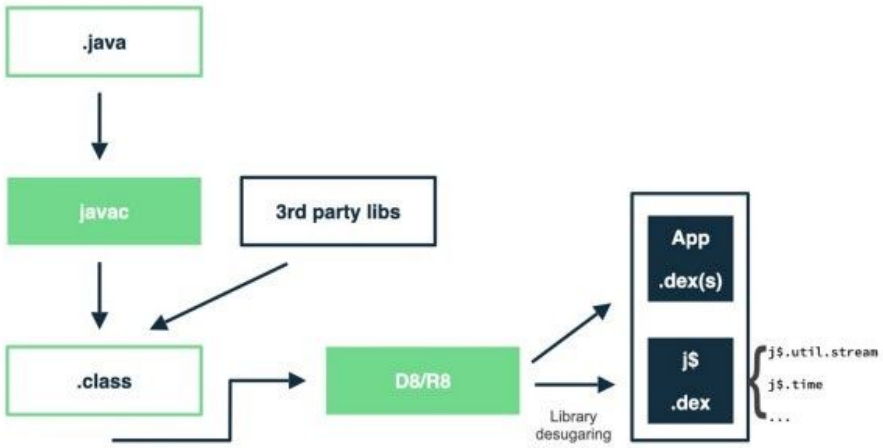
1. Using Java 8+ APIs on Android Devices
    a. A lot of crashes on devices with the version < 26.
    b. Why? The reason is just the **device didn't ship the classes necessary for the newer Java API**.
    c. Android 11 supports a number of APIs from newer OpenJDK. (up to 13)
    d. **Android Gradle Plugin 4.0.0 and newer** one can enable us to use tons of new APIs from newer OpenJDK.
    e. Developers can use some of these newer Java APIs in Android 11 for older devices (Basically through **backporting, and desugaring** on older devices where the Android platform doesn't have the APIs on the runtime)



2. D8/R8 Desugaring
    a. The Android Gradle Plugin 4.0
        i. Built-in support for using certain Java language APIs and 3rd-party libraries
        ii. With this, almost all versions of Android are supported.
    b. Desugaring performed by D8/R8
        i. Java Source code to Java bytecode [By Java Compiler]
        ii. Implementation of the new APIs [Bytecode Transformations]
        iii. Bytecode to Dex Library (The necessary Java 8 runtime code is separate dex library)
        iv. This process is called **desugaring!** (a set of Java 8 APIs enabled on all existing devices, except parallel (supported from 21)
    c. How to set up the desugaring in build.gradle.
        i. (Don't forget to version up your AGP to 4.0 and above!)
        ii. If minSDK is 20 or lower, declare **multiDexEnabled as true**.

```
android {
    defaultConfig {
        multiDexEnabled true
    }

    compileOptions {
        coreLibraryDesugaringEnabled true
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    dependencies {
        coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.0.5'
    }
}
```

3. Added newer APIs
    a. java.time - based on *JodaTime* library / no concurrency issues with immutability
        i. If no need timezone data and Date and Time object
            1. LocalDate
            2. LocalTime

```
// Current date
LocalDate date = LocalDate.now();

// Current time
LocalTime time = LocalTime.now();

// Date of Android 11 beta launch
LocalDate date2 = LocalDate.of(2020, 6, 10); //yyyy,MM,dd
```

        ii. If requiring time zones
            1. ZoneDateTime - immutable representation of **Datetime**
                a. Hold **LocalDateTime, ZoneId, ZoneOffset**
                b. Useful when we need to store date and time (not relying on the device or app)
                c. **ZoneOffsets** to calculate the difference between current one and UTC

2. OffsetDateTime - keeps the offset from Greenwich/UTC instead of ZoneId.

```
ZonedDateTime currentDate = ZonedDateTime.now();
System.out.println("the current zone is "+ currentDate.getZone());
//the current zone is America/Los_Angeles

ZoneId anotherZone = ZoneId.of("Europe/Istanbul");
ZonedDateTime currentDateInIstanbul =
currentZone.withZoneSameInstant(anotherZone);
//2020-06-08T23:50:00.159+03:00[Europe/Istanbul]
```

iii. Summarize
   1. ZoneDateTime: good for **displaying timezone-sensitive datetime data**
   2. OffsetDateTime: best for **storing datetime data** to a database / other use cases where the data needs to be **serialized**

iv. Period and Duration

```
LocalDate now = LocalDate.now();
LocalDate androidBeta = LocalDate.of(2020, 6, 10);
Period period = Period.between(androidBeta, now);
```

b. java.util.stream
   i. It allows us to perform **functional-style operations on collections**.
   ii. Intermediate operations / Terminal operations
   iii. It may perform better when working on large data sources.

```
List<Person> list = Arrays.asList(personArray);
Stream<Person> stream1 = list.stream();

//or
Stream<Person> stream2 = Stream.of(personArray);

//or
Stream.Builder<Person> streamBuilder = Stream.builder();
for (Person p: personArray){
    streamBuilder.accept(p);
}
Stream<Person> stream3 = streamBuilder.build();
```

c. Other Java APIs
    i. Map, Collection, Comparator
        1. Can be simple to handle with Lambda express

```
import static java.util.Comparator.comparing;

comparing(Person::getFirstname).thenComparing(Person::getSurname);
```

    ii. Optional and its counterpart for primitive types
        1. Optional<Object>
        2. OptionalInt, OptionalLong, OptionalDouble
        3. Robust to null as well as Kotlin does.

```
Optional<String> opt = Optional.of("Java 8");
if (opt.isPresent()){ //true
   //...
}

opt = Optional.ofNullable(null);
if (opt.isEmpty()){ //true
   //…
}

opt.ifPresent(str -> System.out.println(str)); //false, doesn't print
```

    iii. Atomic classes in java.util.concurrent.atomic package
        1. AtomicInteger
        2. AtomicLong
        3. AtomicReference
    iv. ConcurrentHashMap (Thread-safe alternative to HashMap)
        1. ConcurrentHashMap on Android 21 and 22 had a bug.
        2. Fixed version is in the desugared library
        3. Allows any number of thread to perform operations
        4. **Performing thread must lock the particular segment** where the data is being processed when updating or inserting data.
        5. Due to the synchronization, performance is slower than a HashMap.
d. Full listups
    i. https://developer.android.com/studio/write/java8-support-table