ОСОБЕНОСТИ НА РАБОТАТА СЪС СЕНЗОРИ И С УСЛУГАТА ЗА МЕСТОПОЛОЖЕНИЕ В ОС ANDROID

Д-р инж.Мартин Пъшев Иванов,
главен асистент, НБУ –департамент „Информатика”,
e-mail: mivanov@nbu.bg

Анотация: Представеният материал разглежда фундаменталните предпоставки и техники за работа със сензорите и за достъп до услугата за местоположение в ОС Android. Представени са принципите на действие и особеностите на техническите компонени, систематизирано е разнообразието им, описани са библиотеките, съдържащи необходимите класове и интерфейси за създаване на софтуерни приложения, ползващи тези възможности на апаратната част на мобилните устройства. Изложението е илюстрирано с фрагменти от програмен код, написан на Java, които изясняват употребата на софтуерните средства в някои типични ситуации.

Ключови думи: *OS Android, сензори, android.hardware, android.location, GPS, услуга за месоположение*

1. Въведение. Мобилни устройства и сензори.

Мобилните устройства отдавна са се превърнали от средство за изпълнение на някои прости комуникационни операции в инструмент за многофункционална употреба. Съвремените смартфони и таблети, управлявани от ОС Android, разполагат с множество сензори (датчици), които реагират на преместване, на пространствена ориентация, на различни параметри на околната среда (магнитно поле, температура на въздуха, атмосферно налягане) и т.н. Наличието на такива сензори превръща мобилното устройство в изключително полезен, при това достъпен за потребителя инструмент. Упоребата на сензори става както в развлекателния софтуер, така и в приложения с много по-сериозна и професионална насоченост.

Започвайки от Android 1.5 (API Level 3) системата разполага с един разширяващ се стандартен набор от сензори и свързания с него програмен интерфейс. В Android 2.3 (API Level 9) са добавени нови сензори и средства, както необходимите за използването им инструменти за разработване. Към стандартните сензори са добавени акселерометър, жироскоп, магнитометър (компас), сензор за осветеност, сензор за близост, сензор за относителна влажност на въздуха и сензор за атмосферно налягане. Инструментите, добавени в API Level 9, включват методи за получаване на ротационните матрици, алтернативни представяния на ротацията и „синтетични” сензори. Тези инструменти предоставят на разработчиците разширен набор от възможности за физическа навигация, управление на игри, моделиране на виртуална реалност и много други полезни приложения.

Сензорът представлява елекронно устройство, което е в състояние да възприема физически величини и да ги преобразува в електрически сигнали, които могат да бъдат непосредствено измервани, предавани и кодирани. Всеки сензор има свой принцип на действие, както и специфични технически параметри. Сензорите са в състояние да осигурят изходните (необработени) данни с необходимата прецизност и точност, да ги представят пред потребителя по подходящ начин или да ги предадат на ползващите ги софтуерни приложения.

2. **Видове сензори, поддържани от ОС Android**.

Android поддържа разнообразни сензори, които обикновено се систематизират в няколко по-общи (и донякъде условни) групи:

- **Сензори за преместване** (Motion sensors) – тези сензори измерват линейни и ъглови ускорения. В тази категория влизат акселерометри, сензори за гравитация, жироскопи, сензори за ротация и др.
- **Сензори, измерващи параметри на средата** (Environmental sensors) – измерват разнообразни параметри на средата като например температура и влажност на въздуха, атмосферно налягане и т.н. Тук влизат различните барометри, термометри, фотометри и др.
- **Сензори за местоположение** (Position sensors) – измерват физическото местоположение на устройстото в географски координати. Включват сензори за ориентация и магнитометри.

Друга класификация на сензорите ги разделя на **хардуерно-базирани** и **софтуерно-базирани**. Хардуерно-базираните сензори са физически конструктивни компоненти, които са вградени в апаратната част на устройстовото. Те измерват непосредствено конкретни физически величини – такива сензори са сензорите за ускорение, на геомагнитното поле, за ъглови изменения, за осветеност и др. Софтуерно-базираните сензори не са технически реализирани, а всъщност извличат данни от един или повече хардуерно-базирани сензори и въз основа на техните стойности изработват собствен резултат. Такива сензори носят още имената „композитни", „виртуални" или „синтетични". Такива са например сензорът за гравитация, за линейно ускорение и др. От гледна точка на съставянето на програмния код хардуерно-базираните и софтуерно-базираните сензори се интерпретират по един и същ начин.

В зависимост от конструктивното изпълнение на мобилното устройство, броят и видът на включените в него сензори може да е различен, но обикновено акселерометърът е задължителен компонент. В някои случаи устройствата разполагат с няколко сензора от един и същи тип.

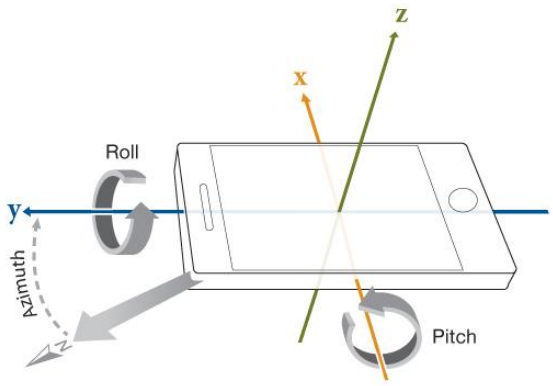
3. **Основни технически параметри на сензорите – обхват, точност, прецизност, интервал на отчитане.**

Всеки сензор е конструиран така, че изработваният от него електрически сигнал да може да бъде преобразуван в някаква скала мерни единици. Физическите величини, измервани от сензорите са различни, поради което и скалите, в които те се представят са различни. Като обхват на сензора се определя разликата между максималната и минималната стойност на физическата величина, която може да бъде измерена от сензора. Обхватът се измерва в същите мерни единици, в които се измерва и съответната физическа величина. Точността на сензора е мярка за фактическото отклонение на измерената стойност от действителната стойност на физическия параметър. Прецизността и резолюцията на сензора са твърде близки като понятия. Те се определят от минималната промяна в стойността на измервания параметър, която може да бъде отчетена. Доколкото данните от някои сензори постъпват като

непрекъснат поток от стойности във времето, то друг характерен показател е т.нар. „честота на отчитане”. Нейната стойност е реципрочна на продължителността на интервала (периода) от време между две последователни отчитания и се измерва в херци (Hz). Като показател, характеризиращ сензора, се използва и продължителността на самия времеви интервал (delay) между измерванията. Съществени параметри за мобилните устройства, захранвани от батерия, са и консумираният ток или електрическа мощност от сензора.

4. Координатни системи и скали в сензорите.

Със сензорите, отчитащи пространствено-насочени характеристики – местоположение и премествания е свързана координатна система, чиито оси са ориентирани спрямо мобилното устройство както е показано на фигурата. Скалите по всяка от осите са в единици, специфични за съответния сензор и за настройката му.



Фиг. 1. Координатна система, използвана от сензорите в мобилното устройство.

5. Кратко представяне на някои типични сензори в Android.

Акселерометърът измерва ускорението по всяка от кординатните оси, с което могат да бъдат оценени силите, приложени върху мобилното устройство. Той е хардуерен сензор. Когато устройството е неподвижно, то изпитва земното ускорение от 1g (9.81 m/s^2) по една от осите си или чрез векторната сума от компонентите по всички оси. Стойностите на акселерометъра минус факторът „гравитация” дават т.нар. „индуцирано” ускорение. В най-общия случай акселерометърът се използва за да се определи ориентацията на мобилното устройство, както и в компютърните игри. Измерителната единица, използвана от сензора (в система Si) е m/s^2. Следва да се вземе под внимание, че акселерометърът не измерва скорост, а интензивността на нейната промяна!

Сензорът за магнитно поле (Magnetic field Sensor) е хардуерен и измерва магнитното поле на околната среда по трите координатни оси на устройството (фиг.1) – x, y и z. Измерителната единица на сензора в система Si е микротесла (μT). Този сензор е в състояние да установи интензивността и посоката на земното магнитно поле и да бъде използван като магнитен компас. Често магнитният сензор се използва в комбинация с акселерометъра.

Сензорът за ориентация (Orientation Sensor) се реализира като софтуерен. Той измерва ъглите на завъртане на устройството около трите му координатни оси. Показанията на сензора

се получават като резултат от работата на други два сензора – магнитния и акселерометъра. От API Level 8 сензорът се счита за отречен (deprecated).

Сензорът за гравитация (Gravity Sensor) може да се реализира както като хардуерен, така и като софтуерен. Във втория случай той използва данни, постъпващи от акселерометъра. Определя направлението на вектора на земната гравитация.

Сензорът за линейно ускорение (Linear Acceleration Sensor) е софтуерен и се базира на данните от акселерометъра. Показанията му се получават от последния, като векторно се извади земната гравитация. Мерната единица е в m/s^2.

Жироскопът (Gyroscope Sensor) е хардуерен и измерва интензивността на ротацията на устройството спрямо всяка от координатните оси (фиг. 1). Когато устройството не е подложено на ротация, стойностите на сензора са нули. Той има три основни атрибута:

- Наклон (Pitch) – около остта X;
- Завъртане (Roll) – около остта Y;
- Азимут (Azimuth) – Z (от 0 до 360 градуса).

Жироскопът като сензор има известни несъвършенства – неговите показания могат да натрупват грешка с течение на времето на използване. Проблемът се изразява в „плаване" (дрейф) на стойностите му и се преодолява с комбиниране на показанията на жироскопа с тези на акселерометъра. Показанията на сензора са в мерни единици rad/s (радиани за секунди).

Сензорът за осветеност (Light sensor) е хардуерен. Поместен е обикновено на лицевата страна на устройството, непосредствено до предната камера. Той измерва равнището на осветеност в мерната единица lux. Обхватът на измерването е от 0 до максималната стойност за конкретния сензор.

Сензорът за близост (Proximity sensor) установява близостта на някакъв обект до екрана на мобилното устройство. Това може да бъде разстоянието до обекта в сантиметри или просто флаг, който индицира близостта на обекта. Сензорът за близост в някои случаи хардуерно се реализира на базата на сензора за осветеност. Android третира двата типа сензори като логически различни компоненти на устройството.

Сензорът за температура (Temperature Sensor) е хардуерен. Обикновено сензорът измерва вътрешната температура на устройството, в частност – тази на батерията. От Android API Level 14 сензорът е заменен от такъв, който измерва температурата на околната среда (**`TYPE_AMBIENT_TEMPERATURE`**) в целзиеви градуси (^{0}C).

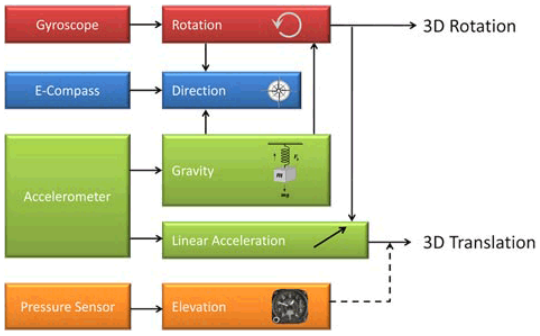
Сензорът за налягане (Pressure Sensor) също е хардуерен и измерва барометричното налягане, чрез което се определя надморската височина. Същият може да бъде използван в приложения за прогнозиране на времето. Измерителната единица на сензора е хектопаскал - hPa (респ. милибар).

Сензорът за относителната влажност на въздуха на околната среда (Humidity sensor) е хардуерен и измерва параметъра в проценти (от 0 до 100%).

Съществуват техники за филтриране на данните от сензорите и за комбиниране на показанията на два или повече сензора. Тези техники са известни под името Sensor fusion и се основават на два подхода:

- Използване на Калман-филтър;
- Използване на комплементарен филтър.

Поради спецификата на техниките и необходимостта от използване на специализиран математически апарат, същите няма да бъдат разглеждани тук. Полезна информация по темата може да се намери в много информационни източници, в т.ч. и в Internet. На схемата от фиг. 2 са илюстрирани някои от идеите за комбиниране на данните, постъпващи от хардуерните сензори с цел получаване на нови резултати.



Фиг. 2. Комбиниране и филтриране на данните от сензорите.

6. Библиотеки за работа със сензори. Основни класове.

Всички класове и интерфейси, необходими за работа със сензорите (Android Sensor Framework) са съсредоточени в пакета **`android.hardware`** . Типичните задачи, които могат да бъдат решавани с тяхна помощ са:

- Определяне на наличните сензори в устройството;
- Определяне на характеристиките на конкретните сензори в устройството – обхват, прецизност, консумирана мощност и т.н.;
- Получаване на необработени данни от сензорите и определяне на миниммалния и максимален интервал (период) от време, с които тези данни могат да бъдат получени;
- Регистрация и отмяна на регистрацията на класове-слушатели, които следят за промяна на състоянието на сензорите.

Основните класове от пакета са представени в таблица 1 .

Таблица 1. Класове, съдържащи се в пакета **`android.hardware`** за работа със сензори.

Клас/интерфейс	Описание
`Camera`	Клас, който позволява на приложенията да си взаимодействат с видеокамерата за да правят снимки, клипове, да изискват изображения за

	предварителен преглед върху екрана, да настройват параметрите за работа с камерата.
Sensor	Класът представя референция към сензор. За получаване на списък на достъпните сензори се използва методът **getSensorList(int)**.
SensorManager	Клас, който разрешава достъпа до сензорите, достъпни в платформата Android.
SensorEventListener	Интерфейс, използван за получаване на уведомления от SensorManager когато се променят стойностите на сензора. Приложението имплементира този интерфейс за да следи промените в данните и състоянието на наличние в устройството сензори.
SensorEvent	Класът представя събитие, свързано с работата на сензора и поддържа информация, отнасяща се до събитието – тип на сензора, контролно време, прецизност, както и самите данни от сензора.
GeoMagneticField	Класът се използва за оценяване на интензивността на геомагнитното поле в дадена точка.

Класът **SensorManager** е основен за работата със сензорите на мобилното устройство. Той предоставя достъп до услугата Sensor Manager от приложния слой (Application Layer) на програмния стек на Android, чрез която са достъпни основните функции за работа със сензорите: получаване на списък от наличните в устройството сензори, регистриране на слушател и следене на състоянието сензорите. Най-съществените методи, характеризиращи функциите на класа, са:

- **Sensor getDefaultSensor(int type)-** връща подразбиращият се за устройството сензор от типа, посочен в аргумента ;
- **List getSensorList(int type)** – връща списък с достъпните сензори за устройството от съответния тип;
- **boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)** – регистрира слушател от типа на интерфейса **SensorEventListener** за сензора, посочен в параметъра sensor.
- **void unregisterListener(SensorEventListener listener, Sensor sensor)** – отменя регистрацията на регистрирания слушател за дадения сензор.

Методите **registerListener(...)** и **unregisterListener(...)** имат няколко разновидности, съдържащи различен списък от параметри, отнасящи се до типа на сензорите, за които се прилагат, период на получаване на данните, поддържащ обект от клас **Handler** и др. Те се употребяват по целесъобразност в различни случаи на следене на състоянието на сензора. Пълна информация за тези методи може да се намери на страницата [4] .

Класът съдържа и някои важни целочислени константи, които се използват в настройката на параметрите на сензорите.

Константите на класа, свързани с честотата и интервала на отчитане от сензора са показани в таблица 2

:

Таблица. 2. Константи на класа **`SensorManager`**, свързани с периода на отчитане.

Константа	Описание
`SENSOR_DELAY_FASTEST`	Информацията от сензора се получава по най-бързия възможен начин (с най-кратък период).
`SENSOR_DELAY_GAME`	Интервал на получаване на информация, подходящ за игри
`SENSOR_DELAY_NORMAL`	Нормален интервал на получаване, подходящ за промени в ориентацията на екрана.
`SENSOR_DELAY_UI`	Интервал на получаване, подходящ за потребителския интерфейс.

От Android 4.0.3 API Level 15 стойностите на тези константи са фиксирани съответно на 0, 20, 67 и 200 ms (милисекунди). Възможно е да бъдат определени и собствени стойности на интервала на отчитане при регистрирането на сензора. Стойностите не са императивни, а препоръчителни при отчитането на данните. Обикновено данните, постъпващи от сензорите се получават по-бързо, ако хардуерът позволява това.

Целочислените константи, свързани с прецизността на измерването от сензора са показани в таблица 3 .

Таблица 3. Константи на класа **`SensorManager`** за прецизността на измерването.

Константа	Описание
`SENSOR_STATUS_ACCURACY_HIGH`	Сензорът предоставя данни с максимална прецизност
`SENSOR_STATUS_ACCURACY_LOW`	Сензорът предоставя данни с минимална прецизност, възможно е да се наложи калибрация на сензора.
`SENSOR_STATUS_ACCURACY_MEDIUM`	Този сензор предоставя данните със средно равнище на прецизността, евентуална калибровка на сензора може да доведе до подобрение на прецизността.
`SENSOR_STATUS_NO_CONTACT`	Сензорите, получени от сензора, са недостоверни поради това, че той няма контакт с измерваната характеристика.
`SENSOR_STATUS_UNRELIABLE`	Стойностите, получени от сензора, са недостоверни, нужна е калибровка или има проблем с четенето на данни от средата.

Освен показаните тук константи с най-честа и типична употреба, класът **`SensorManager`** съдържа и други константи, свързани с отчитане на данни по осите на координатната система, специфични константи за отчитане на гравитацията и осветеността при различни условия (вкл. и на различни планети в слънчевата система!). Някои от тях са отбелязани като deprecated в документацията.

Инстанция на класа **`SensorManager`** и достъп до услугата Sensor Manager се получава чрез извикване на метода **`Context.getSystemService(...)`** с аргумент **`android.content.Context.SENSOR_SERVICE`**:

```
...
String service_name = Context.SENSOR_SERVICE;
```

```
SensorManager sensorManager =
                (SensorManager) getSystemService(service_name);

...
```

Примери за упоребата на клас **SensorManager** за някои типични задачи са показани по-долу:

- Получаване на списък на всички налични сензори в устройството:

```
...
//Списък на всички налични сензори в устройството
List<Sensor> allSensors =
          sensorManager.getSensorList(Sensor.TYPE_ALL);
...
```

- Получаване на списък на всички сензори от типа, посочен в параметъра на метода:

```
...
//Списък на всички налични сензори от тип Sensor.TYPE_PRESSURE:
List<Sensor> pressureSensors =
          sensorManager.getSensorList(Sensor.TYPE_PRESSURE);
...
```

- Установяване на сензора по подразбиране за указания тип (предполага се, че сензорите от този тип са повече от един):

```
...
// Достъп до сензор по подразбиране от тип
Sensor.TYPE_GYROSCOPE
Sensor defaultGyroscope =
     sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
// Връща null ако няма такъв
if (defaultGyroscope != null){
  // Наличен е жироскоп и defaultGyroscope го реферира.
     ...
  }
else {
  // Грешка! Жироскоп не е наличен.
     ...
  }
...
```

Класът **android.hardware.Sensor** предоставя софтуерна абстракция на сензор с неговите технически параметри. Негови основни методи са:

- **public float getMaximumRange ()** – максимален обхват на сензора в присъщите за него измерителни единици;
- **public int getMinDelay ()** – минимален интервал от време между две последователни събития в микросекунди или нула – ако сензорът връща стойност само когато има промяна в стойността на данните.
- **public String getName ()** – символен низ с името на сензора;
- **public float getPower ()** – консумираният ток от сензора в mA, ако е активен;

- **`public float getResolution ()`** – резолюция на сензора в съответната измерителна единица.
- **`public String getVendor()`** – връща името на производителя на сензора.

Класът съдържа множество предефинирани финализирани целочислени (тип **`int`**) стойности (константи), сред които са особено важни тези, идентифициращи типа на сензора (табл. 4).

Таблица. 4. Константи на класа **`Sensor`** за типа на сензора.

Константа - означение	Тип на сензора
`TYPE_ACCELEROMETER`	Константата описва акселерометър.
`TYPE_ALL`	Константата се отнася за всички типове сензори.
`TYPE_AMBIENT_TEMPERATURE`	Константана описва сензор за температира на околна среда.
`TYPE_GRAVITY`	Константата описва сензор за гравитация.
`TYPE_GYROSCOPE`	Константата описва жироскоп.
`TYPE_LIGHT`	Константата описва сензор за осветеност.
`TYPE_LINEAR_ACCELERATION`	Константата описва сензор за линейно ускорение.
`TYPE_MAGNETIC_FIELD`	Константата описва сензор за магнитно поле.
`TYPE_ORIENTATION`	Константата описва сензор за ориенация. Отречена е от API level 8. Препоръчва се вместо нея да се използва метода **`SensorManager.getOrientation()`**.
`TYPE_PRESSURE`	Константата описва сензор за атмосферно налягане.
`TYPE_PROXIMITY`	Константата описва сензор за близост.
`TYPE_RELATIVE_HUMIDITY`	Константата описва сензор за относителна влажност.

Тези константи заменят дефиниции на константи с аналогични имена, включени в класа **`SensorManager`**, които се считат остарели и не се препоръчват за употреба.

Достъпът до референция от клас **`Sensor`** се получава чрез двата посочени по-горе метода на класа **`SensorManager`**: **`getDefaultSensor(...)`** и **`getSensorList(...)`**. Както бе отбелязано - първият метод връща сензора по подразбиране от посочения в аргумента тип, вторият – всички сензори от специфицирания в параметъра тип, което бе показано в пример по-горе.

Преглед на списъка на достъпните сензори в устройството може да бъде направен по всеки един от познатите начини в Java за обхождане на съдържанието на обект от клас **`List`**:

```
…
// allSensors съдържа списък на достъпните сензори,
// получен чрез метода getSensorList(Sensor.TYPE_ALL)
for (Sensor sns : allSensors) {
      …
}
…
```

Интерфейсът **SensorEventListener** е класическа за Android абстракция на слушател за следене на събития, възникващи при работата на сензорите. Интерфейсът съдържа два метода, които се активират при настъпване на съответните обстоятелства:

- **public void onSensorChanged(SensorEvent sensorEvent)**- методът се извиква при промяна на стойността на сензора. Данните за променените стойности се съдържат в предавания параметър sensorEvent.
- **public void onAccuracyChanged(Sensor sensor, int accuracy)** – методът се извиква когато се промени прецизността на сензора.

Съгласно класическия модел за обработка на събитията в Java интерфейсът **SensorEventListener** се имплементира в собствен клас, съдържащ кода на горните два публични метода:

```
final SensorEventListener mySensorEventListener =
                        new SensorEventListener() {
   public void onSensorChanged(SensorEvent sensorEvent) {
         // Код обработващ промените в данните от сензора.
         ...
   }
   public void onAccuracyChanged(Sensor sensor, int accuracy) {
         // Код, реагиращ на промяната на прецизността на сензора.
         ...
   }
}
```

Регистрирането на слушател от класа **SensorEventListener** става чрез описаните по-горе методи **registerListener(...)** на класа **SensorManager**. Активирането на сензорите (и свързаната с това консумация на енергия) става при регистрирането на съответния слушател. Подходящо място за регистрацията е методът **onResume()** на activity. Деактивирането (и прекратяването на консумацията от батерията) става при отмяната на регистрацията (с методите **unregisterListener()**). Целесъобразно е да се деактивират сензорите, които не се използват в приложението, особено в случаите, когато устройството е в състояние „пауза", от съображения за икономия на разходваната енергия от батерията на устройството. Примерно съдържание на кода, свързан с регистрацията и дерегистрацията на слушателя е показан в следния фрагмент:

```
…
//Регистриране на слушателя acclrListener в метода onResume()
protected void onResume() {
         super.onResume();
         sensorManager.registerListener(acclrListener,
                        sensor_accelerometer,
                        SensorManager.SENSOR_DELAY_NORMAL);
}
…

//Отмяна на регистрацията и деактивиране на сензора
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(acclrListener);
```

```
}
…
```

Класът **android.hardware.SensorEvent**, който е параметър в метода **onSensorChanged(…)** на интерфейса **SensorEventListener,** представя събитие, възникващо при работата на сензор в унифициран вид. Класът не е наследник на класа **java.uitil.EventObject**! Той притежава четири важни публично достъпни атрибута:

- **public int accuracy** – прецизността на сензора при възникване на събитието;
- **public Sensor sensor** – сензорът, който е предизвикал събитието;
- **public long timestamp** – времето, в наносекунди, когато е възникнал събитието;
- **public final float[] values** – масив от стойности от тип **float**, които съдържа стойностите, получени от сензора. Съдържанието му е специфично за всеки сензор.

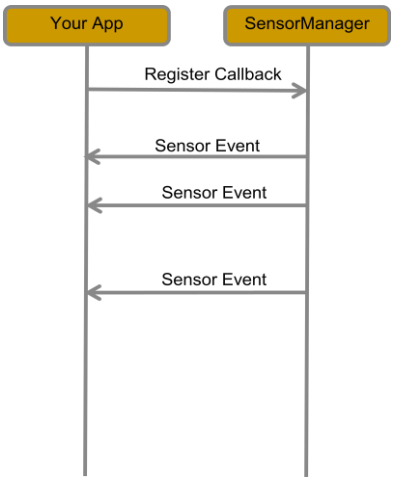
Другите два класа в пакета **android.hardware** – **Camera** и **GeoMagneticField** имат относително самостоятелна употреба и тук няма да бъдат разглеждани.

7. Общ модел на работа със сензори в кода.

Общият модел на работа със сензорите в кода по същество следва модела за обработка на събитията в Java. Той представлява конструктивния шаблон, който осигурява надеждния достъп до сензорите и използването на предоставените от тях функции. Моделът включва като компоненти източника на събитие от клас **SensorEvent** (конкретния сензор), самия обект-събитие и клас-слушател, съдържащ методи с кода, обработващ събитието. Реализира се в няколко стъпки:

- Рефериране на услугата Sensor Manager – чрез създаване на инстанция на класа **SensorManager**;
- Установяване на наличието на целевите сензори в устройството – с методите **getSensorList(…)** и **getDefaultSensor(...)** на **SensorManager**;
- Създаване на клас, имплементиращ интерфейса **SensorEventListener**;
- Съставяне на кода на методите за обработка на събитията от сензорите **onSensorChanged(…)** и **onAccuracyChanged(…);**
- Регистриране на класа-слушател към съответния сензор – с някой от методите **registerListener(...)** на класа **ServiceManager**;
- Евентуално – отмяна на регистрацията на класа-слушател с някой от методите **unregisterListener(…)** на класа **SensorManager** при отпадане на необходимостта от сензора (което е целесъобразно да се прави в методите **onPause()** и **onStop()** на activity с оглед пестене на консумация на енергия от батерията).

Схема на процеса на комуникация между приложението (Your App) и SensorMananger във времето е показана на фиг ... . Схемата илюстрира операцията по първоначална регистрация на слушател към SensorManager (Register Calback) и последователно генерираните съобщения SensorEvent, възникващи при промяна на данните в сензора.



Фиг. 3. Схема на модела за работа със сензори.

Следващият пример показва кратък код, илюстриращ регистрацията и следене на промените на данните от акселерометъра на устройството:

```
// Установяване на условията за работа с акселерометъра
public void sensorSetup() {
      SensorManager sm =
        (SensorManager)getSystemService(Context.SENSOR_SERVICE);
      int sensorType = Sensor.TYPE_ACCELEROMETER;
      // Регистрация на слушател на акселерометъра
      sm.registerListener(mySensorEventListener,
                        sm.getDefaultSensor(sensorType),
                        SensorManager.SENSOR_DELAY_NORMAL);
}

// Обект-слушател за акселерометъра
final SensorEventListener mySensorEventListener =
            new SensorEventListener() {
      // Следене за промяна на стойностите на сензора
      @Override
      public void onSensorChanged(SensorEvent sensorEvent) {
         if (sensorEvent.sensor.getType() ==
                        Sensor.TYPE_ACCELEROMETER) {
             // Прочитане на новите стойности
             float xAxis_lateralA = sensorEvent.values[0];
             float yAxis_longitudinalA = sensorEvent.values[1];
             float zAxis_verticalA = sensorEvent.values[2];
             //... евентуално – действия с променените стойности
         }
      }
      // Следене за промяна в прецизността на акселерометъра
      // Не се използва
      @Override
      public void onAccuracyChanged(Sensor sensor,
                                          int accuracy) {}
   }
};
```

Детайлите около използването и изобразяването на резултатите, получени от сензорите, изграждането на потребителския интерфейс на приложението, както и цялостната му конструкция от компоненти излизат извън обхвата и целите на настоящето изложение, поради което в този пример не са представени.

8. Използване на комбинирани данни от сензорите.

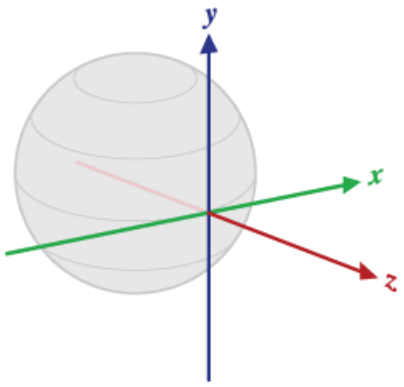
Интересна задача в работата на сензорите е съвместяването на данните, получени от няколко хардуерни източника, при което могат да се генерират синтетични показания за нови характеристики. Типичен пример за такова съчетание на данни е резултатът, който изработват софтуерните сензори. Една илюстрация на тази техника е известната задача за създаване на приложение-компас, което коректно изобразява географските посоки върху екрана на мобилното устройство. Без да се влиза в математическите особености на задачата за преобразуване на действителната ориентация на земното магнитно поле, представена в т.нар.”естествена” координатна система към координатната система на устройството (от фиг. ...), ще се отблежат два особено полезни статични метода на класа **`SensorManager`** за необходимите геометрични трансформации.

Първият статичен метод изчислява нужните за извършването на геометричната трансформация матрици на ротацията и на инклинацията (наклона):

```
public static boolean getRotationMatrix (float[] R,
      float[] I, float[] gravity, float[] geomagnetic).
```

Методът трансформира координатите на вектор от координатната система на устройството (фиг. 2) в т.нар „естествена” координатна система (привързана към местоположението на устройството) – фиг ... . Последната дефинира ортонормиран базис, с оси, ориентирани както следва:

- X е определена чрез векторното прозведение на осите Y и Z. (X = Y×Z) – тангенциална на земната повърхност в точката на текущото местоположение на устройството (ориентирана приблизително на изток);
- Y е тангенциална на земната повърхност в точката на текущото местоположение на устройството и е ориентирана към магнитния северен полюс;
- Z е нормална спрямо земната повърхност и е ориентирана към зенита.



Фиг. 4 . „Естествена” координатна система.

Параметрите на метода са следните:

R – двумерен (на практика) масив от 9 елемента (3×3), съдържащ матрицата на ротация. Получава се като резултат от изпълнението на метода. Параметърът се инициализира с референция от съответния тип, указваща празен масив.
I – масив от 9 елемента (3×3) – подобно на масив R, съдържащ матрицата на инклинация. Също се получава като резултат от изпълнението на метода и се инициализира подобно на матрицата R.
gravity – едномерен масив от три float стойности, съдържащ вектора на гражитацията, представен в координатната система на устройството.
geomagnetic – едномерен масив от три стойности от тип float, съдържащ вектора на геомагнитното поле, представен в координатната система на устройството.

Методът **`getRotationMatrix(…)`** връща резултат от тип **`boolean`**, който има стойност **`true`** в случай на успешно извършване на трансформацията и **`false`** – при неуспешното ѝ завършване.

Вторият метод изчислява фактическата ориентация на устройството въз основа на ротационната матрица R:

```
public static float[] getOrientation (float[] R, float[]
values).
```

Неговите параметри са:

R – **`float`** масив от 9 елемента, съдържащ ротационната матрица, изчислена чрез метода **`getRotationMatrix(…)`**;

values – **`float`** масив от три елемента, съдържащ резултата от трансформацията.

Резултатът от трансформацията се предава и чрез името на метода. Той е идентичен със съдържанието на параметъра му value след неговото изпълнение. Елементите на резултантния масив съдържат следните стойности (съглсасно фиг. 1):

- values [0] - съдържа азимута или ротацията около отрицателната полуос Z (противоположно насочена на оста Z);
- values [1] - съдържа ротацията около отрицателната полуос X (противоположно насочена на оста X);
- values[2] - съдържа ротацията около оста Y (roll).

Трите ъгъла на ротация се измерват в радиани в положителна посока на въртене (обратна на въртенето на часовниковата стрелка). Обхватът на отчитането е: за азимута – от –π до π, за ротацията около X – от –π/2 до π/2, за ротацията около Y – от –π до π.

Математическата формализация на описаните трансформации може да бъде намерена във всеки добър учебник по аналитична геометрия или в [ ]. Поради спецификата и обема си тук тя няма да бъде представена.

Основните задачи, които следва да бъдат решени при съставянето на приложението-компас се решават предимно в кода на класа, имплеметиращ инерфейса **SensorEventListener**, по-конкретно в неговия метод **onSensorChanged()**. Те са следните:

- Осигуряване на достъп до акселерометъра и магнитометъра на усройството и до подаваните от тях данни.
- Осигуряване на масиви за представяне на матрицата на ротация R и на инклинация I. Противно на посоченото в документацията, тези масиви следва да са предварително създадени със съответния тип и размер.
- При наличие на данни от акселерометъра и от магнитометъра – изпълнение на статичния метод **getRotationMatrix(…)**.
- При успешно изпълнение на метода **getRotationMatrix(…)** - изпълнение на метода **getOrienation(…)**.
- Изпълнението на метода връща като резултат трите ъгъла на ротация, които при необходимост могат да бъдат преобразувани от радиани в градуси със статичния метод **radToDeg(...)** на класа **Convert**.

Следният примерен код, илюстрира изпълнението на тези основни стъпки:

```
private class OrientationDemoListener
              implements SensorEventListener {
      private static final int matrix_size = 9;
      float[] gravity;     // Данни от акселерометъра
      float[] geomagnetic; // Данни от магнитометъра
      // Методът не се използва
      public void onAccuracyChanged(Sensor arg0, int arg1) {
      }
      // Регистрира промените в данните на сензорите
      public void onSensorChanged(SensorEvent event) {
         // Четене на данни от акселерометъра
         if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
            gravity = event.values;
         // Четене на данни от магнитометъра
         if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
            geomagnetic = event.values;

         // Ако данните са налични
         if (gravity != null && geomagnetic != null) {
            // Подготовка на матриците R и I
            float R[] = new float[matrix_size];
            float I[] = new float[matrix_size];
            // Пресмятане на матриците на ротация и инклинация
            boolean success = SensorManager
                  .getRotationMatrix(R, I, gravity, geomagnetic);
            // Ако е наличен резулат
            if (success) {
               float orientation[] = new float[3];
               Пресмятане на ориентацията
               SensorManager.getOrientation(R, orientation);
               // Предполага се, че променливите azimuth, pitch
               // и roll са коректно дефинирани от тип float
               // извън кода на метода.
               // Стойностите в orientation[] се преизчисляват
               // от радиани в градуси.
```

```
                azimuth = Convert.radToDeg(values[0]);
                pitch = Convert.radToDeg(values[1]);
                roll = Convert.radToDeg(values[2]);
            }
        }
    }
}
```

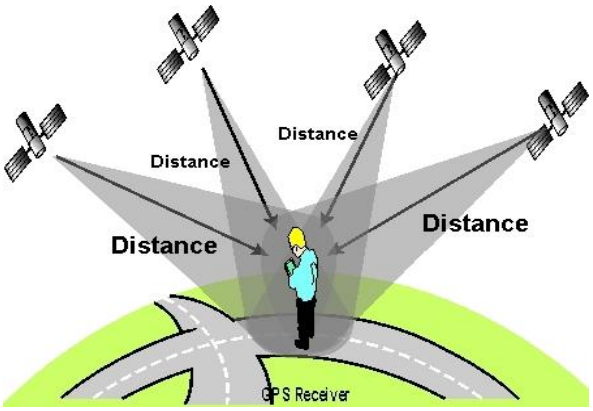
9. **Използване на услугите за местоположение**.

Мобилните приложения все по-често изискват информация относно географското местоположение на устройството и потребителя. Използването на такава информация придобива нарастваща ценност. Тя добавя разширяваща се функционалност към приложенията. Информацията за местоположението е ключов елемент за приложения като Google Maps и Google Navigator, но също така широко се използва в приложения като Google Search, Twitter и Facebook.

Android използва няколко различни източника за да предостави информация за местоположението. В Android е прието тези източници да се наричат „доставчици на местоположение" (location providers). Всеки от тях притежава специфични предимства и недостатъци, които ги правят предпочитани в различни ситуации.

Като основен и най-популярен доставчик на местоположение се определя Глобалната Система за Позициониране (Global Positioning System). Същата използва система от спътници, обикалящи около планетата и осигуряващи отделните GPS-приемници с информация, необходима за определяне на местоположението им. GPS като интегрална система включва около 27 спътника, наземни контролни станции и индивидуалните приемници на устройствата. Всеки от спътниците се движи по точно определена орбита, като пространственото им разпределение е такова, че от всяка наземна точка във всеки момент са „пряко видими" поне четири от тях. Това е необходимо условие, за да могат да бъдат извършени триангулачните изчисления за определяне на местоположението на приемника. Всеки от GPS спътниците в групата предава непрекъснато своята текуща позиция (включена в т.нар. „ефемерни данни" – ephemeris data) и т.нар."алманах" (almanac), който съдържа данни за всички спътници в групата. Алманахът включва орбитални данни и информация за всеки спътник от системата.

Определянето на местоположението на GPS-приемника става чрез триангулиране.За да изчисли своето местоположение GPS-приемникът трябва да определи достатъчно точно разстоянието си до няколко спътника (най-малко четири). За тази цел той използва ефемерните данни, които изпращат спътниците. Наред с данните за позицията на спътниците, тези съобщения съдържат данни и за момента на изпращането им. Всеки GPS-спътник съдържа високоточен механизъм за отчитане на времето, който му позволява да синхронизира своето време с това на останалите спътници. За да бъдат извършени коректно триангулачните изчисления, часовниците на GPS-спътниците и GPS-приемниците трябва да бъдат синхронизирани много точно. Дори и най-малкото отклонение в отчитането на времената може да причини значителна грешка в изчисляване на местоположението на приемника. Увеличаването на броя на GPS-спътниците, от които GPS-приемникът получава сигнал (четири и повече), съществено подобрява и точността на изчисленията.



Фиг. 5. За да се определи коректно местоположението на приемника са необходими най-малко четири GPS-спътника.

Съществено предимство на GPS-доставчика на местопложение е високата прецизност на данните и възможността местоположението на приемника да се определели с минимално отклонение. Недостатъците на тази технология са свързани преди всичко със затрудненията, които възникват при липса на пряка видимост между спътниците и приемника, както и с продължителния начален период за получаване на алманаха. От значение е и повишеният разход на енергия от батерията на устройството.

Известно усъвршенстване на използването на GPS за изчисляване на местоположението на приемника представлява технологията “Assisted GPS” (A-GPS). Тя използва мобилните мрежи за предаване на алманаха, заедно с допълнителна информация, която подпомага изчисленията в мобилното устройство. С това значително се съкращава времето за установяване на началните и текущи позиции на GPS-спътниците и преминаването на GPS-приемника в състояние на готовност.

Друга усъвършенствана технология, базирана на GPS, е S-GPS, която позволява предаването на данни за местоположението да се съвмести с предаването на друга информация (напр.мобилни телефонни разговори). Същата дава възможност за ускорено постъпване на данните по GPS, но изисква разширена апаратна част на приемащата страна.

Други възможности за получаване на данни за местоположението на устройството са чрез безжичната (Wi-Fi) мрежа или чрез идентификация на текущата клетка на мобилната телефонна мрежа. Това са така наречените „мрежови” доставчици на услугата. Те използват сходен по между си подход за осигуряване на услугата, като разчитат на регистрирания MAC-адрес на безжичната точка на достъп или на идентификатора на мобилната клетка. Прилаганият от тази технология механизъм за определяне на местоположението оценява и силата на сигнала, получен от мобилното устройство на потребителя. Използването на безжичните и мобилни мрежи като цяло дава значително по-ниска прецизност на данните за местоположението от тази на GPS-доставчика, но в някои случаи се оказва приложима в условия, в които спътниковата система има пролеми с достъпа и покритието (интензивно застроени градски райони, вътрешни помещения и т.н.).

10. Приложен програмен интерфейс на услугата за местоположение.

Основната част от приложния програмен интерфейс, необходима за работа с услугите за местоположение е съсредоточена в пакета **`android.location`**. Това са четирите класа **`LocationManager`**, **`LocationProvider`**, **`Location`** и **`Criteria`**, както и интерфейсът **`LocationListener`** (табл. 5).

Таблица 5 . Състав на приложния програмен интерфейс

Клас/интерфейс	Описание
`LocationManager`	Основният клас, който осигурява достъпа до услугата за местоположение. Класът позволява да бъдат инициализирани необходимите действия за работа с услугата за местоположение, да бъде получена информация за текущото състояние на системата за локализация, за наличните и за разрешените доставчици на услугата, за статуса на GPS информацията и др.
`LocationProvider`	Абстрактен суперклас за доставчиците на местоположение. Всеки доставчик предоставя периодични съобщения за географското местоположение на устройството.
`Location`	Инкапсулира данните за текущото местоположение, предоставени на приложението от избрания доставчик на услугата. Той съдържа географски координати като ширина, дължина и надморска височина.
`Criteria`	Класът се използва за да се състави заявка към **`LocationManager`** в търсене на доставчици на местополжение, които съдържат определени характеристики.
`LocationListener`	Интерфейс-слушател, съдържащ група методи, които се извикват в отговор на промените на текущото местоположение на устройството или при промяна на състоянието на съответната услуга.

`LocationManager` е основният клас в изграждането на структурата на кода, ползващ услугата за местопожение. Чрез него се извършва началната подготовка на операциите по получаване на данни за местоположението и за периодичното им актуализиране. Класът съдържа множество методи, които могат да бъдат групирани по функционален признак:

- Получаване на информация за доставчиците, отговарящи на определени характеристики и достъп до определен доставчик – достъпни, разрешени, по посочен критерий (чрез клас **`Criteria`**) - **`getAllProviders()`**, **`getBestProvider(Criteria criteria, boolean enabledOnly)`**, **`getProvider(String name)`**, **`getProviders(boolean enabledOnly)`** и др. Тук влиза и проверката за достъпност на доставчика – **`isProviderEnabled(String provider)`** ;
- Регистриране на слушатели на GPS-статуса и следене за актуализация на данните за местоположението – методите ... , както и групата методи **`requestLocationUpdates(...)`**;
- Установяване на тестов доставчик и тестови стойности .

Класът **LocationManager** също така предоставя и информация за последното (хеширано) местоположение на мобилното устройство – чрез метода **getLastKnownLocation(String provider)**.

Класът съдържа важни константи-символни низове, свързани с избора и иденификацията на доставчика на местоположение – **GPS_PROVIDER**, **NETWORK_PROWIDER**, **PASSIVE_PROVIDER**, чиито имена са достатъчно описателни.

Създаването на инстанция на класа **LocationManager** и на референция към услугата Location Mananger става по стандартен начин – чрез метода **getSystemService(...)** с посочване на вида на услугата:

```
LocationManager locationManager =
            (LocationManager)
            getSystemService(Contex.LOCATION_SERVICE);
```

Класът **LocationProvider** е абстракция на различните източници на информация за местоположението в Android. Както бе отбелязано вече, възможни са различни източници за такива данни, които имат драстично различаващи се характеристики, представени в различна форма. Предназначението на класа е да осигури достъп до тези данни по еднотипен, универсален начин за използващите ги приложения. Класът **LocationProvider** предоставя методи, чрез които могат да бъдат получени параметрите на различните доставчици:

Класът **LocationProvider** разполага и с няколко константи, които обикновено се използват при установяване на състоянието на доставчика на местоположение – **AVAILABLE** (достъпен), **OUT_OF_SERVICE** (недостъпен за неопределен срок), **TEMPORARILY_UNAVAILABLE** (временно недостъпен).

Класът **Location** инкапсулира данните за текущото местоположение, предоставени на приложението от избрания доставчик на услугата. Той съдържа атрибути на местоположението като географски координати за ширина, дължина и надморска височина. След получаване на обект **Location** приложението може да започне обработката на представените чрез него данни. Важно е да се знае, че класът съдържа много широк набор от атрибути, които не всички доставчици на местоположение използват. Типични методи на класа са тези, които прочитат стойностите на тези атрибути: **getAccuracy()**, **getAltitude()**, **getLatitude()**, **getLongitude()**, **getProvider()**, **getSpeed()**, **getTime()**. Налични в класа са и методи за установяване на стойността на същите атрибути – **setAccuracy(…)**, **setAltitude(…)**, **setLatitude(…)**, **setLongitude(…)** и др.

Класът разполага и с методи, които позволяват да се провери дали атрибутите на конкретния обект от класа съдържат съответната информация (напр.методите **hasAccuracy()**, **hasAltitude()** и др.). Друга група методи са тези, които са предназначени за премахване на съдържанието на съответните атрибути от конкретния обект в случаите, когато не се изплзват – **removeAccuracy()**, **hasAltitude()**, **removeSpeed()** и т.н.

Приложението получава референция към обект от клас **Location** или чрез метода **getLastKnownLocation(String provider)** на клас **LocationManager**, или чрез

параметъра на метода **onLocationChanged(Location location)** на интерфейса **LocationListener**, извикван при промяна на данните за местоположението. Данните от референцията са достъпни чрез методите на класа.

Класът **Criteria** се използва за да се създаде заявка към **LocationManager** в търсене на доставчици на местоположение, които съдържат определени характеристики. Полезен е за случаите, когато приложението се интересува с предимство от получаването на определени характеристики на данните, отколкото от доставчика им. Класът спестява на приложението необходимостта да се грижи за детайлите, които се изискват за директна работа с индивидуалните доставчици на местоположение. Атрибутите на класа (частни) могат да бъдат четени и променяни при необходимост в процеса на изпълнение на приложението.

Методите на класа са предназначени основно за достъп и манипулиране на атрибутите на класа. Това са методи за установяване на техните стойности (напр. **setAccuracy(int accuracy)**, **setAltitudeRequired(boolean altitudeRequired)**, **setCostAllowed(boolean costAllowed)**, **setPowerRequirement(int level)**) и за получаване на стойностите им (**getAccuracy()**, **getHorizontalAccuracy()**, **getPowerRequirement()**, **isAltitudeRequired()**, **isCostAllowed()** **isSpeedRequired()**).

Класът **Crieria** съдържа няколко целочислени константи, които се използват при задаване на стойностите на неговите атрибути. Константите са два типа – такива, които определят изискванията към прецизността на данните за местоположението **ACCURACY_COARSE**, **ACCURACY_FINE**, **ACCURACY_HIGH** и **ACCURACY_MEDIUM**, както и такива, които са свързани с равнището на консумация на енергия от батерията **POWER_HIGH**, **POWER_LOW** и **POWER_MEDIUM**.

Обект от клас **Criteria** се създава преди извикването на методите на класа **LocationManager**, които връщат референция или списък от референции към доставчици, отговарящи на специфицираните характеристики (методи като **getBestProvider()** и **getProviders()**). Същият обект се използва като параметър в съответните методи.

```
...
// Създаване на обект, съдържащ изискваните атрибути
Criteria criteria = new Criteria();
// Установяване на арибута accuracy
criteria.setAccuracy(Criteria.ACCURACY_FINE);
// Изисква ли се надморска величина (не)?
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setCostAllowed(true);
// Изискване към ниско потребление на енергия
criteria.setPowerRequirement(Criteria.POWER_LOW);
// Референция към „най-добрия” доставчик
String provider =
           locationManager.getBestProvider(criteria, true);
...
```

Методът **getBestProvider(Criteria criteria, boolean enabledOnly)** връща референция към доставчика на услугата, който най-добре съответства на съставения

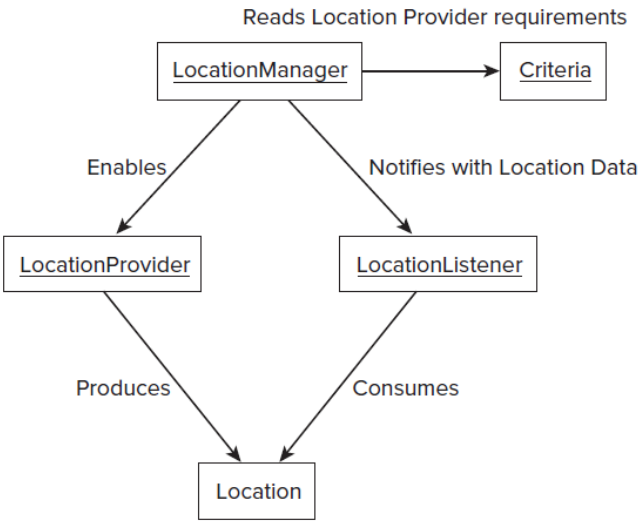
критерий. Списък на всички доставчици, отговарящи на критериите, може да се получи чрез метода **`getProviders(Criteria criteria, boolean enabledOnly)`** на **`LocationManager`**, както е показано в пример по-горе.

`LocationListener` е интерфейс-слушател, който съдържа групата методи, които се извикват в отговор на промените на текущото местоположение на устройството или при промяна на състоянието на съответната услуга. Съставянето на кода на методите, естествено, е отговорност на програмиста. Регистрацията или отмяната на регистрация на класа, имплементиращ слушателя, може да стане чрез няколко метода на класа **`LocationManager`** (напр. requestSingleLocation()).

Методите, които съдържа интерфейсът **`LocationListener`** за обработка на постъпващите от услугата за местоположение събития са следните:

- **`abstract void onLocationChanged(Location location)`** – извиква се, когато се променят данните за местоположението. Тези данни се предават на метода чрез параметъра му location.
- **`abstract void onProviderDisabled(String provider)`** – извиква се, когато доставчикът на местоположението бива забранен от потребтеля (от менюто за настройките на Android);
- **`abstract void onProviderEnabled(String provider)`** – извиква се, когато доставчикът на местоположението бива разрешен от потребтеля;
- **`abstract void onStatusChanged(String provider, int status, Bundle extras)`** – извиква се, когато статусът на доставчика се промени – когато той бива активиран или деактивиран. Тази промяна не е свързана с действията на потребителя, както в горните два метода. Параметърът provider съдържа идентификацията на доставчика, status – текущия му статус (**`OUT_OF_SERVICE`**, **`AVAILABLE`**, **`TEMPORARY_UNAVAILABLE`**), а extras – допълнителна информация, представена в обект от клас Bundle.

Връзката между класовете, участващи в осигуряването на данните за местоположението е показана на фиг. 6 .



Фиг. 6. Отношения между класовете в пакета **`android.location`**.

11. Особености на използването на услугата за местоположение.

Достъпът до услугите за местоположение и работата с класовете от пакета **`android.location`** са свързани с някои особености, които следва да бъдат взети под внимание. Най-важните от тях са:

- специфични механизми за уведомяване при промяна на местоположението;
- избор на подходящ доставчик на местоположение;
- прилагане на техники и стратегии за икономия на енергия.

За първата от посочените особености е съществено е да се отбележи, че механизмът за получаване на актуализациите в местоположението има две разновидности:

- Съставяне на клас, имплементиращ **`LocationListener`** и регистриране на съответен обект-слушател. Този подход е близък (но не идентичен) на регистрирането на слушатели в общия модел за обработка на събитията в Java.
- Уведомяване (предимно на външни приложения) за промените с използването на „висящ" intent (**`PendingIntent`**).

Първият вариант се приема за по-опростен, докато вторият – за по-гъвкав и универсален, особено в случаите когато за промените в местоположението трябва да бъдат уведомени едновременно няколко приложения. И в двата случая е необходимо приложението да информира **`LocationManager`** за намерението си да започне следене на промените чрез някой от методите на класа:

- **`void requestLocationUpdates(long minTime, float minDistance, Criteria criteria, LocationListener listener, Looper looper)`** – методът регистрира слушател listener, имплементиращ интерфейса за следене на промените в местоположението, като посочва желаните характеристики на досавчика в обект от клас **`Criteria`**;
- **`void requestLocationUpdates(long minTime, float minDistance, Criteria criteria, PendingIntent intent)`** – регистрира следене на промените в местоположението чрез „висящ" intent (параметър intent). Характеристиките на доставчика са включени в параметъра citeria.

Съществуват варианти на горните два метода, в които като първи параметър директно се посочва конкретният доставчик на местоположение.

Честотата на уведомяването се управлява от параметрите minTime (в милисекунди) и minDistance (в метри) на показаните по-горе методи. Ако параметърът minTime е по-голям от 0, Location Manager най-вероятно ще изчака този интервал време, преди да предаде информацията за промененото местоположение. Това се прави с оглед намаляване на честотата на актуализация на данните и за икономисване на изразходваната енергия от батерията. Ако параметърът minDistance се установи със стойност, по-голяма от 0 – промените в местоположението ще бъдат оповестени само ако се регистрира преместване на посоченото

разстояние. Съвместното действие на двата параметра най-точно може да се опише от логическата операция AND. Ако и двата параметъра се установят в 0 – това ще доведе до максимална честота на уведомяването, но с цената на значителен разход на енергия.

Отмяната на регистрацията и прекратяване на следенето на промените в местоположението става с методите **`removeUpdates(LocationListener listener)`** и **`removeUpdates(PendingIntent intent)`**.

Класът **`LocationManager`** разполага още и с методи, които се използват за еднократно актуализиране на данните за местоположението - **`requestSingleUpdate(Criteria criteria, LocationListener listener, Looper looper)`**, **`requestSingleUpdate(String provider, PendingIntent intent)`** и др. В този случай параметрите minTime и minDistance не се употребяват.

Втората особеност - определянето на подходящ доставчик на местоположение – е свързана с основния въпрос за търсения баланс между по-висока прецизност на отчитането и по-нисък разход на енергия (в някои случаи – и на парични средства). Тези две характеристики са противоположно свързани. Мобилните източници на данни за местополжението предоставят различни равнища на прецизност и на консумация на енергия. В повечето случаи разработчиците и потребителите отдават предпочитание на по-високата прецизност (осигурявана предимно от GPS-доставчик), дори и в случаите когато такава фактически не е необходима. Посочването на желания доставчик на услугата в кода наприложението може да стане по два начина:

- чрез изричното регистриране на желания доставчик в **`LocationManager`**;
- чрез специфициране на желаните атрибути на доставчика в обект от клас **`Criteria`**. Този начин в полезен с това, че дава възможност на потребителя да подбере източника на данни за местоположението по време на изпълнение на приложението, тъй като ползването на някои от източниците може да бъде свързано с допълнителни разходи.

GPS-доставчикът е с най-висока прецизност, но технически използва специализиран радио-хардуер в устройството и консумира повече енергия от другите доставчици. В допълнение към това GPS-доставчикът изисква допълнително време за да получи информация за изходното разположение на спътниците от GPS – така нареченото “time to first fix” (TTFF), което може да продължи повече от минута. GPS-доставчикът е неподходящ за отчитане на местоположението при липса на пряка видимост към спътниците, доставящи данните.

При мрежовите доставчици на местоположението времето TTFF е значително по-кратко от това на GPS-доставчика, но прецизността естествено е много по-ниска и не се поддава на различни софтуерни техники и алгоритми за корекция. Консумацията при този тип доставчици е в рамките на нормалната консумация на устройството, тъй като не се използва допълнителен хардуер за реализиране на комуникацията, по която постъпват данните за местоположението.

Като трета възможна група доставчици на местоположение се посочват т.нар. „пасивни” (passive) доставчици. Те позволяват на приложението да получи информация за местоположението без изрично да я изискват от **`LocaionManager`**. Пасивните досавчици предоставят данни за положението с посредничеството на друго приложение, което ползва

GPS- или мрежов досавчик. Доставката на данните от пасивния източник не е гарантирана, доколкото към текущия момент може да няма активни приложения, които ползват GPS или мрежа. Тази група доставчици не изискват допълнителен разход на енергия.

Третата особеност - изборът на подходящи техники и стратегии за спестяване на енергия, особено в случаите, когато се използва GPS-доставчик, е особено важен елемент от разработване на приложението, въпреки формалната простота при използването на класовете от софтуерната рамка на пакета **android.location**. Във всички случаи се препоръчва активиране на услугата за местоположение (чрез методите **requestLocationUpdates(...)** на **LocationManager**) при активиране на приложението (обикновено става в метода **onResume()** на изпълняваната activity) и деактивирането на тази услуга, когато приложението премине в състояние „пауза" или бъде спряно (свързано е с изпълнението на методите **onPause()** и **onStop()** на activity). Подобно на работата с другите типове сензори и тук идеята е специализираният хардуер да консумира електроенергия единствено когато следенето на местоположението е необходимо. Примерна регистрация и дерегистрация за следене за промени в на данните за местоположението е показана в следните фрагменти от код:

```
...
// Активиране на следенето на местоположението от доставчик GPS
@Override
  protected void onResume() {
    super.onResume();
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
                                           1000, 10, this);
  }
...
// Деактивиране на доставчика в метода onPause()
@Override
  protected void onPause() {
    super.onPause();
    locationManager.removeUpdates(this);
  }
...
```

При така отбелязаните особености за ползването на софуерните средства за установяване и следене на местоположението на усройството, моделът за достъп и използване на услугите за местоположението в кода на приложението включва следните стъпки:

- Получаване на референция към **LocationManager** и достъп до услугата за местоположение. Обикновено това се прави в метода **onCreate()** на главната activity, тъй като това е първият метод, извикван в нейния жизнен цикъл.
- Избор на доставчик **LocationProvider** – чрез изричното му посочване или по зададени атрибути в **Criteria**.
- Избор на начин за уведомяване за промените в местоположението – чрез **PendingIntent** или **LocationListener**.
- Регистриране на кода, обработващ промените в местоположението и в състоянието на доставчика („слушателя" на събитието). Регистрацията се извършва с посочените по-горе методи на класа **LocationManager** и зависи от избрания подход на

уведомяването (с **LocationListener** или с **intent**). Кодът на регистрацията се поставя в метода **onResume()** на компонента activity.

- Отмяна на регистрацията в случаите, когато приложението става неактивно и при прекратяване на работата му, с оглед икономия на енергия. Обикновено се прави в метода **onPause()** на компонента activity.

Следният кратък пример илюстрира най-съществените моменти от използването на софтуерната рамка (класовете от пакета **android.location**) за ползване на услугата за местоположение:

```
...
// Импортирането на пакетите и класовете тук
// не е показано, тъй като е тривиално
// Класът LocationDemoActivity наследява клас Activity
// и ИМПЛЕМЕТИРА интерфейса LocationListener
public class LocationDemoActivity extends Activity
                                    implements LocationListener {
      protected LocationManager locationManager;
      protected Location location;
      protected String provider;
      private double latitude;
      private double longitude;

      // Методът onCreate() се извиква при стартиране
      // на activiry. Тук обикновено е мястото за
      // инициализращите операции за ползване на услугата
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            // Обичайни действия за изграждане на GUI ...
            ...
            // Достъп до услугата за местоположение
            locationManager = (LocationManager)
                        getSystemService(Context.LOCATION_SERVICE);
            // Съставяне н критерий за избор на доставчик
            Criteria criteria = new Criteria();
            criteria.setAccuracy(Criteria.ACCURACY_FINE);
            criteria.setAltitudeRequired(false);
            criteria.setBearingRequired(false);
            criteria.setCostAllowed(true);
            criteria.setPowerRequirement(Criteria.POWER_LOW);
            // Избор на „най-добър" доставчик ...
            provider = locationManager
                        .getBestProvider(criteria, true);
            // ... или пък директното задаване на GPS-доставчик
            // provider = LocationManager.GPS_PROVIDER;
            // Инициализация на местоположението
            location = locationManager
                        .getLastKnownLocation(provider);
            // Ако инициализацията е успешна
            if (location != null) {
                  Log.d(TAG, location.toString());
                  // Получаване на географска ширина и дължина
                  latitude = location.getLatitude();
                  longitude = location.getLongitude();
            }
      }

      // Методът onResume() изпълнява регистрацията на слушателя
```

```
	@Override
	public void onResume() {
		super.onResume();
		locationManager
			.requestLocationUpdates(provider, 2000, 1, this);
	}

	// Методът onPause() изпълнява дерегистрацията на слушателя
	@Override
	public void onPause () {
		super.onPause();
		locationManager
			.removeUpdates(this);
	}

	// Изпълнява се при промяна на данните за местоположението
	@Override
	public void onLocationChanged(Location location) {
		if (provider.equals(LocationManager.GPS_PROVIDER))
		String msg = "New Latitude: " + location.getLatitude()
				+ "New Longitude: "
				+ location.getLongitude();
		// Контролно извеждане на координатите
		Toast.makeText(getBaseContext(), msg,
			Toast.LENGTH_LONG).show();
		}
	}

	// Изпълнява се  при промяна на статуса на доставчика
	@Override
	public void onStatusChanged(String provider,
				int status, Bundle extras) {
		// Действия при промяна на статуса ...
	}

	// Изпълнява се при неактивиран доставчик
	@Override
	public void onProviderDisabled(String provider) {
		// Стартиране на компонента за настройки на системата
		Intent intent = new Intent(Settings
				.ACTION_LOCATION_SOURCE_SETTINGS);
		startActivity(intent);
		Toast.makeText(getBaseContext(),
				"Provider is turned off!! ",
				Toast.LENGTH_SHORT).show();
	}

	// Изпълнява се при включен доставчик
	@Override
	public void onProviderEnabled(String provider) {
		// Кнтролно съобщение за включен доставчик
		Toast.makeText(getBaseContext(),
			"Provider is turned on!! ",
			Toast.LENGTH_SHORT).show();
	}
}
```

12. Деклариране на използването на услугите за местоположение в `AndroidManifest.xml`.

Както е при повечето услуги, които се ползват в Android, услугите за местоположение изискват приложението да декларира намерението си за ползването им в своя манифестен файл. Декларацията в **`AndroidManifest.xml`** трябва да определи изискваната прецизност на данните за местоположението. Двете възможни стойности на исканото разрешение са **`android.permission.ACCESS_COARSE_LOCATION`** и **`android.permission.ACCESS_FINE_LOCATION.`** Освен това разрешение задължително следва да се заяви и разрешението за достъп до **`Internet`** в случая, когато се ползват данни за местоположението, доставени по безжичната мрежа или пък GPS-услугата ползва технологията A-GPS.

```
…
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
…
```

Счита се, че за GPS- и за пасивните доставчици е целесъобразно да ползват разрешение от типа **`android.permission.ACCESS_FINE_LOCATION`**, докато за мрежовите доставчици е подходящо разрешението **`android.permission.ACCESS_COARSE_LOCATION`**.

Заключение:

Представеното съдържание насочва вниманието предимно към основните принципи и техники за работа с разнообразието от сензори и с услугата за местоположение, които предоставя ОС Android. Същите съставят фундамента, върху който се изграждат разнообразните полезни софтуерни приложения и услуги, представляващи интерес за съвременния потребител. Множеството популярни услуги, които ползват усъвършенстваната софтуерна рамка Google Location Services API (част от Google Play Services), преоставят например значително по-разширени възможности за работа с услугите местоположение, но се нуждаят от задълбочено самостоятелно разглеждане.

Литература:

1. Milette Greg, Adam Stroud, PROFESSIONAL Android™ Sensor Programming, John Wiley & Sons, Inc., 2012.
2. Phillips Bill, Brian Hardy, Android Programming: The Big Nerd Ranch Guide, Big Nerd Ranch, Inc., 2013.
3. Komatineni Satya, Dave MacLean Pro Android 4, Apress, 2012.
4. Интернет ресурс: Sensors Overview, Developers, http://developer.android.com/guide/topics/sensors/sensors_overview.html.
5. Интернет ресурс: SensorManager, Developers, http://developer.android.com/reference/android/hardware/SensorManager.html