Cours 8

Création de vues personnalisées

- Jusqu’à présent, nous avons utilisé les classes de vues fournies par Android. Il est possible de créer nos propres vues et d’implémenter nous-même leur comportement.
- La majorité des engins sont faits avec une seule activité contenant une seule vue dans laquelle l’engin dessine.

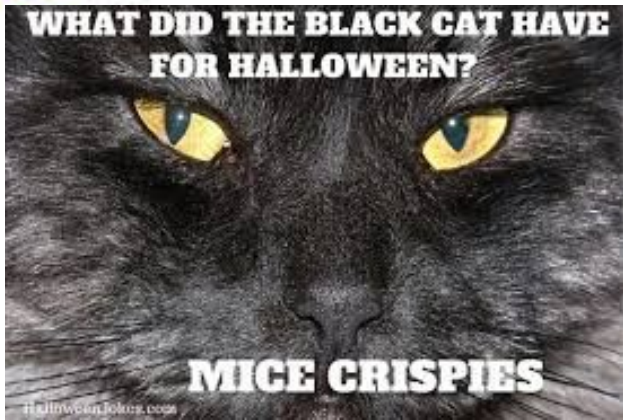
- Pour créer une sous-classe de vue, il suffit de créer une classe Java héritant de **`android.view.View`** et d'implémenter un ou plusieurs de ses constructeurs:

```
import android.content.Context;
import android.view.View;
import android.util.AttributeSet;

public class MyView extends View {

    public MyView(Context context) {
        this(context, null);
    }

    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

- Le plus simple constructeur prend un contexte en paramètre (c'est-à-dire, l'activité qui a instancié la vue) et est uniquement utile pour créer la vue par code.
- Les autres constructeurs servent à instancier la vue à partir d'attributs XML présents dans un fichier de "*layout*". Certains permettent même de charger des valeurs par défaut à partir du thème.

- Vous pouvez maintenant l'ajouter dans un fichier de "*layout*" en spécifiant le nom de la classe complet (incluant le paquet):

```
<ca.bart.customview.MyView
    android:layout_width="300dp"
    android:layout_height="300dp" />
```



- Pour dessiner le contenu de la vue, on doit implémenter la méthode **`onDraw`**. Celle-ci reçoit un **`Canvas`** en paramètre permettant de dessiner.

```
public class MyView extends View {

    public MyView(Context context) {
        this(context, null);
    }

    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    protected void onDraw(Canvas canvas) {

        // dessiner votre vue ici!
    }
}
```

- Les engins utilisent généralement une variante de vue permettant de dessiner à l'aide de OpenGL ES pour une performance supérieure.
- Par contre, la classe **`Canvas`** que nous utiliserons se sert elle-même de l'accélération du matériel sans nécessiter d'apprendre OpenGL ES.

- Le **`Canvas`** est créé pour vous, et représente une toile de dessin de la taille de votre vue pour que vous dessiniez.
- Comme pour n'importe quel peintre, pour dessiner sur votre toile, vous avez besoin de peinture, donc vous devrez créer des instances de **`Paint`**.
- **`Paint`** représente toutes les caractéristiques de trait, police et remplissage.

- Puisque vous devez éviter de créer de nouveaux objets dans le **`onDraw`** (pour la performance), il est préférable de créer votre peinture dans votre constructeur:


```
public class MyView extends View {

    private final Paint bluePaint = new Paint();

    public MyView(Context context) {
        this(context, null);
    }

    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);

        bluePaint.setColor(Color.BLUE);
        bluePaint.setStyle(Paint.Style.FILL);
    }

    @Override
    protected void onDraw(Canvas canvas) {

        // dessiner votre vue ici!
    }
}
```

- Vous pouvez maintenant vous servir de votre peinture dans le **`onDraw`** avec une des multiples méthodes **`draw*`** de la classe **`Canvas`**.

```
@Override
protected void onDraw(Canvas canvas) {

    canvas.drawCircle(100, 100, 100, bluePaint);
}
```

- Pour effectuer des calculs selon la taille de la vue, il est préférable de tous les effectuer en dehors du **`onDraw`** uniquement lorsque la taille de la vue change. Pour ceci, on utilise la méthode **`onSizeChanged`**:

```
private int cx, cy, radius;

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {

    cx = w / 2;
    cy = h / 2;

    radius = Math.min(cx, cy);
}

@Override
protected void onDraw(Canvas canvas) {

    canvas.drawCircle(cx, cy, radius, bluePaint);
}
```

- **`onSizeChanged`** est garantie d'être appelée à chaque changement de taille, incluant la taille initiale de la vue.

- **<u>Pour résumer</u>**:
  - Faites la majorité des calculs dans **`onSizeChanged`**.
  - Créer les objets dans votre constructeur.
  - **`onDraw`** devrait être le plus léger possible.

- Android n'appellera pas **`onDraw`** à chaque cycle de rafraîchissement comme vous vous attendez des engins.
- Puisque les appareils mobiles essaient d'utiliser le moins de batterie possible, Android n'appelle **`onDraw`** que si le contenu à dessiner a changé.
- Pour dire à Android que le contenu doit être redessiné, vous devez appeler **`invalidate()`**.

- **Attention!** **`invalidate()`** ne fait que dire à Android que la vue doit être redessinée. Une vue n'est pas immédiatement redessinée, elle doit attendre que Android effectue son prochain cycle de rafraîchissement.
- Il est même possible d'appeler plusieurs fois **`invalidate()`** dans avant d'avoir un seul appel à **`onDraw`**. Lors de ce **`onDraw`**, elle sera maintenant considérée à jour et tous les appels superflus à **`invalidate()`** seront ignorés.

- De façon générale, tout changement à une propriété utilisée dans l'affichage devrait inclure un appel à **invalidate()**:

```
public void setColor(int color) {

    paint.setColor(color);
    invalidate();
}
```

- Une telle propriété devrait également être exposée dans les fichiers de “*layout*” pour permettre de l’ajuster à l’utilisation.
- Pour faire ceci, votre projet doit inclure une ressource “*styleable*” représentant votre vue avec sa liste d’attributs. Généralement, nous mettons ceci dans un fichier **`attrs.xml`**.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="MyView">
        <attr name="color" format="color" />
    </declare-styleable>
</resources>
```

- Il faut alors modifier votre constructeur pour récupérer cet attribut:

```
private final Paint paint = new Paint();

public MyView(Context context) {
    this(context, null);
}

public MyView(Context context, AttributeSet attrs) {
    super(context, attrs);

    TypedArray a = context.getTheme().obtainStyledAttributes(attrs, R.styleable.MyView, 0, 0);
    try {
        paint.setColor(a.getColor(R.styleable.MyView_color, Color.BLUE));
    } finally {
        a.recycle();
    }
    paint.setStyle(Paint.Style.FILL);
}
```

- Vous pourrez alors ajouter ces attributs dans un fichier de “layout” en y ajoutant votre paquet comme préfix valide:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ca.bart.customview.MyView
        android:layout_width="300dp"
        android:layout_height="300dp"
        app:color="@android:color/holo_red_dark" />

</RelativeLayout>
```

Pour plus d’information, vous pouvez lire:

- Creating Custom Views
  https://developer.android.com/training/custom-views/index.html

- Custom Components
  https://developer.android.com/guide/topics/ui/custom-components.html