```kotlin
package net.zeevox.nearow.input

import ...

/**
* [DataCollectionService] is a foreground service that receives sensor and location updates and
* handles the lifecycle of the tracking process
*/
class DataCollectionService : Service(), SensorEventListener {

   /** [SensorManager] is a gateway to access device's hardware sensors */
   private lateinit var mSensorManager: SensorManager

   /**
    * [NotificationManagerCompat] is a wrapping library around [NotificationManager] Used to push
    * foreground service notifications, which are necessary to prevent the service from getting
    * killed
    */
   private lateinit var mNotificationManager: NotificationManagerCompat

   /**
    * Instance of [DataProcessor] to which we pass sensor and location updates for number-crunching
    */
   private lateinit var mDataProcessor: DataProcessor

   /** A direct reference to the [DataProcessor] currently in use by the [DataCollectionService] */
   val dataProcessor: DataProcessor
       get() = mDataProcessor

   /**
    * Whether this instance of the [DataCollectionService] is currently running as a foreground
    * service
    */
   private var inForeground: Boolean = false

   /** Contains parameters used by [FusedLocationProviderClient]. */
   private lateinit var mLocationRequest: LocationRequest

   /** Provides access to the Fused Location Provider API. */
   private lateinit var mFusedLocationClient: FusedLocationProviderClient

   /** Callback for changes in location. */
   private lateinit var mLocationCallback: LocationCallback

   companion object {
       const val NOTIFICATION_ID = 7652863

       // 20,000 us => ~50Hz sampling
       const val ACCELEROMETER_SAMPLING_DELAY = 20000

       /**
        * The desired interval for location updates. Inexact. Updates may be more or less frequent.
        */
       private const val UPDATE_INTERVAL_IN_MILLISECONDS: Long = 1000L

       /**
        * The fastest rate for active location updates. Updates will never be more frequent than
        * this value.
        */
       private const val FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS: Long = 0L

       /** Logcat tag used for debugging */
       private val TAG = DataCollectionService::class.java.simpleName
   }
```

```kotlin
/** https://developer.android.com/guide/components/bound-services#Binder */
private val binder = LocalBinder()

/**
 * Class used for the client Binder. Because we know this service always runs in the same
 * process as its clients, we don't need to deal with IPC.
 */
inner class LocalBinder : Binder() {
    // Return this instance of LocalService so clients can call public methods
    fun getService(): DataCollectionService = this@DataCollectionService
}

override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    return START_STICKY
}

override fun onCreate() {
    super.onCreate()

    Log.i(TAG, "Starting Nero data collection service")

    mNotificationManager = NotificationManagerCompat.from(this)

    // start the data processor before registering sensor and GPS
    // listeners so that it is ready to receive values as soon as
    // they start coming in.
    mDataProcessor =
        DataProcessor(applicationContext).also {
            // physical sensor data is not permission-protected so no need to check
            registerSensorListener()

            initGpsClient()

            // measuring GPS is neither always needed (e.g. erg) nor permitted by user
            // check that access has been granted to the user's geolocation before starting gps
            // collection
            if (isGpsPermissionGranted()) enableGps()
        }

    startService(Intent(applicationContext, DataCollectionService::class.java))
    startForeground()
}

/**
 * Called when a client comes to the foreground and binds with this service. The service should
 * cease to be a foreground service when that happens.
 */
override fun onBind(intent: Intent?): IBinder {
    Log.i(TAG, "Client bound to service")
    stopForeground()
    return binder
}

/**
 * Called when a client comes to the foreground and binds with this service. The service should
 * cease to be a foreground service when that happens.
 */
override fun onRebind(intent: Intent?) {
    Log.i(TAG, "Client rebound to service")
    stopForeground()
    super.onRebind(intent)
}
```

```kotlin
/**
 * Called when the last client unbinds from this service. If a track is being recorded,
 * make this service a foreground service.
 */
override fun onUnbind(intent: Intent?): Boolean {
    Log.i(TAG, "Last client unbound from service")

    if (mDataProcessor.isRecording) startForeground() else stopForeground()
    return true
}

/**
 * Switch the [DataCollectionService] to a foreground service so that sensor and location
 * updates can continue to be processed even though the application UI has gone out of view
 */
private fun startForeground() {
    Log.i(TAG, "Switching to foreground service")
    startForeground(NOTIFICATION_ID, NotificationUtils.getForegroundServiceNotification(this))

    // Pushing notifications on main thread is warned against by StrictMode
    CoroutineScope(Dispatchers.Default).launch {
        mNotificationManager.notify(
            NOTIFICATION_ID,
            NotificationUtils.getForegroundServiceNotification(this@DataCollectionService))
    }

    inForeground = true
}

/** Stop being a foreground service if the GUI comes back into view. */
private fun stopForeground() {
    Log.i(TAG, "Cancelling foreground service")
    stopForeground(true)
    inForeground = false
}

fun setDataUpdateListener(listener: DataProcessor.DataUpdateListener) =
    mDataProcessor.setListener(listener)

private fun registerSensorListener() {
    CoroutineScope(Dispatchers.IO).launch {
        mSensorManager = getSystemService(AppCompatActivity.SENSOR_SERVICE) as SensorManager
        mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION)?.also { accelerometer
            ->
            mSensorManager.registerListener(
                this@DataCollectionService, accelerometer, ACCELEROMETER_SAMPLING_DELAY)
        }
    }
}

private fun isGpsPermissionGranted(): Boolean {
    return ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) ==
            PackageManager.PERMISSION_GRANTED
}
```

```kotlin
    /**
                                                                                                         *
https://github.com/android/location-samples/blob/main/LocationUpdatesForegroundService/app/src/main/jav
a/com/google/android/gms/location/sample/locationupdatesforegroundservice/LocationUpdatesService.java
     */

    private fun initGpsClient() {
        mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)

        mLocationCallback =
            object : LocationCallback() {
                override fun onLocationResult(locationResult: LocationResult) {
                    super.onLocationResult(locationResult)
                    mDataProcessor.addGpsReading(locationResult.lastLocation)
                }
            }

        createLocationRequest()
    }

    fun enableGps() {
        try {
            mFusedLocationClient.requestLocationUpdates(
                mLocationRequest, mLocationCallback, Looper.getMainLooper())
        } catch (unlikely: SecurityException) {
            Log.e(TAG, "Lost location permission. Could not request updates.", unlikely)
        }
    }

    private fun createLocationRequest() {
        mLocationRequest =
            LocationRequest.create().apply {
                interval = UPDATE_INTERVAL_IN_MILLISECONDS
                fastestInterval = FASTEST_UPDATE_INTERVAL_IN_MILLISECONDS
                priority = LocationRequest.PRIORITY_HIGH_ACCURACY
            }
    }

    /** Stop requesting location updates */
    fun disableGps() {
        Log.i(TAG, "Requesting GPS location updates to stop")
        try {
            mFusedLocationClient.removeLocationUpdates(mLocationCallback)
        } catch (unlikely: SecurityException) {
            Log.e(TAG, "Lost location permission. Could not remove updates.", unlikely)
        }
    }

    /**
     * Called by the system to notify a Service that it is no longer used and is being removed. The
     * service should clean up any resources it holds (threads, registered receivers, etc) at this
     * point. Upon return, there will be no more calls in to this Service object and it is
     * effectively dead.
     */
    override fun onDestroy() {
        disableGps()
        super.onDestroy()
    }
    override fun onSensorChanged(event: SensorEvent?) {
        if (event == null) return

        when (event.sensor.type) {
            Sensor.TYPE_LINEAR_ACCELERATION -> mDataProcessor.addAccelerometerReading(event.values)
        }
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        // TODO accuracy handling?
        Log.w(TAG, "Unhandled ${sensor?.name} sensor accuracy change to $accuracy")
    }

    private class NotificationUtils private constructor() {
        companion object {
            private const val CHANNEL_ID = "tracking_channel"

            @RequiresApi(Build.VERSION_CODES.O)
            internal fun createServiceNotificationChannel(context: Context) {
                val notificationManager = NotificationManagerCompat.from(context)
                notificationManager.createNotificationChannel(
                    NotificationChannel(
                            CHANNEL_ID,
                            context.getString(R.string.notification_channel_tracking_service),
                            NotificationManager.IMPORTANCE_MIN)
                        .apply {
                            enableLights(false)
                            setSound(null, null)
                            enableVibration(false)
                            vibrationPattern = longArrayOf(0L)
                            setShowBadge(false)
                        })
            }

            internal fun getForegroundServiceNotification(context: Context): Notification {

                val notificationBuilder =
                    NotificationCompat.Builder(context, CHANNEL_ID)
                        .setAutoCancel(true)
                        .setDefaults(Notification.DEFAULT_ALL)
                        .setContentTitle(context.resources.getString(R.string.app_name))
                        .setContentText(
                            context.getString(R.string.notification_background_service_running))
                        .setWhen(System.currentTimeMillis())
                        .setOngoing(true)
                        .setVibrate(longArrayOf(0L))
                        .setSound(null)
                        .setSmallIcon(R.mipmap.ic_launcher_round)

                // Notifications channel required for Android 8.0+
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                    createServiceNotificationChannel(context)
                    notificationBuilder.setChannelId(CHANNEL_ID)
                }

                return notificationBuilder.build()
            }
        }
    }
}
```