Android UI, more verbose app.

- Целта на днешното занятие е да покажем, UI компонентите в Android, като код – Java и .xml;
- В Лекция #3 се запознахме с основните групи видими обекти – **View** и **ViewGroup**;
- Днес ще ги разгледаме, като „жив" код в рамките на приложение - **TryView**;
- Тъй като UI компонентите, в много от случаите са основната част на дадено приложение, ще запознаем и с начина сами да си разработваме такива;
- Или по друг начин казано – Какво ни очаква, решавайки се на custom UI development?

TryView app, some notes.

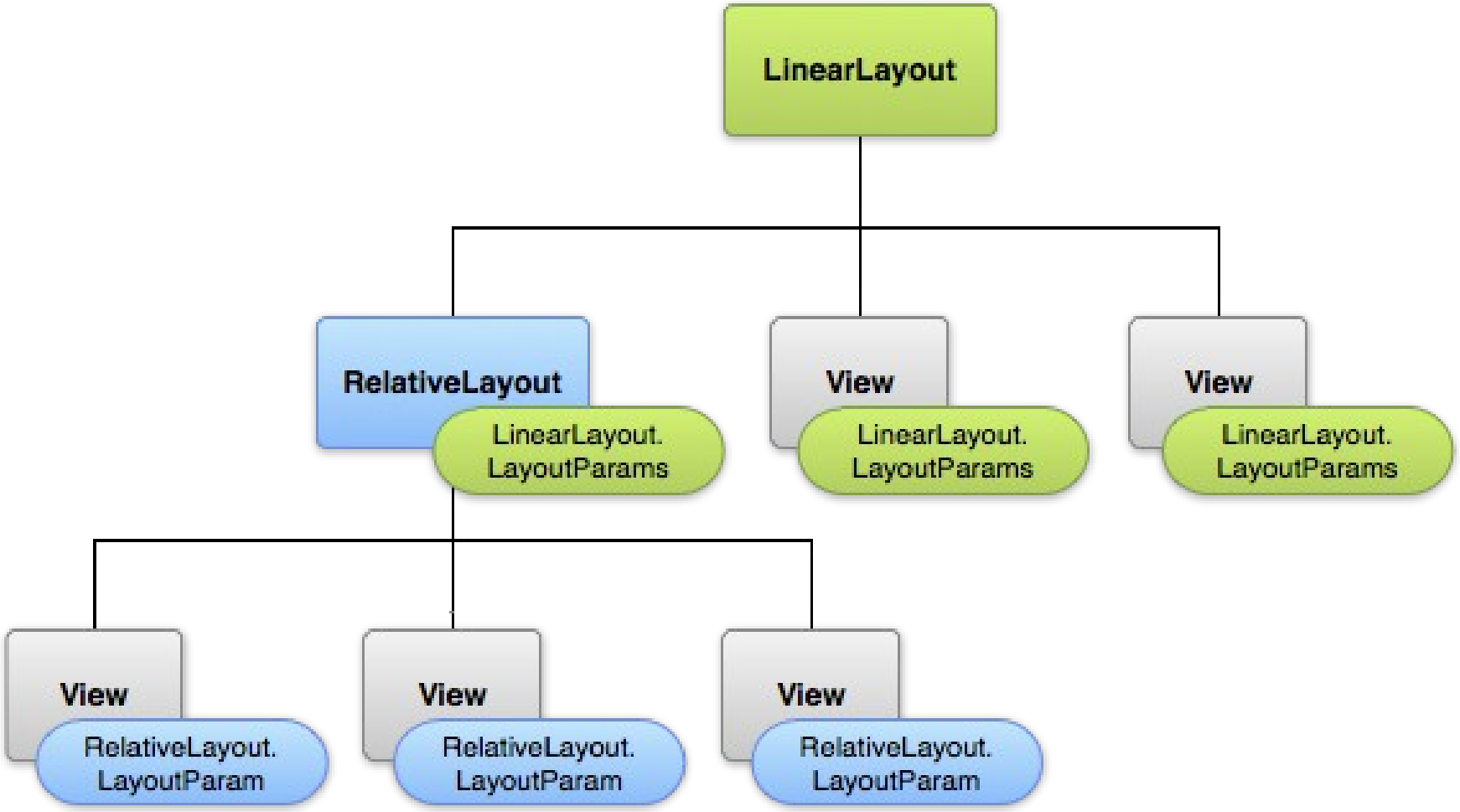
- Приложението използва единствено activity – **MainActivity**;
- Условно съм го разделил на три части, които се управляват /активират :)/ от **onCreate**() метода на нашето activity;
- Част 1 – По-подробно запознаване с **LinearLayout**, **RelativeLayout** и „MixedLayout" като .xml и properties;
- Част 2 – Да си конструираме layout, само с Java код;
- Част 3 – Да си направим сами **View** (UI) компонент и да го „управляваме" както през .xml, така и през Java код;

Част 1 – layouts, layouts ...

- За да активираме примерните layouts:
- onCreate() в MainActivity;
- setContentView(R.layout.<name>);
- <name> := empty_linear, linear_horizontal, linear_vertical, relative, mixed;
- empty_linear е празен и се използва по-подразбиране;
- Казахме че layouts са ViewGroup обекти, чиято основна задача е да организират (подреждат) View обекти или други ViewGroup обекти;

Част 1 – layouts, layouts ...

- Йерархично, това изглежда така:



Част 1 – layouts, layouts ...

- Тоест в началото, почти винаги имаме layout обект, който съдържа нашите UI обекти, като последните могат да бъдат и други layout обекти (т.е. .xml markup описващ ViewGroup съдържа .xml markup описващ View обект);
- В нашите примерни layouts използваме като етикети три TextView компонента, с различен фонов цвят (зелен, син, жълт) и текстови имена, за да покажем подреждането, което се налага от използваният layout;
- Базовите (задължителни) properties на за всеки View или ViewGroup обект бяха:
- android:layout_width="match_parent|wrap_content|<pixels>"
- android:layout_height="match_parent|wrap_content|<pixels>"
- android:id="@+id/<component_label>"

Част 1 – layouts, layouts ...

- match_parent – указва width/height атрибута на даден обект, да наследи този на съдържащият го обект;
- wrap_content – указва width/height на даден обект, да бъде съобразен със размерите на компонента който той описва (представлява);
- Третата възможност за width/height атрибутите е задаване на фиксирана големина в pixels;
- За изредените примерни layouts (res/layout/<name>.xml):
- linear_horizontal – линеен layout с хоризонтална наредба; Тоест нашите три етикета са разположени един до друг, по width на съдържащото ги activity;
- linear_vertical – линеен layout с вертикална наредба; Тоест етикетите са разположени един под друг, по height на съдържащото ги activity;

Част 1 – layouts, layouts ...

- relative – освен възможността да ориентираме View обектите, спрямо съдържащият ги layout, разполагаме и с възможност да ги ориентираме един спрямо друг;
- За целта активно се използва android:id атрибута, тоест при RelativeLayout дефинирането е един вид задължително;
- При RelativeLayout, получаваме възможност да използваме атрибути като android:layout_above, android:layout_below, android:layout_toLeftOf, android:layout_toRightOf;
- Използвайки тях и android:id ориентираме обекта за който ги използваме, спрямо цитираният (по id) друг View обект този layout;
- Тоест RelativeLayout ни дава доста по-гъвкав начин за организиране на обектите – спрямо колонното или поредово подреждане на LinearLayout например;

Част 1 – layouts, layouts ...

- Комбинирайки LinearLayout и RelativeLayout – mixed;
- Идеята – линеен layout с вертикална организация;
- Син етикет – разположен в ляво;
- Жълт етикет – разположен в дясно;
- Зелен етикет – разположен в центъра на оставащата част от екрана;
- За целта използваме LinearLayout (vertival) както до момента, с един допълнителен атрибут за всеки от етикетите – android:layout_gravity="left|right";
- За останалата част от екрана използваме RelativeLayout, със widh/height = match_parent
- На съдържащият се в него трети етикет, добавяме: android:layout_centerInParent = „true“

Част 2 – layout без .xml;

- С други думи – метод makeLayoutWithCode() в нашето MainActivity;
- Стратегията – създаваме първо всеки от View обектите;
- Създаваме layout-а, който ще ги съдържа, като установяваме базовите му properties;
- За всеки от компонентите, създаваме properties, според съдържащият го layout и му ги прилагаме;
- Ако не създадем такива, се наследяват базовите properties на съдържащият го ViewGroup обект (ключов момент, както се вижда и на слайд №4);
- Добавяме всеки от View обектите, към ViewGroup обекта, който ще ги съдържа;
- Установяваме /setContentView()/ така конструираният ViewGroup обект, като основен изгледа за нашето activity;
- Note: При създаването на всеки View и ViewGroup обект, указваме контекста в който той ще бъде визуализиран; В случаят това е нашето MainActivity, заради което в конструкторите използваме this;

Част 3 – make custom view?

- Видяхме как можем организираме layouts чрез .xml и чрез Java код;
- Използвайки само Java код, без да разчитаме на .xml markup и Android Resource Compiler, моделирането на layout се оказва доста трудно занимание;
- Целта сега е да видим, какво ще ни струва разработката на собствен View компонент;
- Причините могат да са най-различни – от недостатъчна функционалност на вградените в Android компоненти :), до например хрумка за нещо по различно :);
- Базова (и доста подробна) dev документация за това, можем да намерим в описанието на View класа на Android:
- https://developer.android.com/reference/android/view/View.html

Част 3 – make custom view?

- За да видим работещ нашият custom View използваме R.layout.roundb_linear в onCreate()/setContentView() на нашето MainActicity;
- Това е готов linear layout, използващ компонента. Кода, реализиращ компонента се намира в RoundB.java класа на приложението;
- Тоест: public class RoundB extends View { ... } && let the saga begin ... :)
- Кои са основните моменти върху които трябва да се съсредоточим при реализирането на този компонент?
- 1) Какви са неговите properties?
- 2) Какви са неговите размери? Тоест какви размери са му отредени според Android?
- 3) Визуализация ?
- 4) Динамично или не е този компонент?

Част 3 – make custom view?

- 1) в зависимост от това, дали нашето View се създава чрез Java код или чрез .xml имаме два базови конструктора:
- public RoundB(Context context) { ... } - за Java код;
- public RoundB(Context context, AttributeSet attrs) { ... } - за .xml markup;
- Вторият конструктор, получава всички стандартни Android proeprties, като AttributeSet аргумент;
- Задължение на конструкторите е да отработят правилно properties (или липсата им) :);
- 2) Задължително: protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) { ... }
- Тоест компонента трябва да се съобрази със отреденото му петно, за визуализация;
- Ограниченията за визуализация се получават като аргументи на метода;
- След необходимите пресмятания на това, какво всъщност ще може да бъде изобразено задължително се извиква - setMeasuredDimension(<width>, <height>);

Част 3 – make custom view?

- 3) за визуализация предефинираме: protected void onDraw(Canvas canvas) { ... }
- В резултат на стойностите от 2), получаваме готов (оразмерен) canvas обект, в/у който можем да рисуваме;
- До тук, имаме обект, който се съобразява с .xml properties и отреденото му за визуализация място ;), но уви само толкова.
- Ако искаме нашият View обект да е динамичен – например да реагира на Tap/Click, то трябва да се погрижим и за това;
- 4) предефинираме: public boolean onTouchEvent(MotionEvent event) { ... }
- Ключови моменти тук?
- Да разграничим събитията, които се случват (UP/DOWN/MOVE);
- В зависимост от това да променим състоянието на нашият обект;
- Да изискаме изобразяването му наново – invalidate();
- True, като върната стойност за да уведомим Android View manager, че събитието е отработено;