.* artifacts

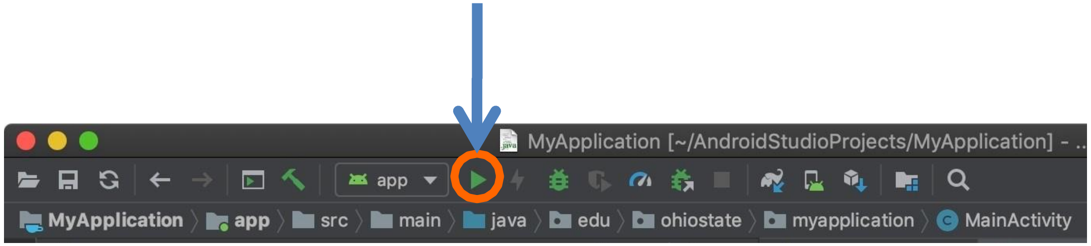
Deploying the App

- Two choices for deployment:
  - Real Android device
  - Android virtual device
- Plug in your real device; otherwise, create an Android virtual device
- Emulator is slow. Try Intel accelerated version, or perhaps `http://www.genymotion.com/`
- Run the app: press “Run” button in toolbar



My Application

Hello World!


MyApplication [~/AndroidStudioProjects/MyApplication] - ..
app
MyApplication 〉 app 〉 src 〉 main 〉 java 〉 edu 〉 ohiostate 〉 myapplication 〉 MainActivity

Underlying Source Code

src/…/MainActivity.java

```
package edu.osu.helloandroid;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Underlying GUI Code

res/layout/activity_main.xml

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

– `RelativeLayouts` are quite complicated. See [13] for details

The App Manifest

AndroidManifest.xml

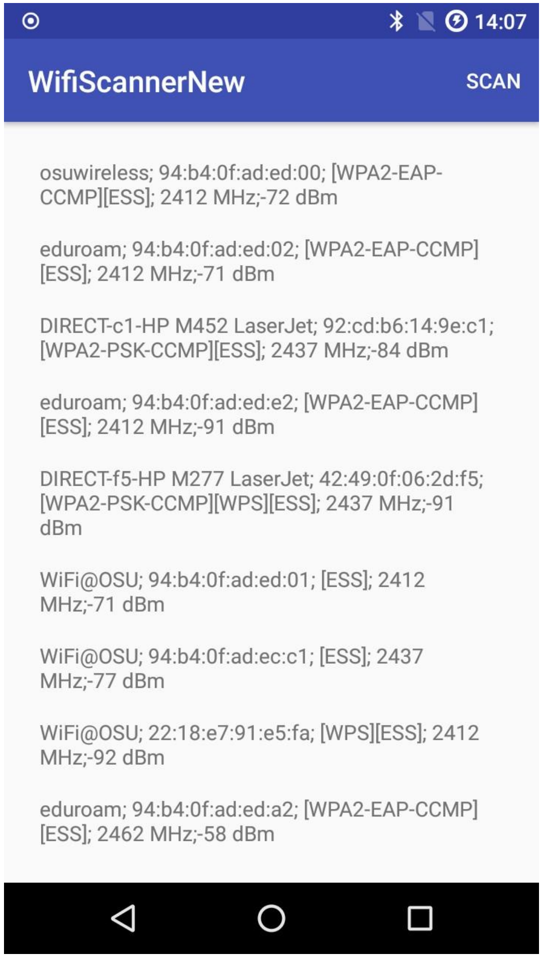
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.osu.helloandroid"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="edu.osu.helloandroid.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

A More Interesting App

- We'll now examine an app with more features: WiFi Scanner (code on class website)
- Press a button, scan for Wi-Fi access points (APs), display them
- Architecture: Activity creates single Fragment with app logic (flexibility)



osuwireless; 94:b4:0f:ad:ed:00; [WPA2-EAP-CCMP][ESS]; 2412 MHz;-72 dBm

eduroam; 94:b4:0f:ad:ed:02; [WPA2-EAP-CCMP][ESS]; 2412 MHz;-71 dBm

DIRECT-c1-HP M452 LaserJet; 92:cd:b6:14:9e:c1; [WPA2-PSK-CCMP][ESS]; 2437 MHz;-84 dBm

eduroam; 94:b4:0f:ad:ed:e2; [WPA2-EAP-CCMP][ESS]; 2412 MHz;-91 dBm

DIRECT-f5-HP M277 LaserJet; 42:49:0f:06:2d:f5; [WPA2-PSK-CCMP][WPS][ESS]; 2437 MHz;-91 dBm

WiFi@OSU; 94:b4:0f:ad:ed:01; [ESS]; 2412 MHz;-71 dBm

WiFi@OSU; 94:b4:0f:ad:ec:c1; [ESS]; 2437 MHz;-77 dBm

WiFi@OSU; 22:18:e7:91:e5:fa; [WPS][ESS]; 2412 MHz;-92 dBm

eduroam; 94:b4:0f:ad:ed:a2; [WPA2-EAP-CCMP][ESS]; 2462 MHz;-58 dBm

Underlying Source Code (1)

```
// WifiScanActivity.java
public class WifiScanActivity extends SingleFragmentActivity {
    @Override
    protected Fragment createFragment() {return new WifiScanFragment(); }
}
// WifiScanFragment.java. Uses RecyclerView to display dynamic list of Wi-Fi ScanResults.
@Override
public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_wifi_scan, container, false);
    mScanResultRecyclerView = (RecyclerView) v.findViewById(R.id.scan_result_recyclerview);
    mScanResultAdapter = new ScanResultAdapter(mScanResultList);
    mScanResultRecyclerView.setAdapter(mScanResultAdapter);
    mScanResultRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));

    setupWifi();
    mIntentFilter = new IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);

    setHasOptionsMenu(true); setRetainInstance(true);

    return v;
}

private void setupWifi() {
    try {
        Context context = getActivity().getApplicationContext();
        if (context != null) {
            mWifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);
        }
    } catch (NullPointerException npe) {
        Log.e(TAG, "Error setting up Wi-Fi");
    }
}
```

Underlying Source Code (2)

- Get system `WifiManager`
- Register Broadcast Receiver to listen for `WifiManager`’s “finished scan” system event (expressed as `Intent` `WifiManager.SCAN_RESULTS_AVAILABLE_ACTION`)
- Unregister Broadcast Receiver when leaving `Fragment`

```
@Override
public void onResume() { // . . .
  super.onResume(); // . . .
  SharedPreferences sharedPreferences =
    PreferenceManager.getDefaultSharedPreferences(getActivity().getApplicationContext());
    boolean hideDialog =
      sharedPreferences.getBoolean(getResources().getString(R.string.suppress_dialog_key), false);
    if (!hideDialog) { // Show user dialog asking them to accept permission request
        FragmentManager fm = getActivity().getSupportFragmentManager();
        DialogFragment fragment = new NoticeDialogFragment();
        fragment.show(fm, "info_dialog"); }
    getActivity().registerReceiver(mReceiver, mIntentFilter);
}

@Override
public void onPause() {
    super.onPause();
    getActivity().unregisterReceiver(mReceiver);
}
```

Underlying Source Code (3)

- Register menu-item listener to perform Wi-Fi scan
- Get user permission first for “coarse” location (required in Android 6+)

```
// WifiScanFragment.java
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    super.onCreateOptionsMenu(menu, inflater);
    inflater.inflate(R.menu.menu, menu); }

public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.menu_scan:
      if (!hasLocationPermission()) { requestLocationPermission(); }
      else { doWifiScan(); }
    return true; }
  return false; }

private void requestLocationPermission() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    if (!hasLocationPermission()) {
      requestPermissions(new String[]{Manifest.permission.ACCESS_COARSE_LOCATION}, PERMISSION_REQUEST_LOCATION); }}}

public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, int[] grantResults) {
    if (requestCode == PERMISSION_REQUEST_LOCATION) {
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) { doWifiScan(); } else { // Error } }}}
```

The Broadcast Receiver

```
// WifiScanFragment.java
private final BroadcastReceiver mReceiver = new BroadcastReceiver()
{
    // Override onReceive() method to implement our custom logic.
    @Override
    public void onReceive(Context context, Intent intent)
    {
        // Get the Intent action.
        String action = intent.getAction();

        // If the WiFi scan results are ready, iterate through them  and
        // record the WiFi APs' SSIDs, BSSIDs, WiFi capabilities,  radio
        // frequency, and signal strength (in dBm).
        if (WifiManager.SCAN_RESULTS_AVAILABLE_ACTION.equals(action))
        {
            // Ensure WifiManager is not null first.
            if (mWifiManager == null) { setupWifi();  }

            List<ScanResult> scanResults = mWifiManager.getScanResults();
            mScanResultList.addAll(scanResults);
            mScanResultAdapter.notifyDataSetChanged();
        }
    }
};
```

User Interface

Updating UI in code

- Two inner classes handle `RecyclerView` items:
  - `ScanResultAdapter` (extends `RecyclerView.Adapter<ScanResultHolder>`)
  - `ScanResultHolder` (extends `RecyclerView.ViewHolder`)
- See code, Big Nerd Ranch (Chapter 8) for details

UI Layout (XML)

```
<!-- fragment_wifi_scan.xml
     (for the RecyclerView fragment) -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent” >
  <android.support.v7.widget.RecyclerView
    android:id="@+id/scan_result_recyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</LinearLayout>

<!-- item_wifi_scan.xml
     (for each RecyclerView item) -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content” >

    <TextView
        android:id="@+id/scan_result_textview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="TextView"/>
</LinearLayout>
```

Android Programming Notes

- Android apps have multiple points of entry: no `main()` method
  - Cannot “sleep” in Android
  - During each entrance, certain `Objects` may be `null`
  - Defensive programming is very useful to avoid crashes, e.g.,
    `if (!(myObj == null)) { // do something }`
- Java concurrency techniques are required
  - Don’t block the “main” thread in Activities
  - Implement long-running tasks such as network connections asynchronously, e.g., as `AsyncTasks`
  - Recommendation: read [4]; chapter 20 [10]; [11]
- Logging state via `android.util.Log` throughout app is essential when debugging (finding root causes)
- Better to have “too many” permissions than too few
  - Otherwise, app crashes due to security exceptions!
  - Remove “unnecessary” permissions before releasing app to public
- Event handling in Android GUIs entails many listener `Objects`

Concurrency: `Threads` (1)

- `Thread`: program unit (within process) executing independently
- Basic idea: create class that implements `Runnable` interface
  - `Runnable` has one method, `run()`, that has code to execute
  - Example:

    ```
    public class OurRunnable implements Runnable {
        public void run() {
            // run code
        }
    }
    ```
- Create a `Thread` object from `Runnable` and `start()` `Thread`, e.g.,

  ```
  Runnable r = new OurRunnable();
  Thread t = new Thread(r);
  t.start();
  ```
- Problems: cumbersome, does not reuse `Thread` code

Concurrency: `Threads`

- Easier approach: anonymous inner classes, e.g.,

  ```
  Thread t = new Thread(new Runnable(
      {
          public void run()
          {
              // code to run
          }
      });
  t.start();
  ```
- Idiom essential for *one-time* network connections in Activities
- However, `Thread`s can be difficult to synchronize, especially with UI thread in `Activity`, `Fragment`; `AsyncTask`s more suitable

Concurrency: AsyncTasks

- AsyncTask encapsulates asynchronous task that interacts with UI thread in Activity:

```
public class AsyncTask<ParamsType, ProgressType, ResultType> {
    protected Result doInBackground(ParamType param) {
        // code to run in background
        publishProgress(ProgressType progress); // UI
        …
        return Result;
    }

    protected void onProgressUpdate(ProgressType progress) {
        // invoke method in Activity to update UI
    }
}
```

- Extend AsyncTask with your own class
- Documentation at http://developer.android.com

Thank You