In-app Billing Version 3

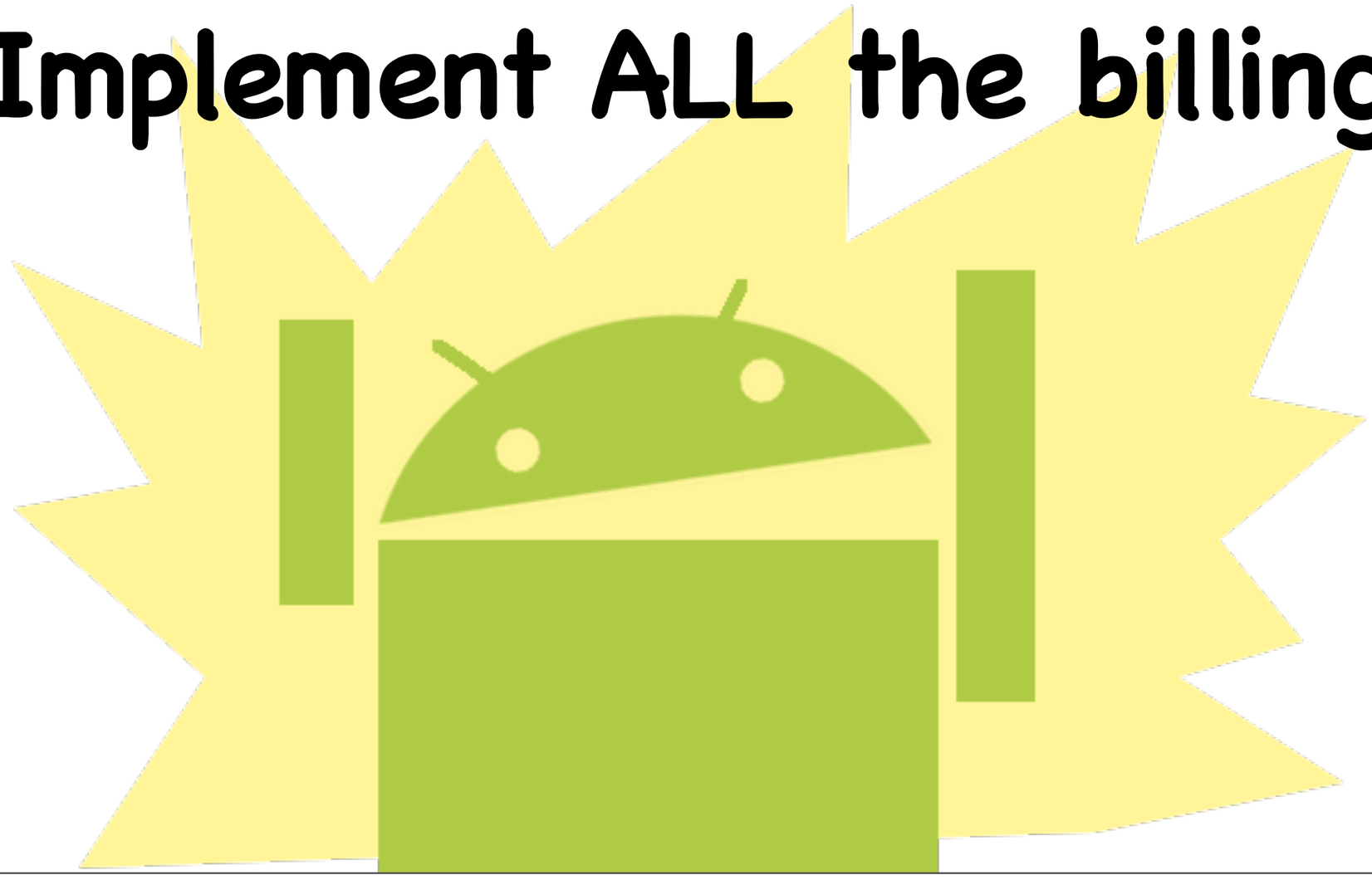
Bruno Oliveira
Developer Relations, Android

Google I/O 13

In-app billing!

In-app billing!

Implement ALL the billing!



In-app billing!

PREVIOUSLY IN

IN-APP BILLING

Easy

Easy

```
mBillingService.requestPurchase(mSku,
Consts.ITEM_TYPE_INAPP, mPayloadContents)
```

Easy

mBillingService.requestPurchase(mSku, Consts.ITEM_TYPE_INAPP, mPayloadContents)

```
public void onPurchaseStateChange(PurchaseState purchaseState, String itemId,
        int quantity, long purchaseTime, String developerPayload) {
    if (Consts.DEBUG) {
        Log.i(TAG, "onPurchaseStateChange() itemId: " + itemId + " " + purchaseState);
    }

    if (developerPayload == null) {
        logProductActivity(itemId, purchaseState.toString());
    } else {
        logProductActivity(itemId, purchaseState + "\n\t" + developerPayload);
    }

    if (purchaseState == PurchaseState.PURCHASED) {
        mOwnedItems.add(itemId);

        // If this is a subscription, then enable the "Edit
        // Subscriptions" button.
        for (CatalogEntry e : CATALOG) {
            if (e.sku.equals(itemId) &&
                    e.managed.equals(Managed.SUBSCRIPTION)) {
                mEditSubscriptionsButton.setVisibility(View.VISIBLE);
            }
        }
    }
    mCatalogAdapter.setOwnedItems(mOwnedItems);
    mOwnedItemsCursor.requery();
}
```

```
 * change. The signedData parameter is a plaintext JSON string that is
 * signed by the server with the developer's private key. The signature
 * for the signed data is passed in the signature parameter.
 * @param context the context
 * @param signedData the (unencrypted) JSON string
 * @param signature the signature for the signedData
 */
private void purchaseStateChanged(Context context, String signedData, String signature) {
    Intent intent = new Intent(Consts.ACTION_PURCHASE_STATE_CHANGED);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_SIGNED_DATA, signedData);
    intent.putExtra(Consts.INAPP_SIGNATURE, signature);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a "notify" message  indicating that transaction
 * information is available. The request includes a nonce (random number used once) that
 * we generate and Android Market signs and sends back to us with the purchase state and
 * other transaction details. This BroadcastReceiver cannot bind to the
 * MarketBillingService directly so it starts the {@link BillingService}, which does the
 * actual work of sending the message.
 *
 * @param context the context
 * @param notifyId the notification ID
 */
private void notify(Context context, String notifyId) {
    Intent intent = new Intent(Consts.ACTION_GET_PURCHASE_INFORMATION);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.NOTIFICATION_ID, notifyId);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a server response code. The BillingService can
 * then report the status of the response if desired.
 *
 * @param context the context
 * @param requestId the request ID that corresponds to a previous request
 * @param responseCodeIndex the ResponseCode ordinal value for the request
 */
private void checkResponseCode(Context context, long requestId, int responseCodeIndex) {
    Intent intent = new Intent(Consts.ACTION_RESPONSE_CODE);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_REQUEST_ID, requestId);
    intent.putExtra(Consts.INAPP_RESPONSE_CODE, responseCodeIndex);
    context.startService(intent);
}
```

Easy

mBillingService.requestPurchase(mSku, Consts.ITEM_TYPE_INAPP, mPayloadContents)

```
 * change. The signedData parameter is a plaintext JSON string that is
 * signed by the server with the developer's private key. The signature
 * for the signed data is passed in the signature parameter.
 * @param context the context
 * @param signedData the (unencrypted) JSON string
 * @param signature the signature for the signedData
 */
private void purchaseStateChanged(Context context, String signedData, String signature) {
    Intent intent = new Intent(Consts.ACTION_PURCHASE_STATE_CHANGED);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_SIGNED_DATA, signedData);
    intent.putExtra(Consts.INAPP_SIGNATURE, signature);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a "notify" message  indicating that transaction
 * information is available. The request includes a nonce (random number used once) that
 * we generate and Android Market signs and sends back to us with the purchase state and
 * other transaction details. This BroadcastReceiver cannot bind to the
 * MarketBillingService directly so it starts the {@link BillingService}, which does the
 * actual work of sending the message.
 *
 * @param context the context
 * @param notifyId the notification ID
 */
private void notify(Context context, String notifyId) {
    Intent intent = new Intent(Consts.ACTION_GET_PURCHASE_INFORMATION);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.NOTIFICATION_ID, notifyId);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a server response code. The BillingService can
 * then report the status of the response if desired.
 *
 * @param context the context
 * @param requestId the request ID that corresponds to a previous request
 * @param responseCodeIndex the ResponseCode ordinal value for the request
 */
private void checkResponseCode(Context context, long requestId, int responseCodeIndex) {
    Intent intent = new Intent(Consts.ACTION_RESPONSE_CODE);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_REQUEST_ID, requestId);
    intent.putExtra(Consts.INAPP_RESPONSE_CODE, responseCodeIndex);
    context.startService(intent);
}
```

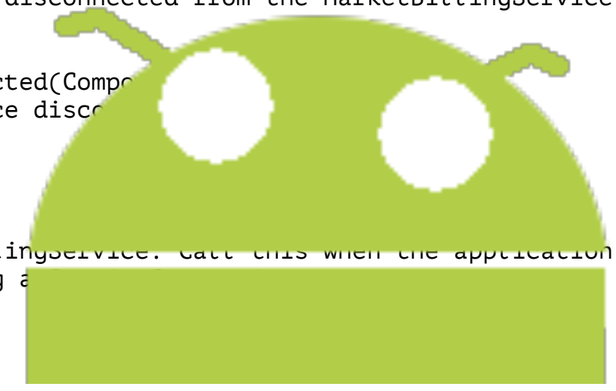
```
                maxStartId = request.getStartId();
            }
        } else {
            // The service crashed, so restart it. Note that this leaves
            // the current request on the queue
            bindToMarketBillingService();
            return;
        }
    }

    // If we get here then all the requests ran successfully.  If maxStartId
    // is not -1, then one of the requests started the service, so we can
    // stop it now.
    if (maxStartId >= 0) {
        if (Consts.DEBUG) {
            Log.i(TAG, "stopping service, startId: " + maxStartId);
        }
        stopSelf(maxStartId);
    }
}

/**
 * This is called when we are connected to the MarketBillingService.
 * This runs in the main UI thread.
 */
@Override
public void onServiceConnected(ComponentName name, IBinder service) {
    if (Consts.DEBUG) {
        Log.d(TAG, "Billing service connected");
    }
    mService = IMarketBillingService.Stub.asInterface(service);
    runPendingRequests();
}

/**
 * This is called when we are disconnected from the MarketBillingService.
 */
@Override
public void onServiceDisconnected(ComponentName name) {
    Log.w(TAG, "Billing service disconnected");
    mService = null;
}

/**
 * Unbinds from the MarketBillingService. Call this when the application
 * terminates to avoid leaking a ServiceConnection.
 */
public void unbind() {
    try {
        unbindService(this);
    } catch (IllegalArgumentException e) {
        // This might happen if the service was disconnected
```

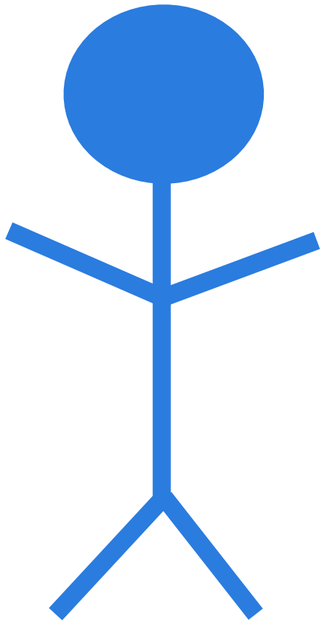
```
 * change. The signedData parameter is a plaintext JSON string that is
 * signed by the server with the developer's private key. The signature
 * for the signed data is passed in the signature parameter.
 * @param context the context
 * @param signedData the (unencrypted) JSON string
 * @param signature the signature for the signedData
 */
private void purchaseStateChanged(Context context, String signedData, String signature) {
    Intent intent = new Intent(Consts.ACTION_PURCHASE_STATE_CHANGED);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_SIGNED_DATA, signedData);
    intent.putExtra(Consts.INAPP_SIGNATURE, signature);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a "notify" message  indicating that transaction
 * information is available. The request includes a nonce (random number used once) that
 * we generate and Android Market signs and sends back to us with the purchase state and
 * other transaction details. This BroadcastReceiver cannot bind to the
 * MarketBillingService directly so it starts the {@link BillingService}, which does the
 * actual work of sending the message.
 *
 * @param context the context
 * @param notifyId the notification ID
 */
private void notify(Context context, String notifyId) {
    Intent intent = new Intent(Consts.ACTION_GET_PURCHASE_INFORMATION);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.NOTIFICATION_ID, notifyId);
    context.startService(intent);
}

/**
 * This is called when Android Market sends a server response code. The BillingService can
 * then report the status of the response if desired.
 *
 * @param context the context
 * @param requestId the request ID that corresponds to a previous request
 * @param responseCodeIndex the ResponseCode ordinal value for the request
 */
private void checkResponseCode(Context context, long requestId, int responseCodeIndex) {
    Intent intent = new Intent(Consts.ACTION_RESPONSE_CODE);
    intent.setClass(context, BillingService.class);
    intent.putExtra(Consts.INAPP_REQUEST_ID, requestId);
    intent.putExtra(Consts.INAPP_RESPONSE_CODE, responseCodeIndex);
    context.startService(intent);
}
```

```
    public Cursor queryAllPurchasedItems() {
        return mDb.query(PURCHASED_ITEMS_TABLE_NAME, PURCHASED_COLUMNS, null,
                null, null, null, null);
    }

    /**
     * This is a standard helper class for constructing the database.
     */
    private class DatabaseHelper extends SQLiteOpenHelper {
        public DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            createPurchaseTable(db);
            createPurchasedItemsTable(db);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            // Production-quality upgrade code should modify the tables when
            // the database version changes instead of dropping the tables and
            // re-creating them.
            if (newVersion != DATABASE_VERSION) {
                Log.w(TAG, "Database upgrade from old: " + oldVersion + " to " +
                        newVersion);
                db.execSQL("DROP TABLE IF EXISTS " + PURCHASE_HISTORY_TABLE_NAME);
                db.execSQL("DROP TABLE IF EXISTS " + PURCHASED_ITEMS_TABLE_NAME);
                createPurchaseTable(db);
                return;
            }
        }

        private void createPurchaseTable(SQLiteDatabase db) {
            db.execSQL("CREATE TABLE " + PURCHASE_HISTORY_TABLE_NAME + "(" +
                    HISTORY_ORDER_ID_COL + " TEXT PRIMARY KEY, " +
                    HISTORY_STATE_COL + " INTEGER, " +
                    HISTORY_PRODUCT_ID_COL + " TEXT, " +
                    HISTORY_DEVELOPER_PAYLOAD_COL + " TEXT, " +
                    HISTORY_PURCHASE_TIME_COL + " INTEGER)");
            db.execSQL("CREATE TABLE " + PURCHASED_ITEMS_TABLE_NAME + "(" +
                    PURCHASED_PRODUCT_ID_COL + " TEXT PRIMARY KEY, " +
                    PURCHASED_QUANTITY_COL + " INTEGER) ");
        }
    }
```

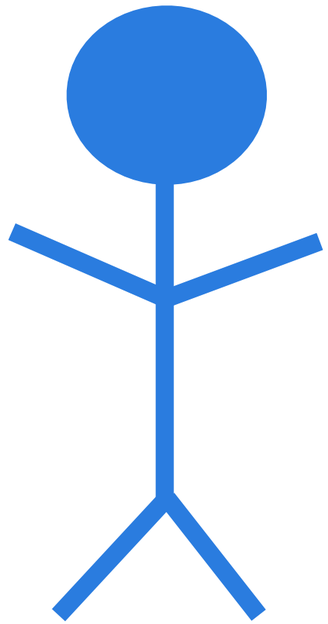
```
        maxStartId = request.getStartId();
    }
} else {
    // The service crashed, so restart it. Note that this leaves
    // the current request on the queue
    bindToMarketBillingService();
    return;
}

// If we get here then all the requests ran successfully.  If maxStartId
// is not -1, then one of the requests started the service, so we can
// stop it now.
if (maxStartId >= 0) {
    if (Consts.DEBUG) {
        Log.i(TAG, "stopping service, startId: " + maxStartId);
    }
    stopSelf(maxStartId);
}
}

/**
 * This is called when we are connected to the MarketBillingService.
 * This runs in the main UI thread.
 */
@Override
public void onServiceConnected(ComponentName name, IBinder service) {
    if (Consts.DEBUG) {
        Log.d(TAG, "Billing service connected");
    }
    mService = IMarketBillingService.Stub.asInterface(service);
    runPendingRequests();
}

/**
 * This is called when we are disconnected from the MarketBillingService.
 */
@Override
public void onServiceDisconnected(ComponentName name) {
    Log.w(TAG, "Billing service disconnected");
    mService = null;
}

/**
 * Unbinds from the MarketBillingService. Call this when the application
 * terminates to avoid leaking a ServiceConnection.
 */
public void unbind() {
    try {
        unbindService(this);
    } catch (IllegalArgumentException e) {
        // This might happen if the service was disconnected
```

Easy why does it get so complicated?



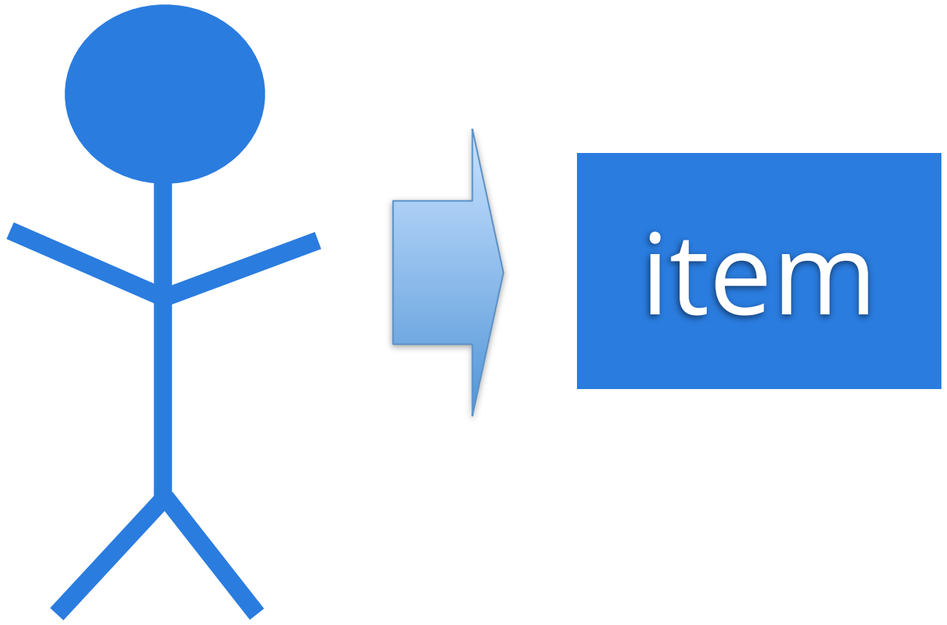


Monday, May 20,

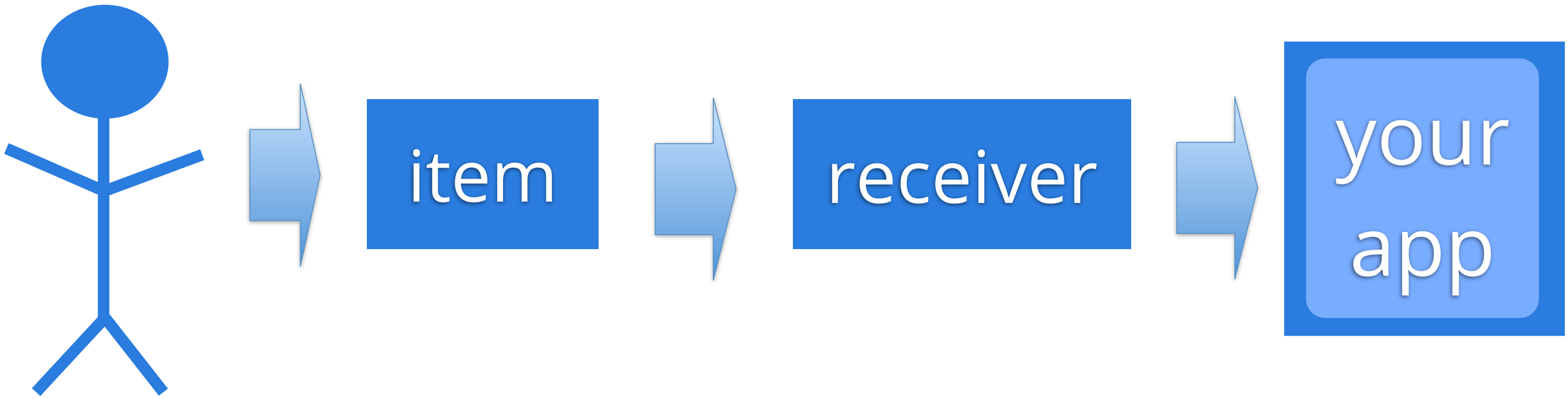


item

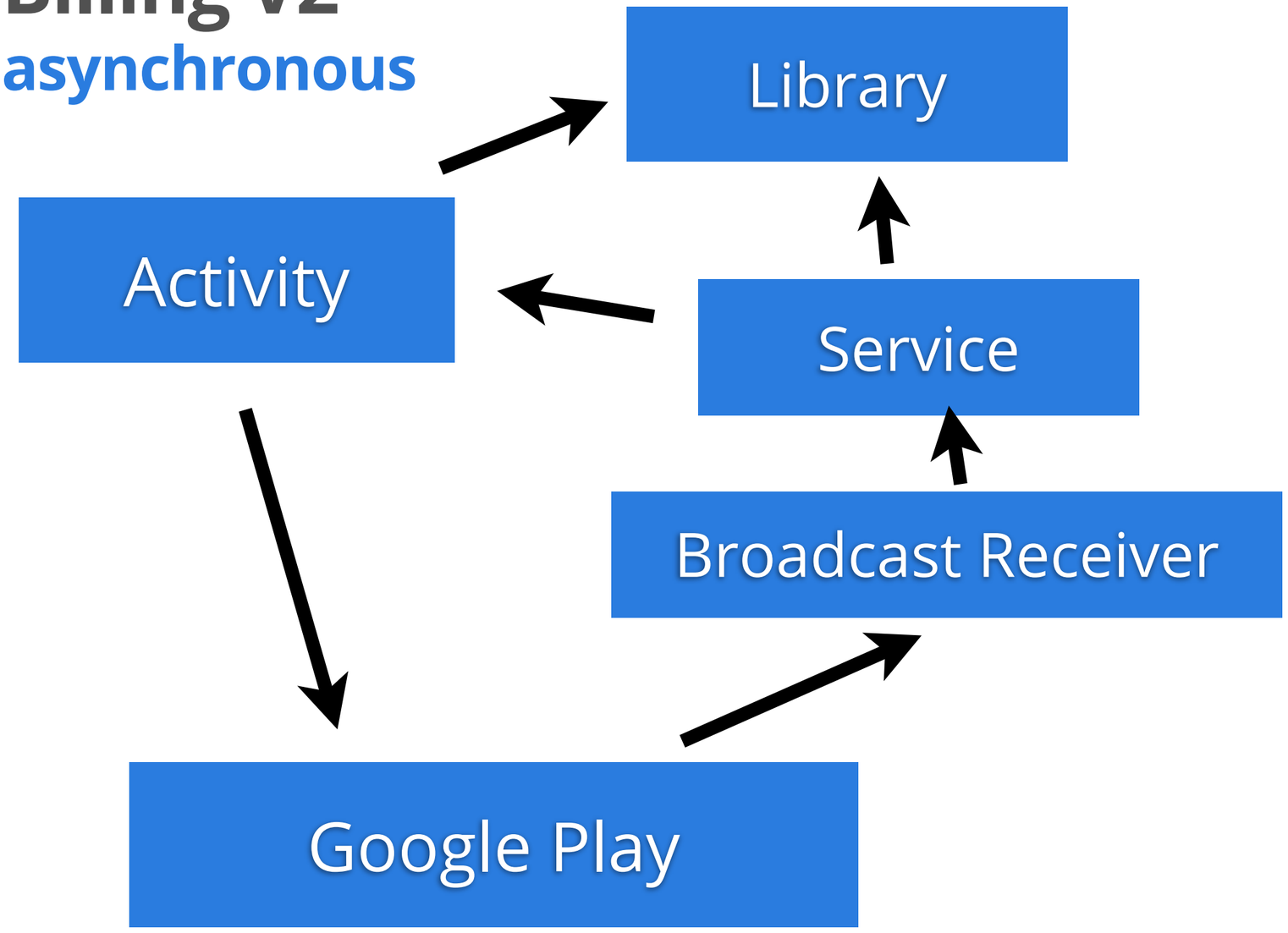

item
your app


item


item
receiver

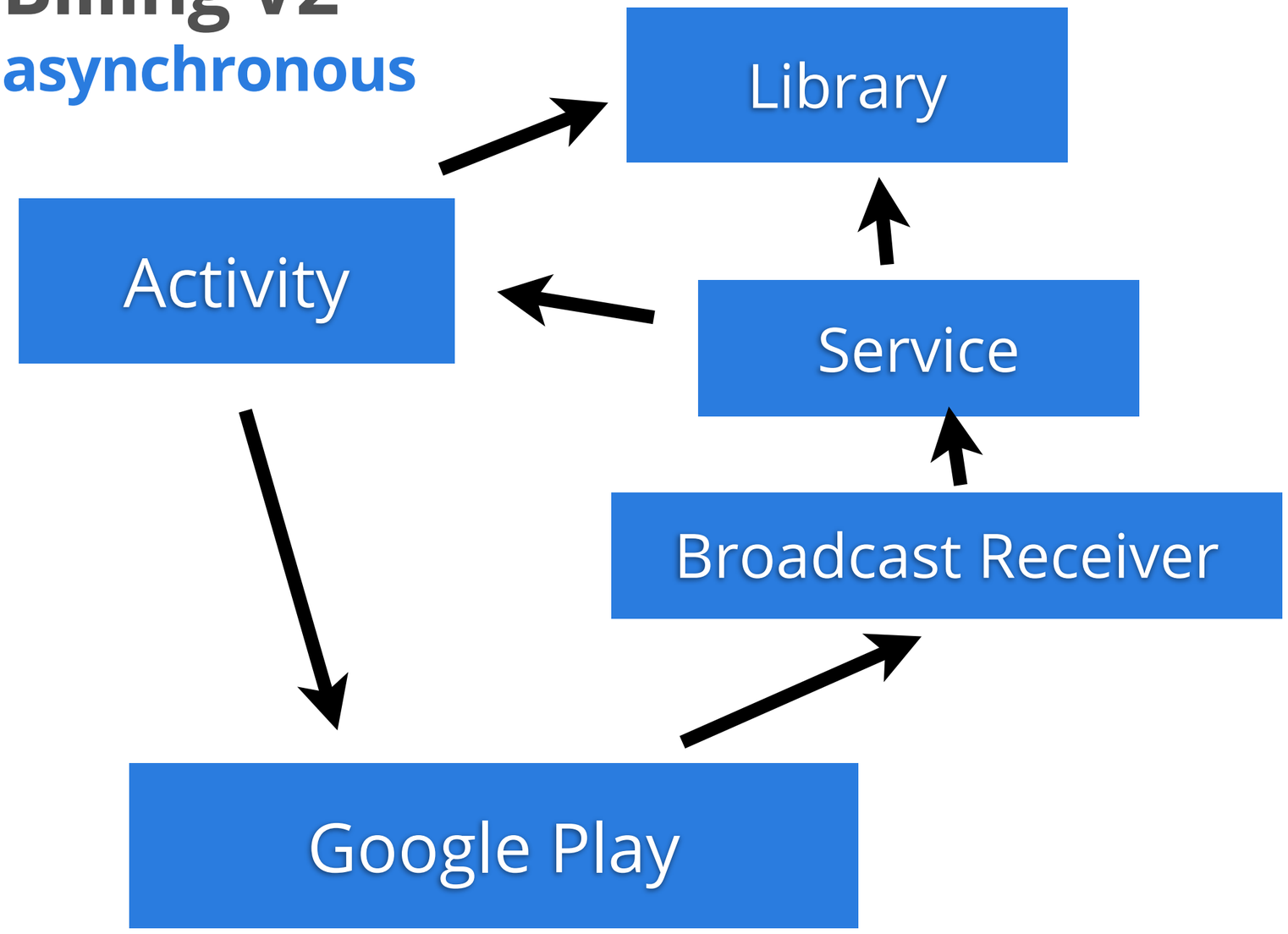

item
receiver
your app

Billing V2
asynchronous



Billing V2

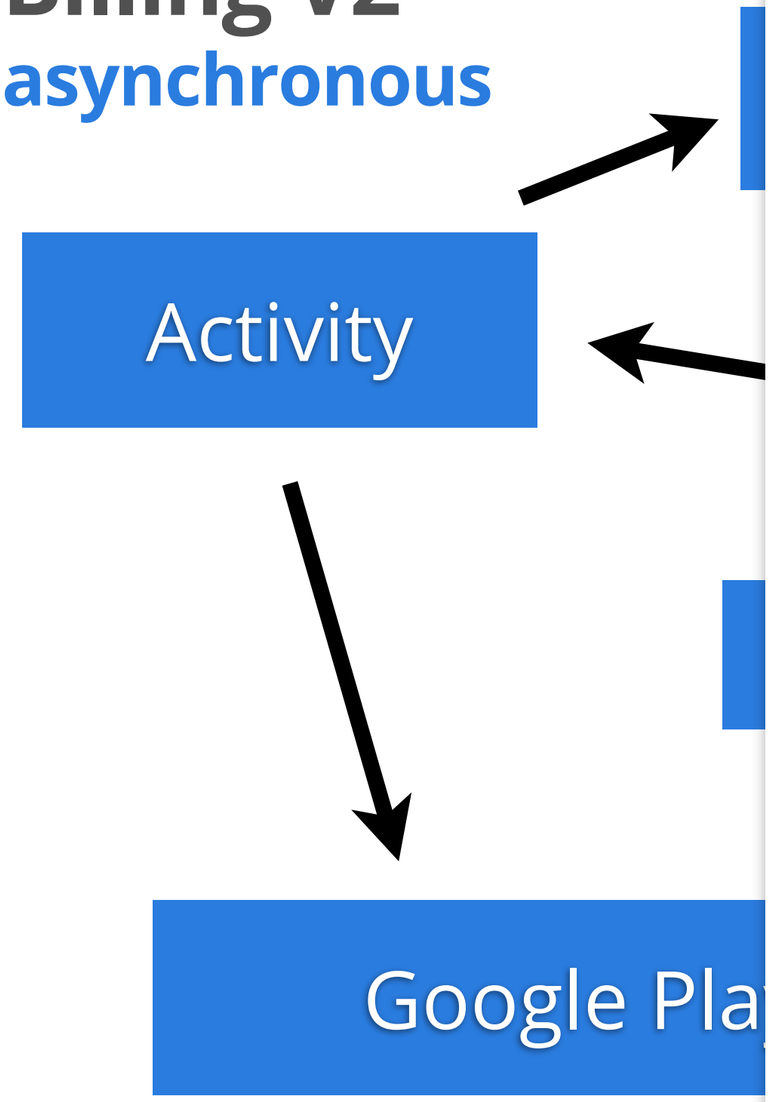
asynchronous



Billing V3

Billing V2

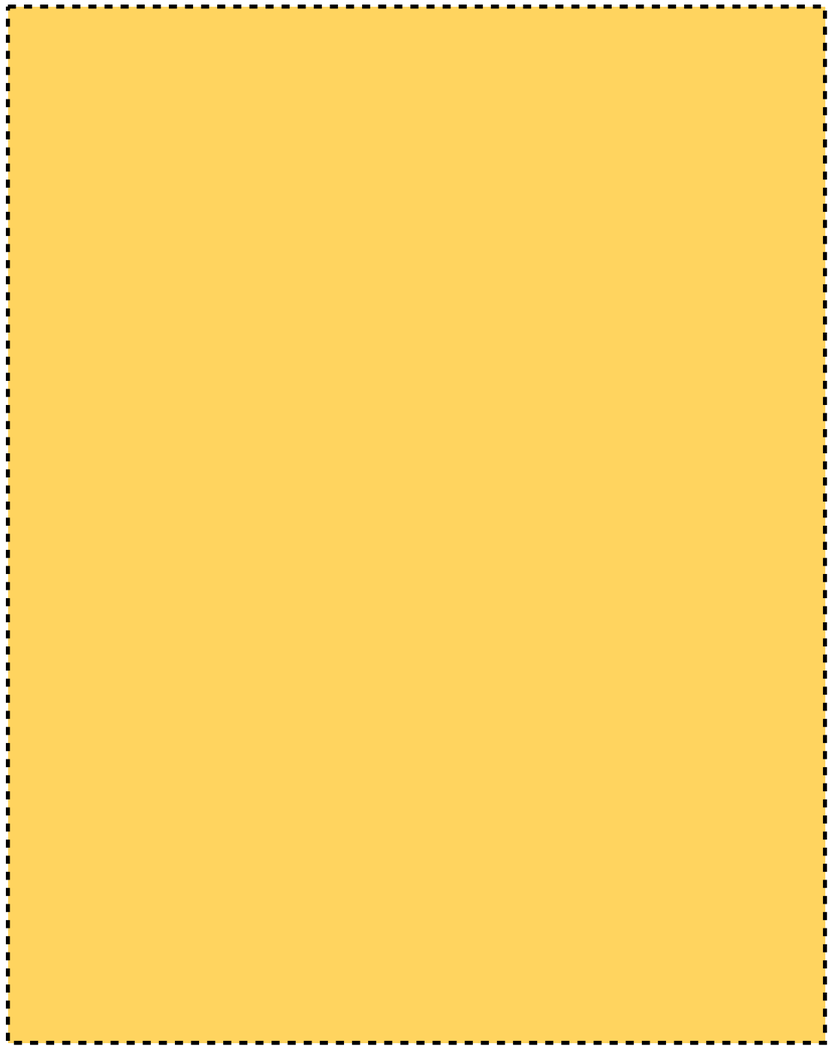
asynchronous

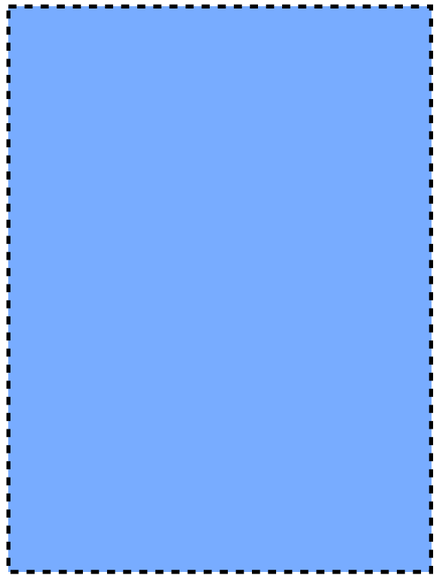


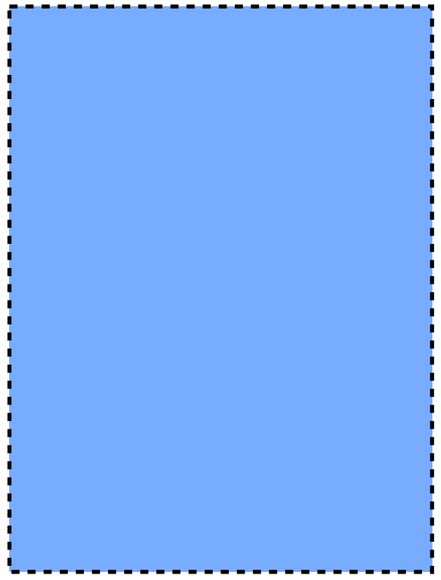
Billing V3

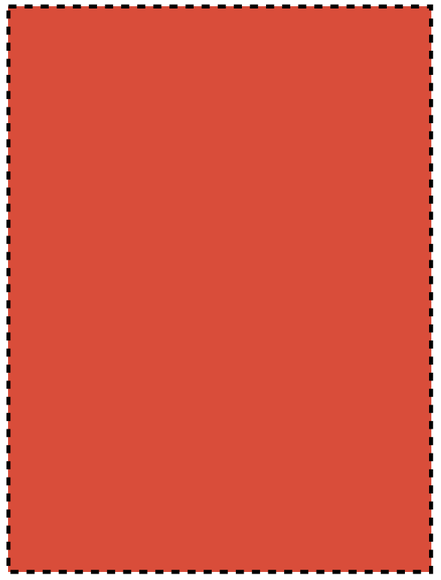
Billing V3

Billing V3

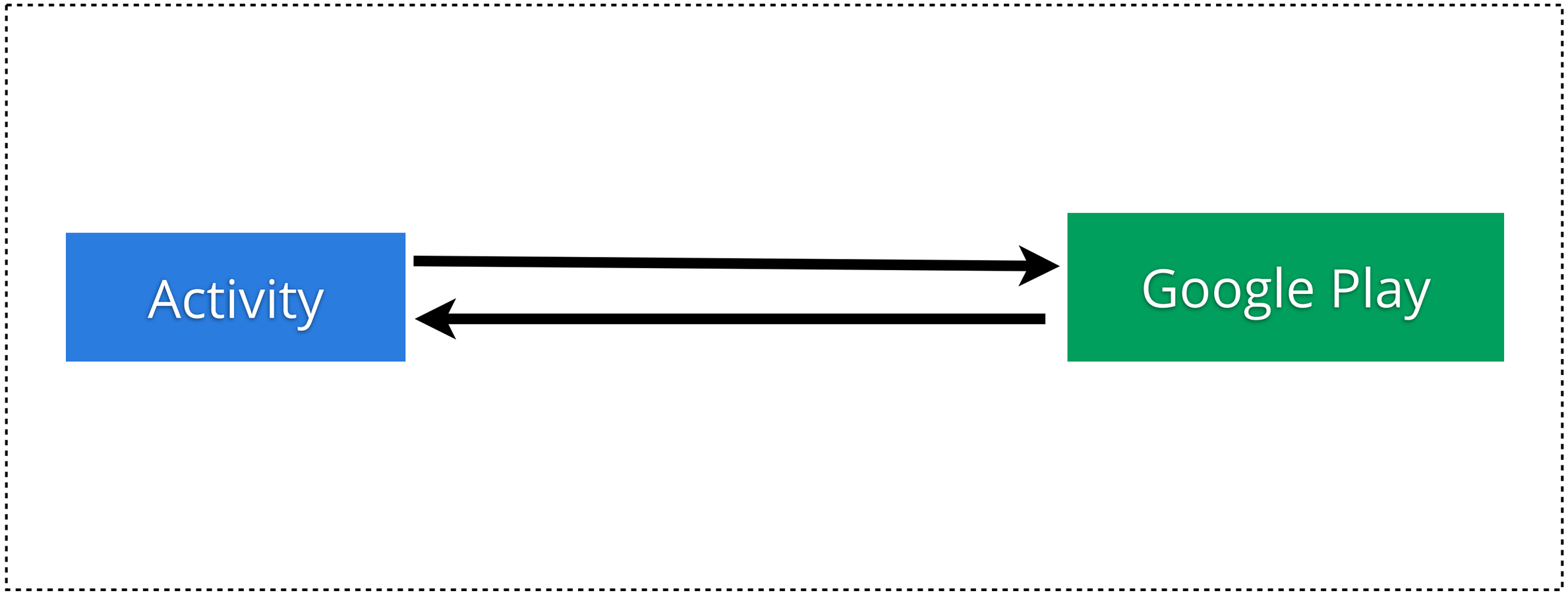




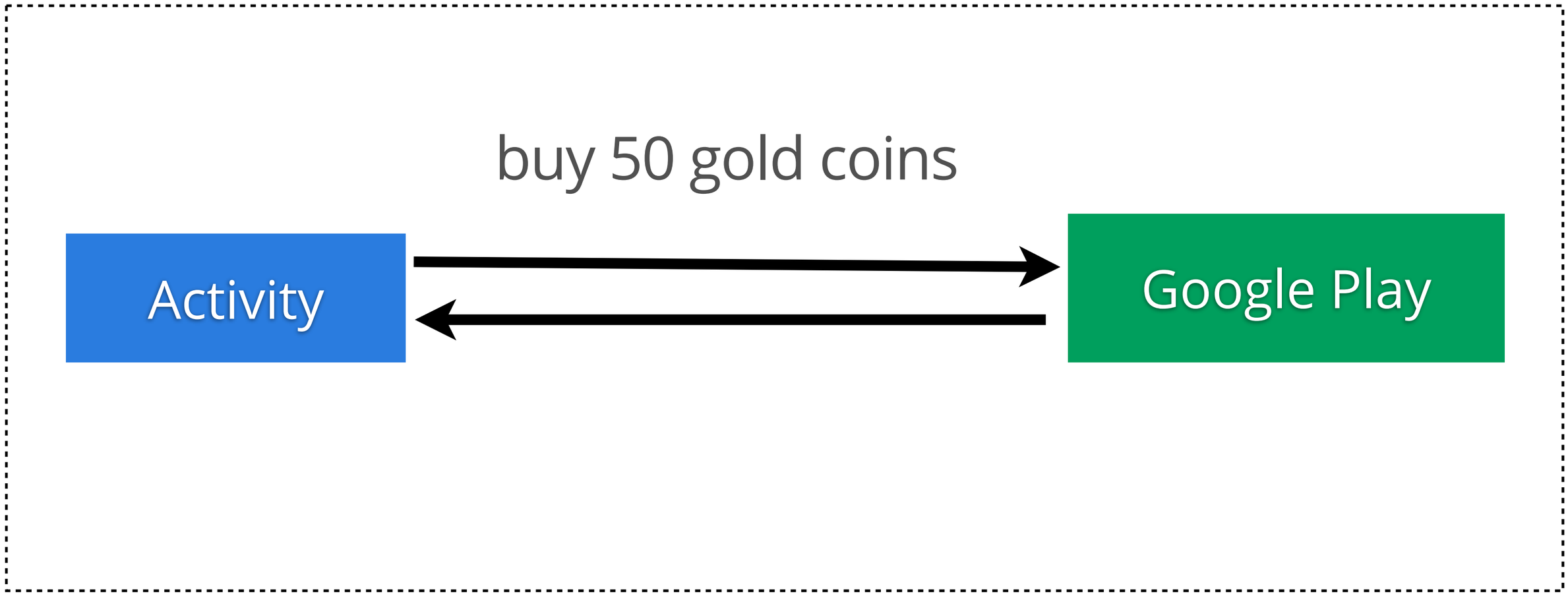




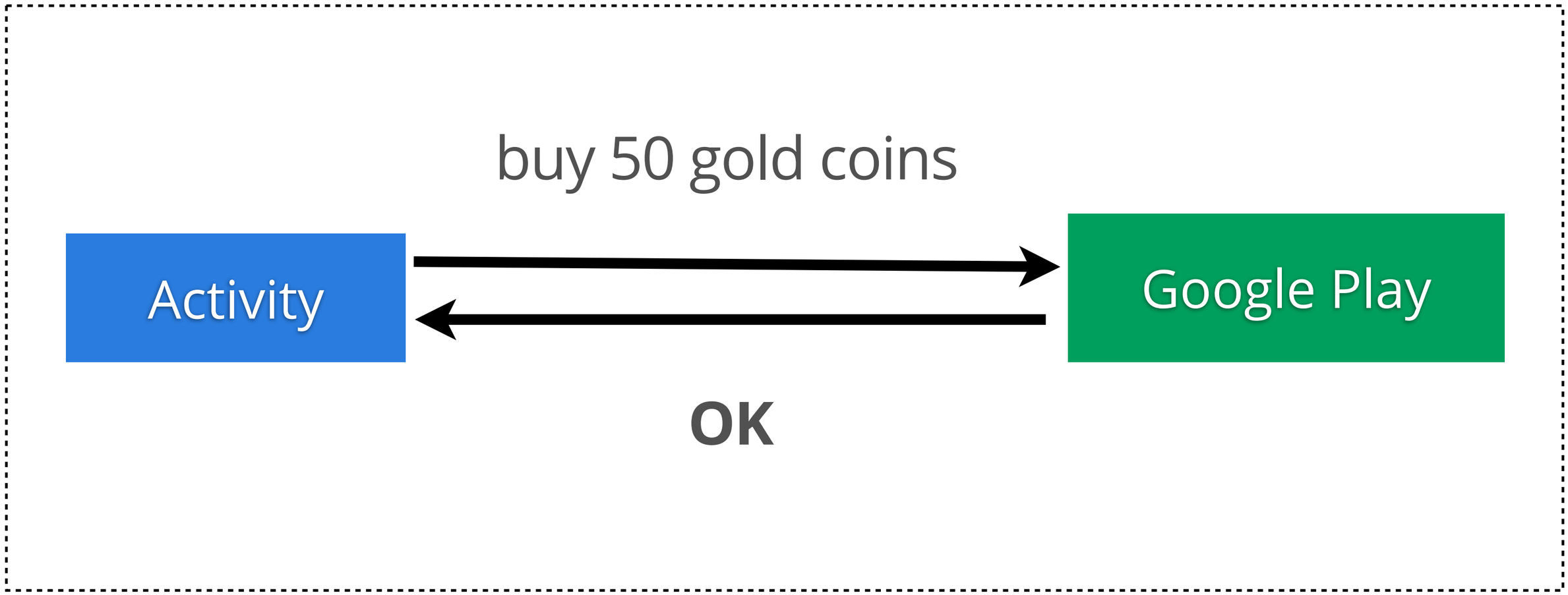
Billing V3 is synchronous



Billing V3 is synchronous



Billing V3 is synchronous



In v2, restoring purchases was expensive

My App
123 SD Card Folder
John's Phone, 12345

Pay to the order of:

Billing V Two

CPU Cycles: many **x 10**

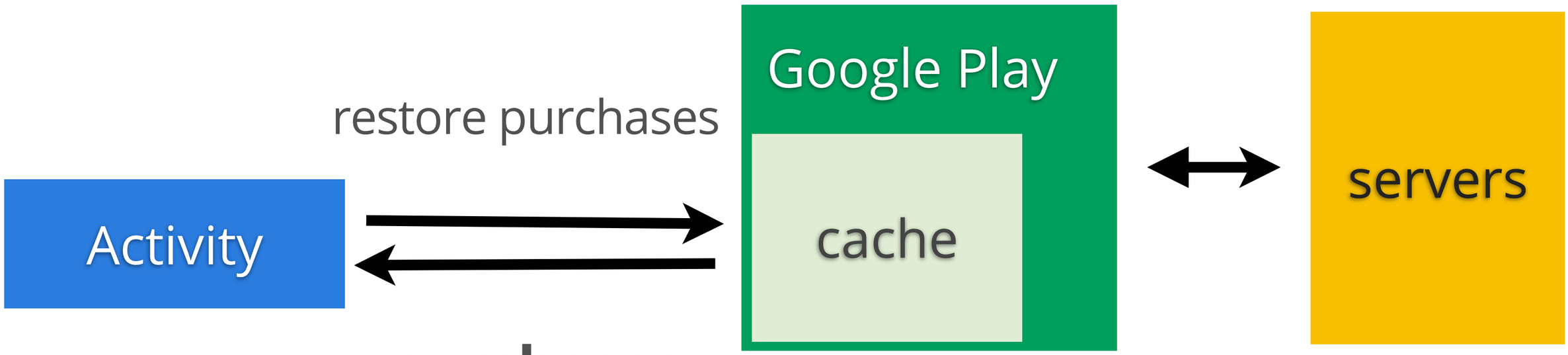
Traffic: lots **( ) MB ( ) GB ( ) TB**

for: restoring purchases

John D. Veloper

:39209382: :109830913: 39813800::

In v3, Google Play maintains a client-side cache



so...

...in v3, restoring purchases is pretty cheap

CPU & NETWORK

99%

OFF

ON ANY GET-PURCHASES CALL W/

IN-APP BILLING VERSION 3

(THROUGH CLIENT-SIDE CACHING)

AVAILABLE ON PARTICIPATING DEVICES.

ENOUGH SALES.
SHOW ME THE
CODE.

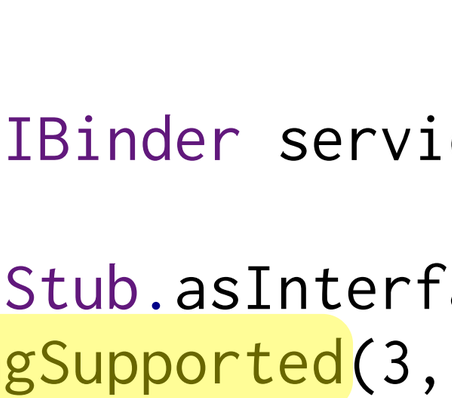
</sales>

Selling Stuff

Is V3 is supported?

```
public void onServiceConnected(
                ComponentName name, IBinder service) {

    mService = IInAppBillingService.Stub.asInterface(service);
    int response = mService.isBillingSupported(3,
                getPackageName(), “inapp”);
    if (response == BILLING_RESPONSE_RESULT_OK) {
        // has billing!
    }
    else {
        // no billing V3...
    }
}
```

```
public void onServiceConnected(
                ComponentName name, IBinder service) {

    mService = IInAppBillingService.Stub.asInterface(service);
    int response = mService.isBillingSupported(3,
                getPackageName(), “inapp”);
    if (response == BILLING_RESPONSE_RESULT_OK) {
        // has billing!
    }
    else {
        // no billing V3...
    }
}
```



+

3.9.16+

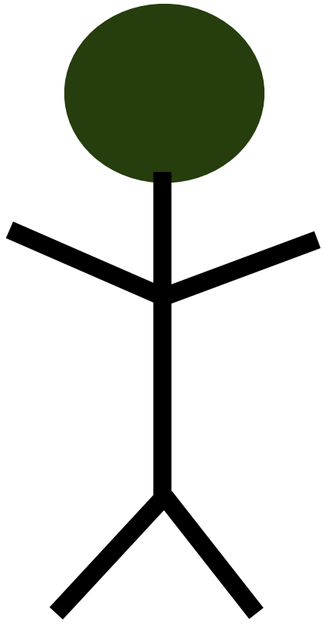
90%+ of active devices

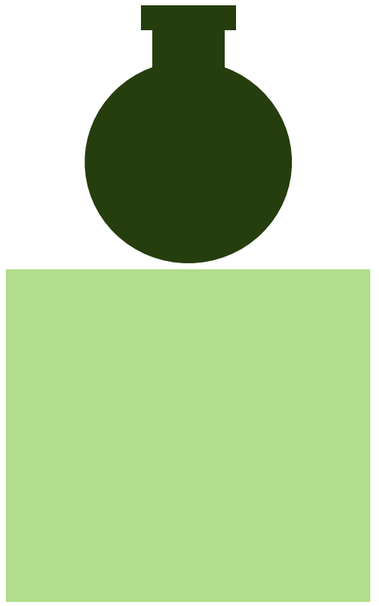
What does the user own?

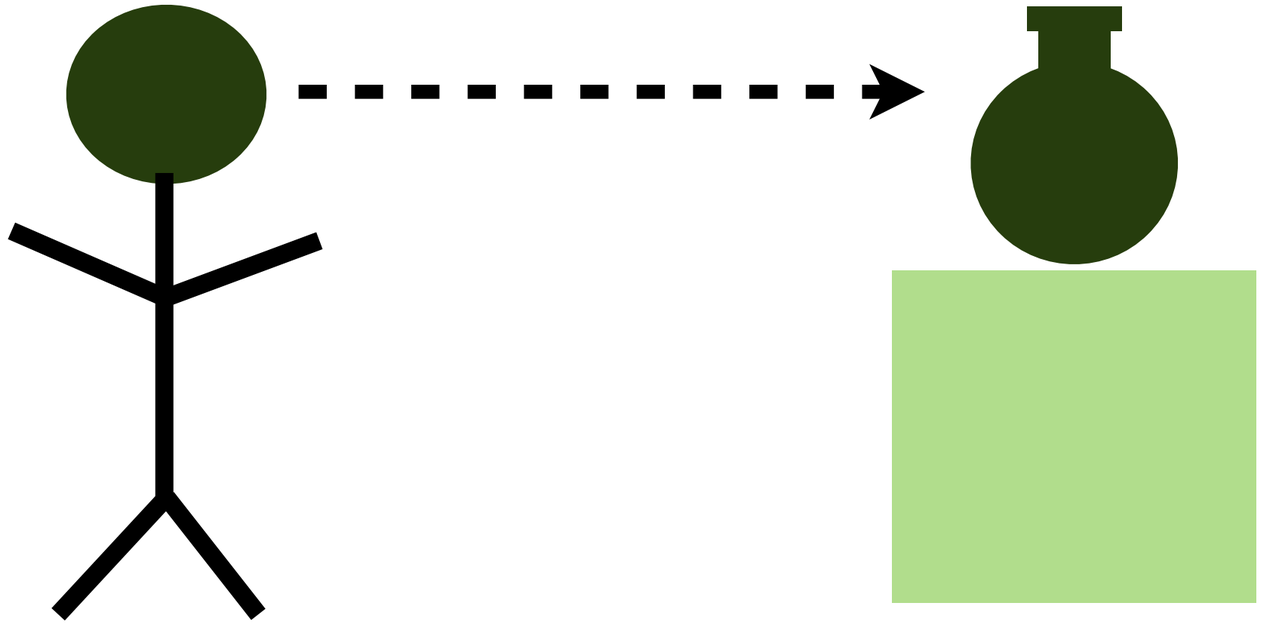
```
Bundle bundle = mService.getPurchases(
        3, mContext.getPackageName(), “inapp”);

if (bundle.getInt(RESPONSE_CODE) == BILLING_RESPONSE_RESULT_OK) {
    ArrayList mySkus, myPurchases, mySignatures;
    mySkus = bundle.getStringArrayList(RESPONSE_INAPP_ITEM_LIST);
    myPurchases bundle.getStringArrayList(
                        RESPONSE_INAPP_PURCHASE_DATA_LIST);
    mySignatures = bundle.getStringArrayList(
                        RESPONSE_INAPP_PURCHASE_SIGNATURE_LIST);

    // handle items here
}
```

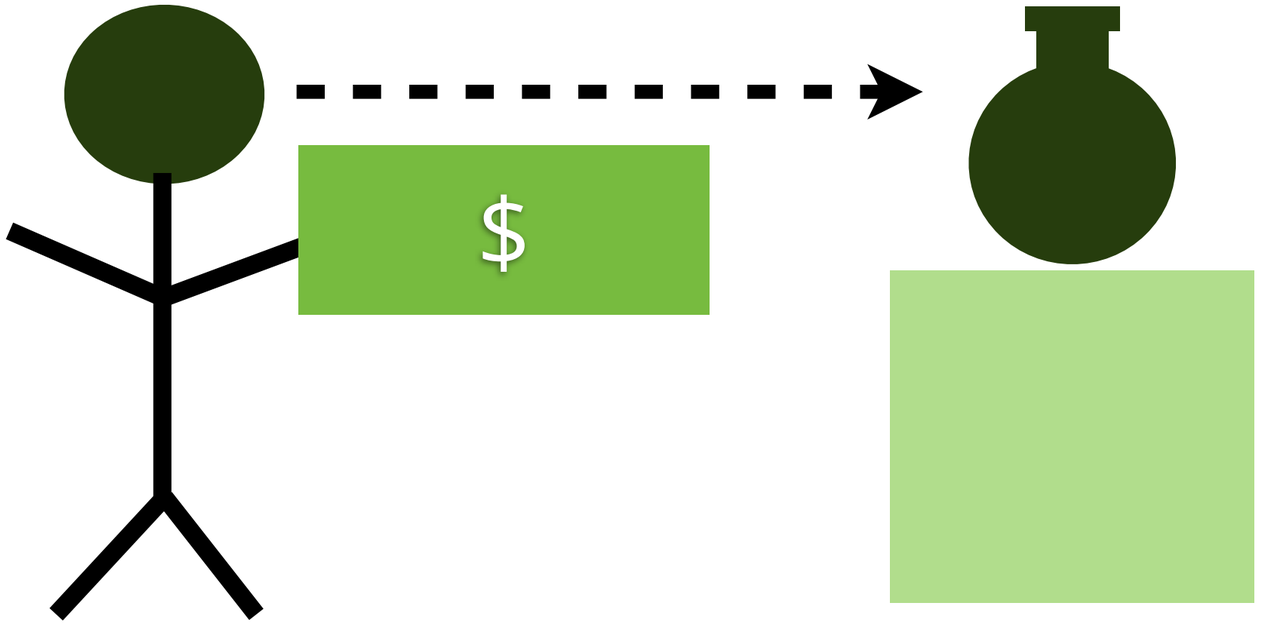
CHEAP

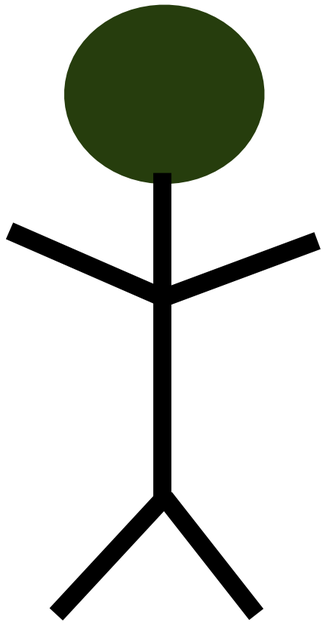
SYNCHRONOUS!

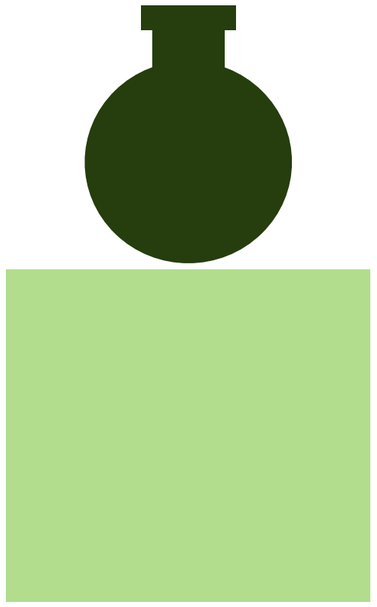


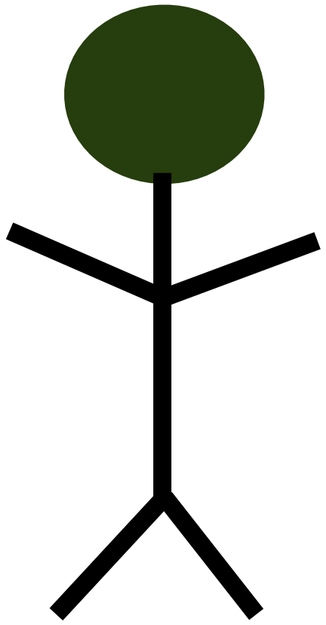


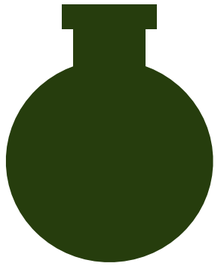


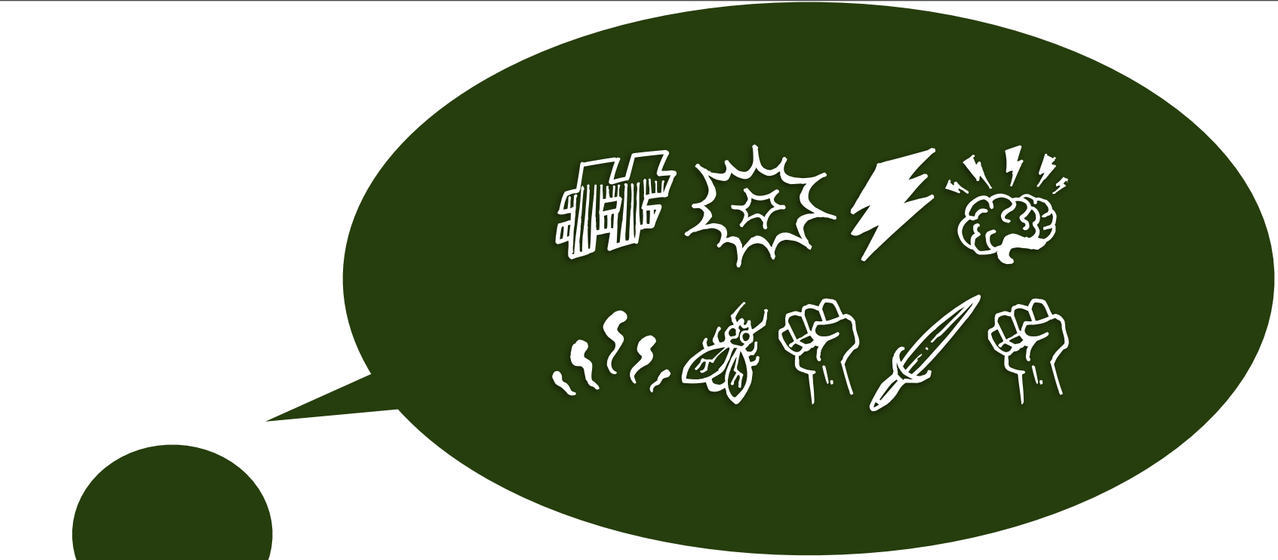


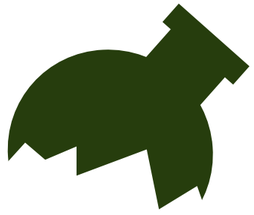




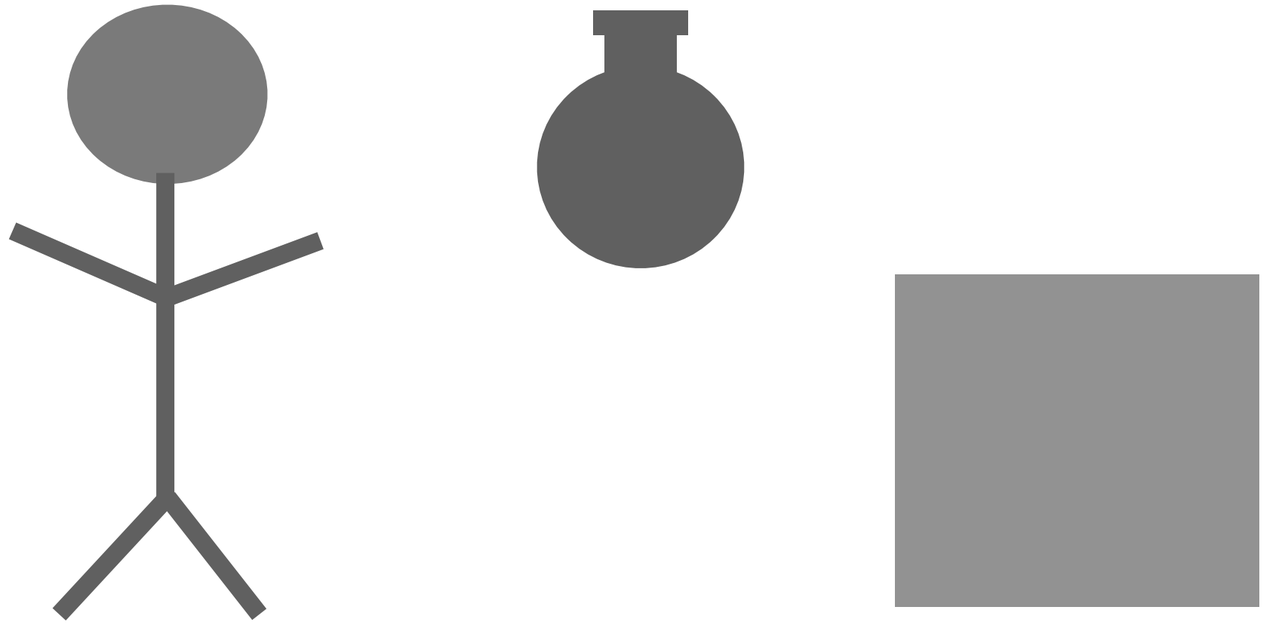




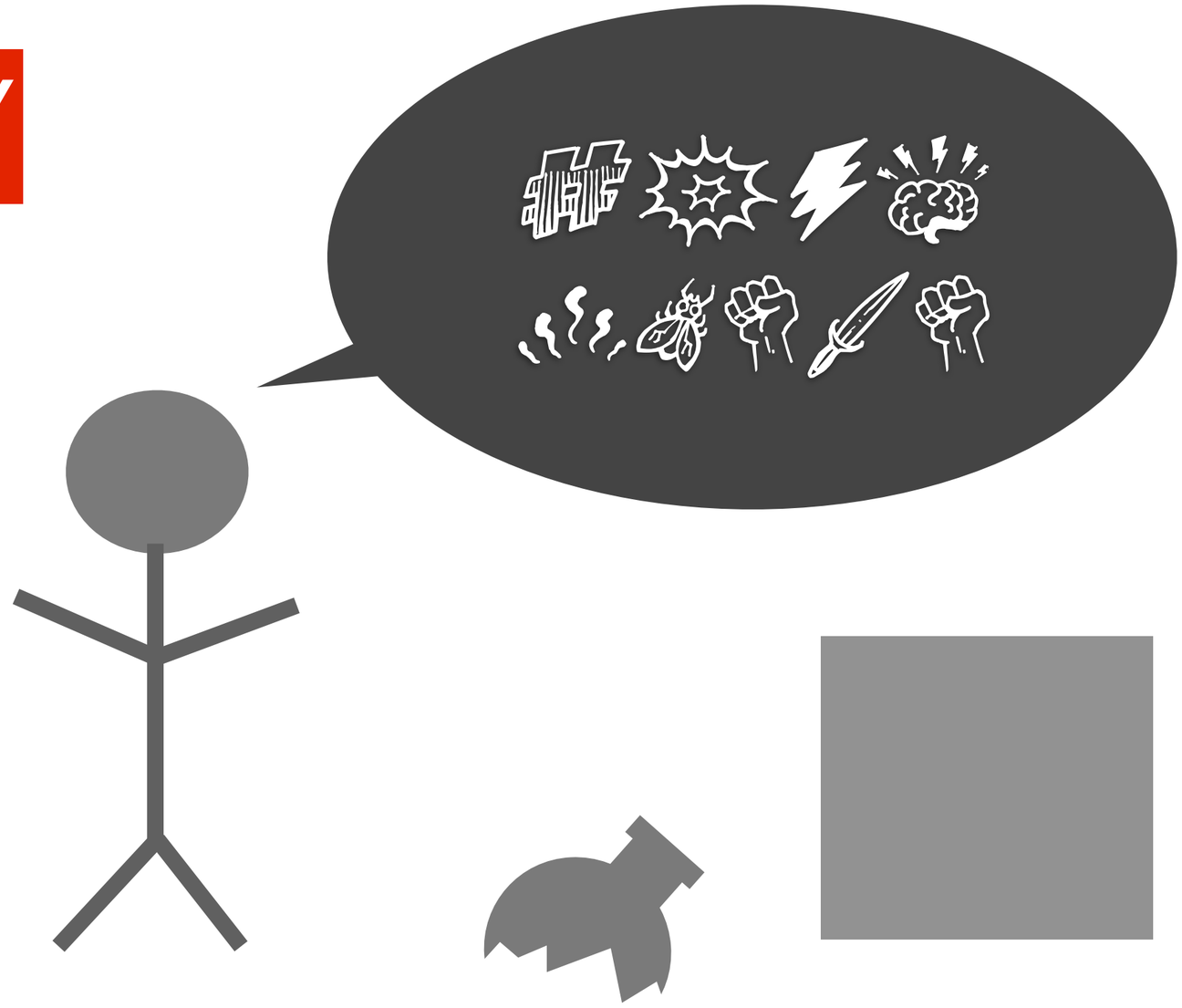




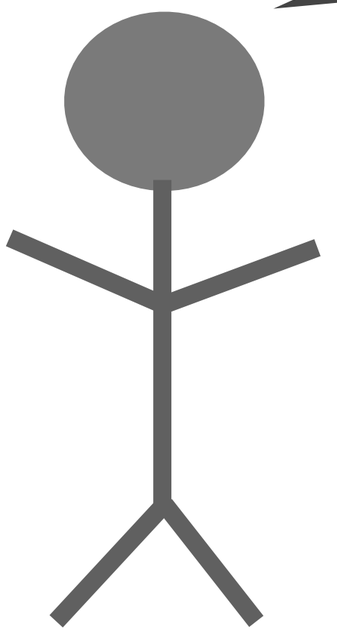
REPLAY

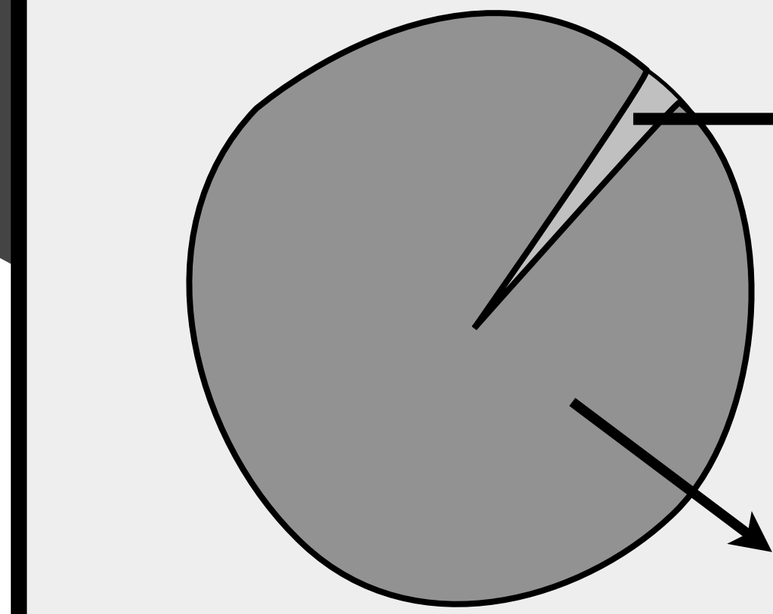


REPLAY



REPLAY




PROBABILITY THAT THIS USER IS...
THRILLED
AN INFLUENTIAL BLOGGER
87% OF ALL PERCENTAGES IN THIS SLIDE MAY BE MADE UP.

~~unmanaged items~~

managed items

Launch the purchase flow

Launch the purchase flow

```
Bundle bundle = mService.getBuyIntent(3, "com.example.myapp",
    MY_SKU, “inapp”, developerPayload);

PendingIntent pendingIntent =
    bundle.getParcelable(RESPONSE_BUY_INTENT);

if (bundle.getInt(RESPONSE_CODE) ==
                    BILLING_RESPONSE_RESULT_OK) {
    startIntentSenderForResult(pendingIntent, RC_BUY,
        new Intent(), Integer.valueOf(0), Integer.valueOf(0),
        Integer.valueOf(0));
}
```

RESULT COMES BACK ON
onActivityResult()

SINGLE PLAYER

1,000 coins (Nostalgic Racer) $0.99

MasterCard- 0541

Google play

BUY

Sign in

Sign in with Google to play with friends and share your progress.

Monday, May 20,

onActivityResult()

```
public void onActivityResult(int requestCode,
                             int resultCode, Intent data) {

    if (requestCode == RC_BUY) {
        int responseCode = data.getIntExtra(RESPONSE_CODE);
        String purchaseData = data.getStringExtra(
            RESPONSE_INAPP_PURCHASE_DATA);
        String signature = data.getStringExtra(
            RESPONSE_INAPP_SIGNATURE);

        // ...
```

Purchase data

```
{
    “orderId”: ...,
    “packageName”: ...,
    “productId”: ...,
    “purchaseTime”: ...,
    “purchaseState”: ...,
    “developerPayload”, ...,
    “purchaseToken”: ...
}
```

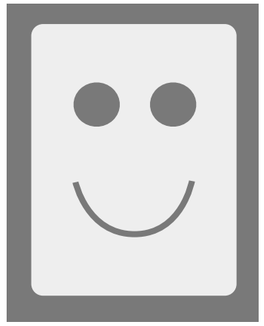
HARD TO LOSE
THE PURCHASE

That's it.... right?

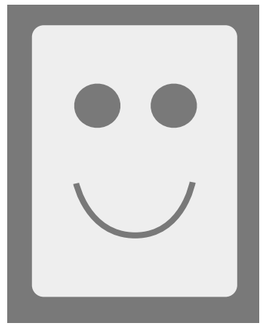
on startup:
getPurchases()

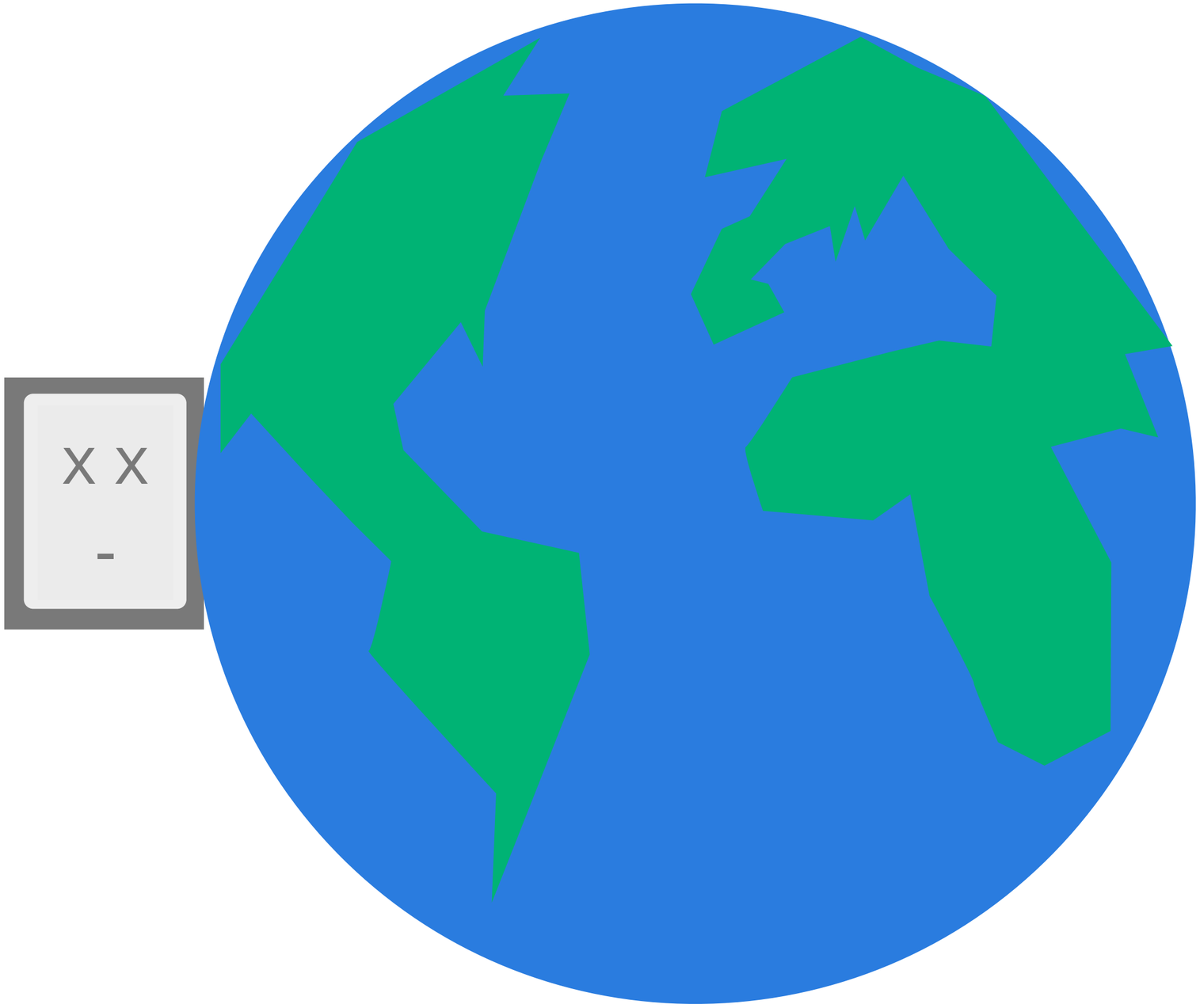
when user wants to purchase:
getBuyIntent(), launch the Intent.

onActivityResult:
handle purchase



Monday, May 20,





** not to scale.*

A purchase is forever... [not]

A purchase is forever... [not]

Premium upgrade

A purchase is forever... [not]

Premium upgrade

Ad-free

A purchase is forever... [not]

Premium upgrade

Ad-free

Special items

A purchase is forever... [not]

Premium upgrade

Ad-free

Special items

Levels

So how do you implement:

So how do you implement:

Health potions

So how do you implement:

Health potions

50 coins

So how do you implement:

Health potions

50 coins

Season pass

Consumption

Consumable items?

Consumable items?

BRUNO'S STUFF

Consumable items?

1. Bruno buys COOL ITEM.

BRUNO'S STUFF

Consumable items?

1. Bruno buys COOL ITEM.

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?

{ COOL ITEM }

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?

   { COOL ITEM }

3. Bruno **consumes** COOL ITEM

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?
   { COOL ITEM }

3. Bruno **consumes** COOL ITEM

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?
   { COOL ITEM }

3. Bruno **consumes** COOL ITEM

4. What items does Bruno own?

Consumable items?

1. Bruno buys COOL ITEM.

2. What items does Bruno own?

   { COOL ITEM }

3. Bruno **consumes** COOL ITEM

4. What items does Bruno own?

   { }

BRUNO'S STUFF

Consumable items?

```
Bundle b = mService.consumePurchase(
    3,                          // API version
    “com.example.xyz”, // package name
    token                       // purchase token
)
```

When should I consume?

When should I consume?

Method 1

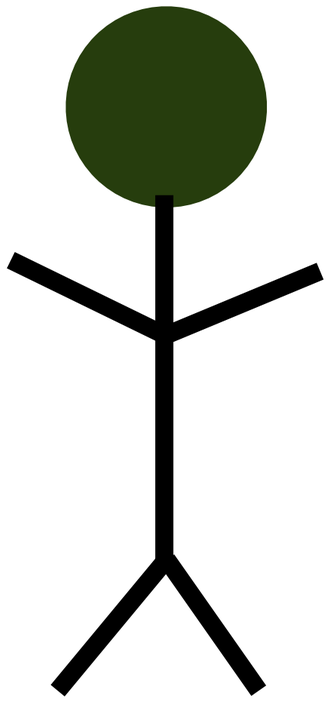
on item’s actual usage

Method 2

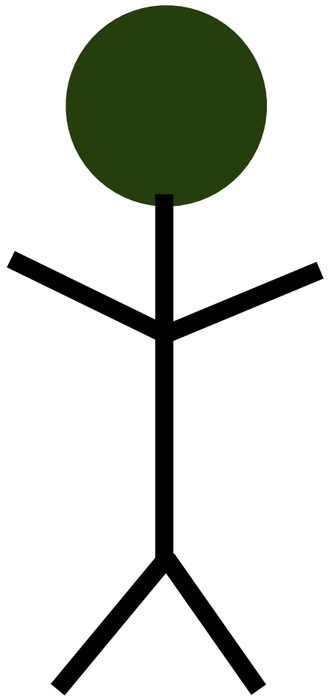
immediately upon purchase

Method 1: On actual usage

Method 1: On actual usage

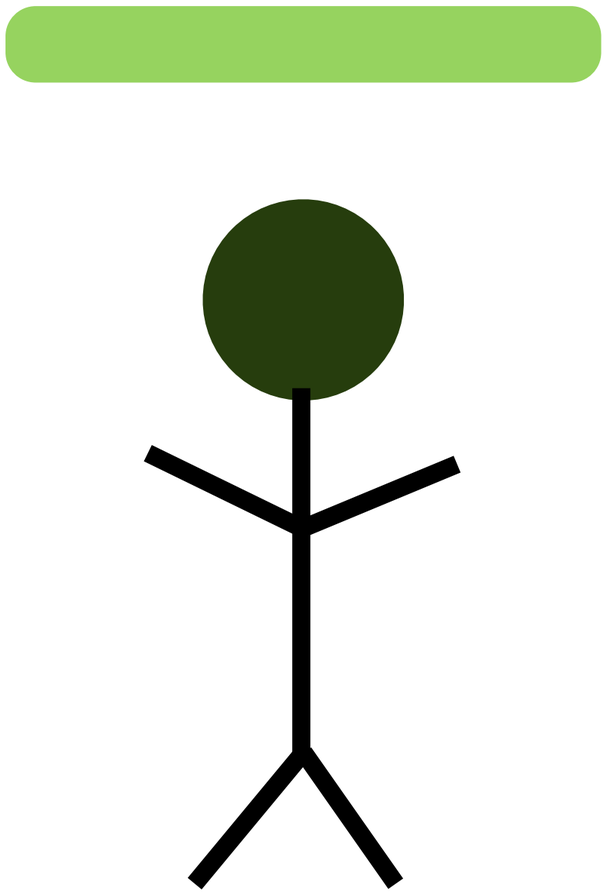


Method 1: On actual usage



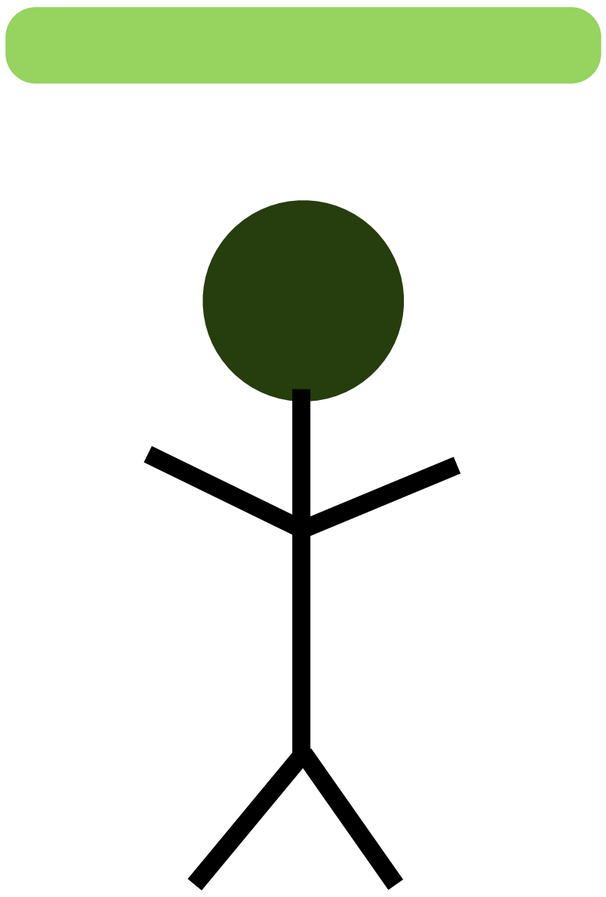
BRUNO'S STUFF

Method 1: On actual usage

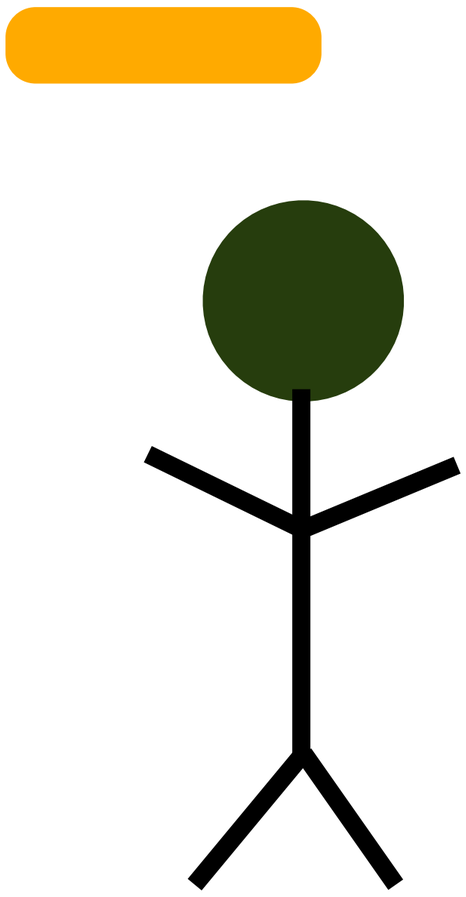


BRUNO'S STUFF

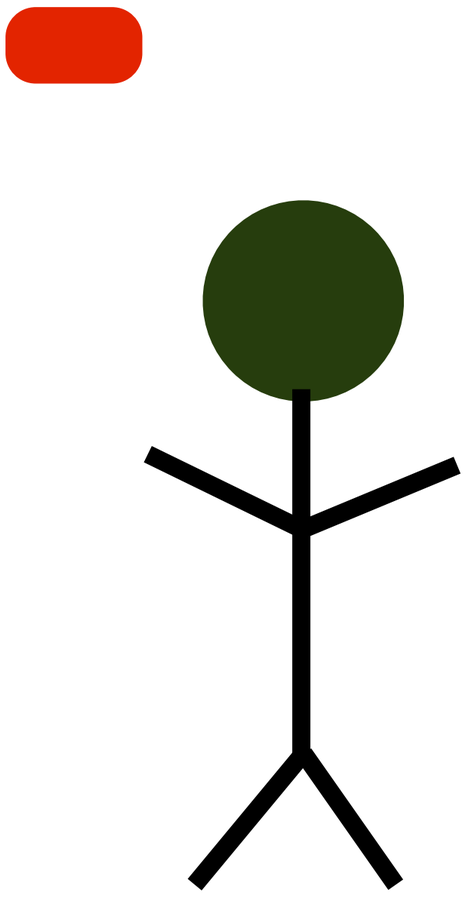
Method 1: On actual usage



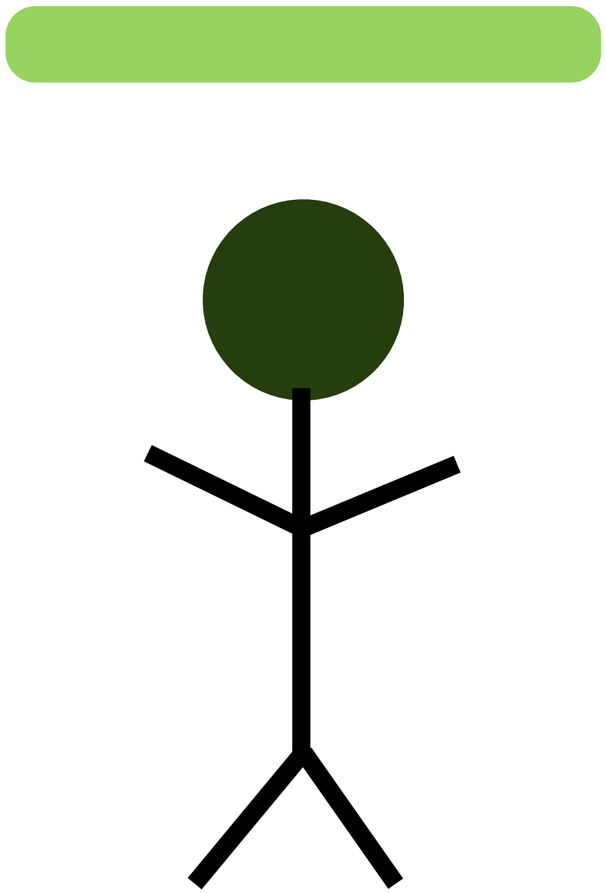
Method 1: On actual usage



Method 1: On actual usage



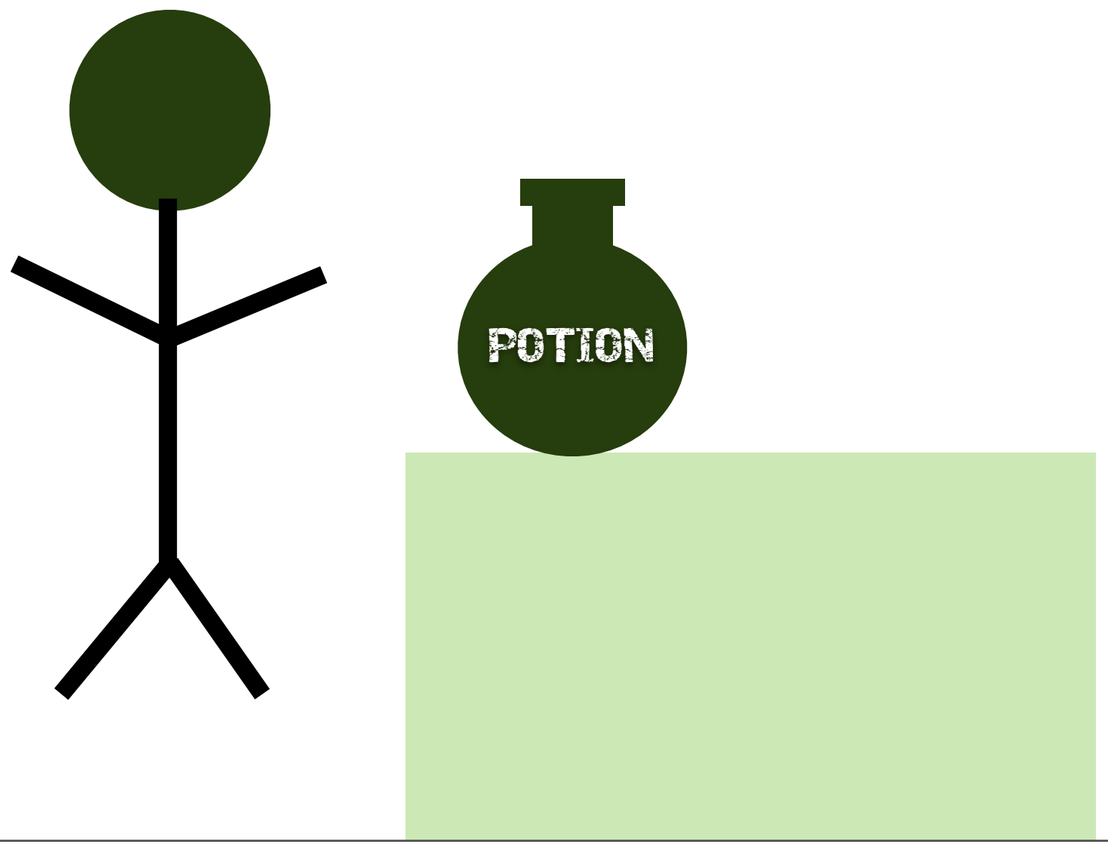
Method 1: On actual usage



BRUNO'S STUFF

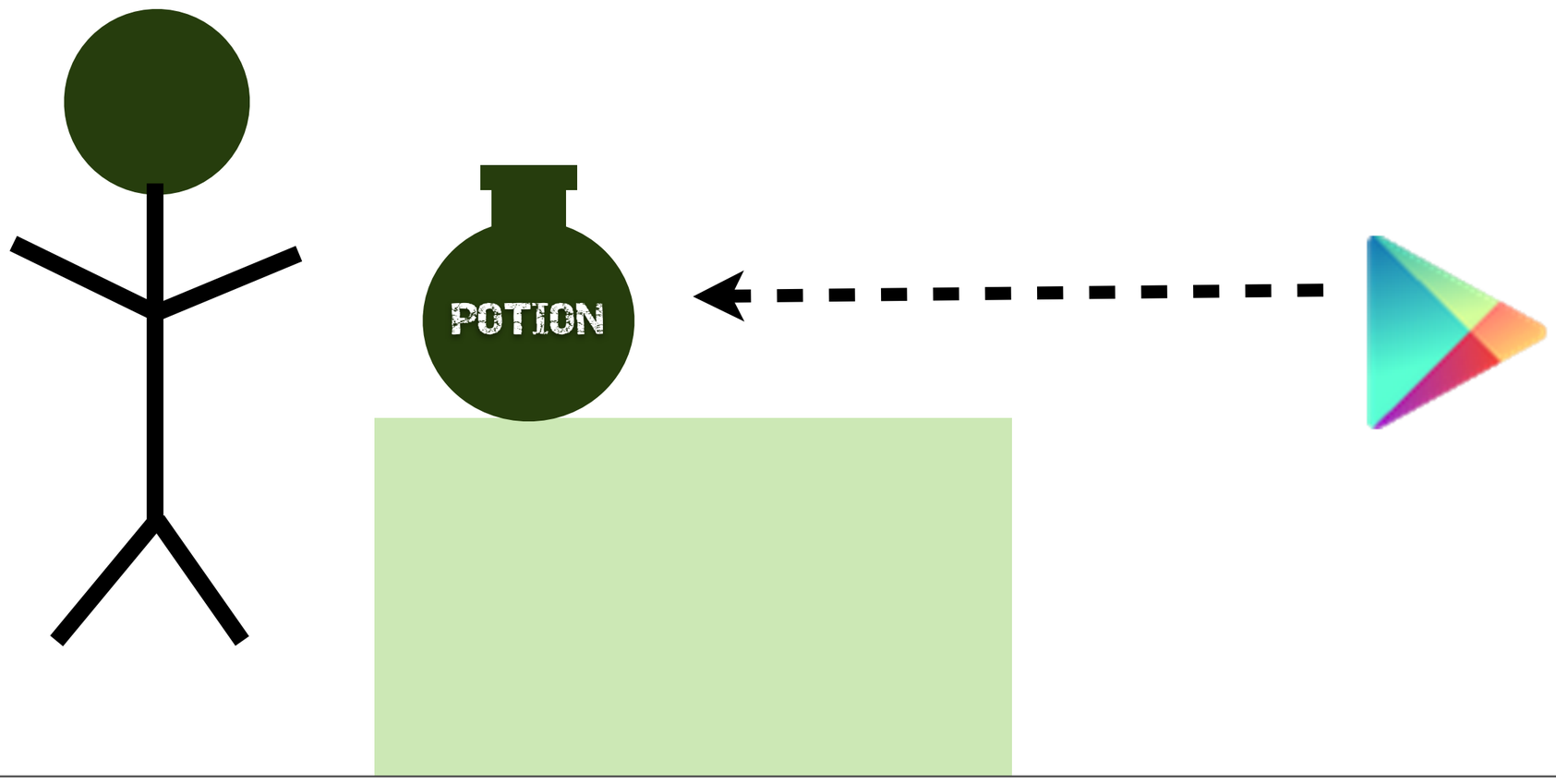
Limitations?

I want a potion!



Limitations?

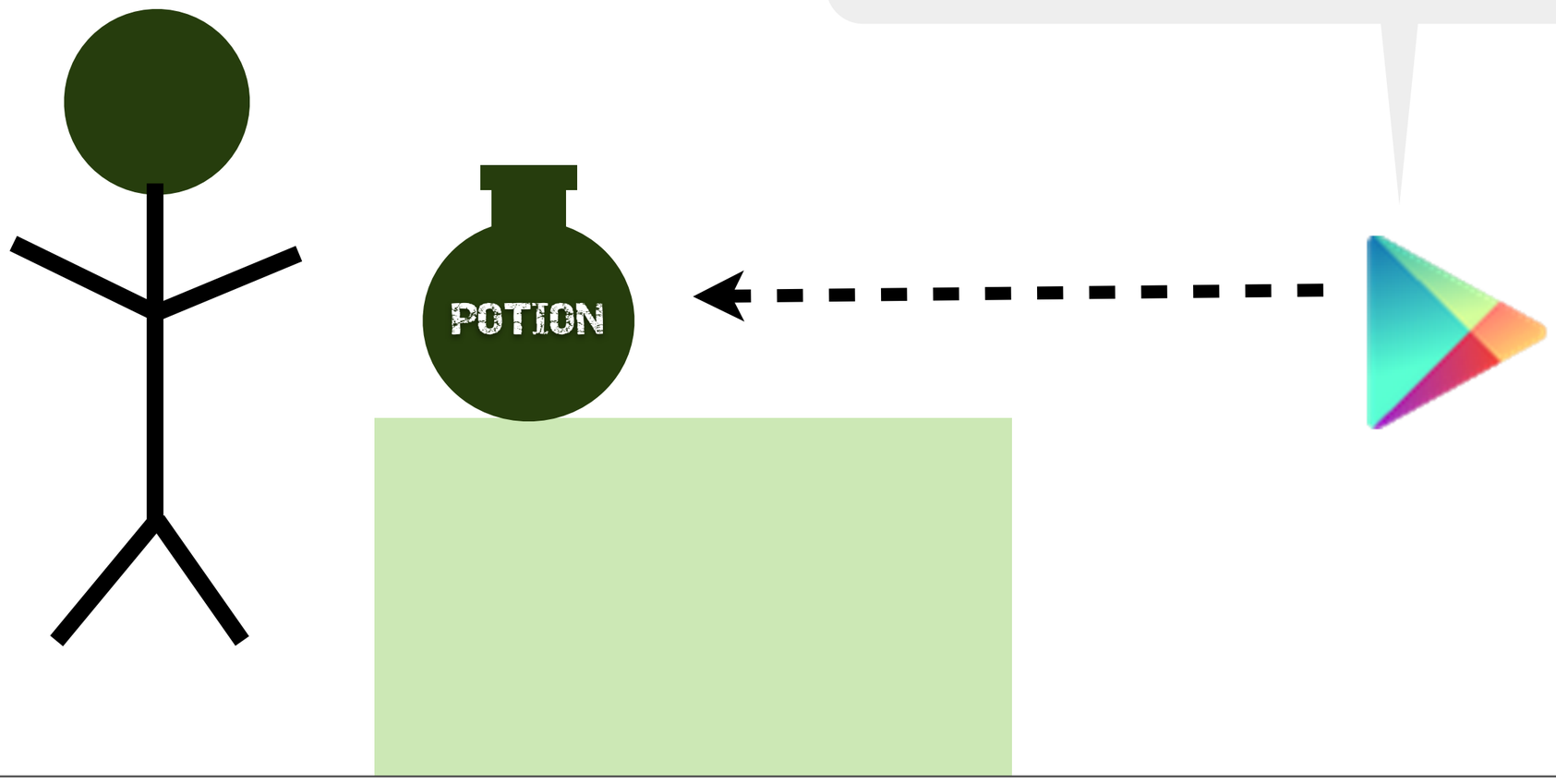
I want a potion!



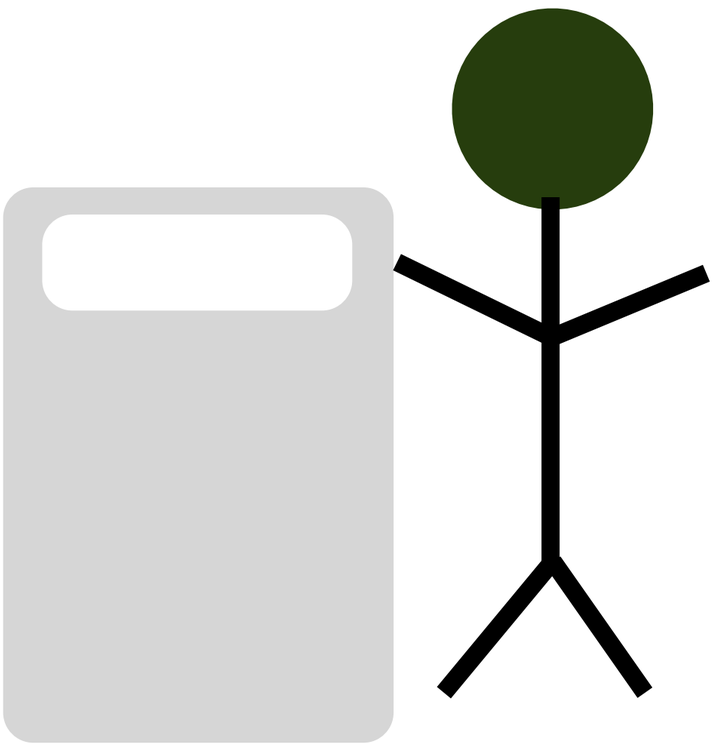
Limitations?

I want a potion!

I find your request illogical, user.

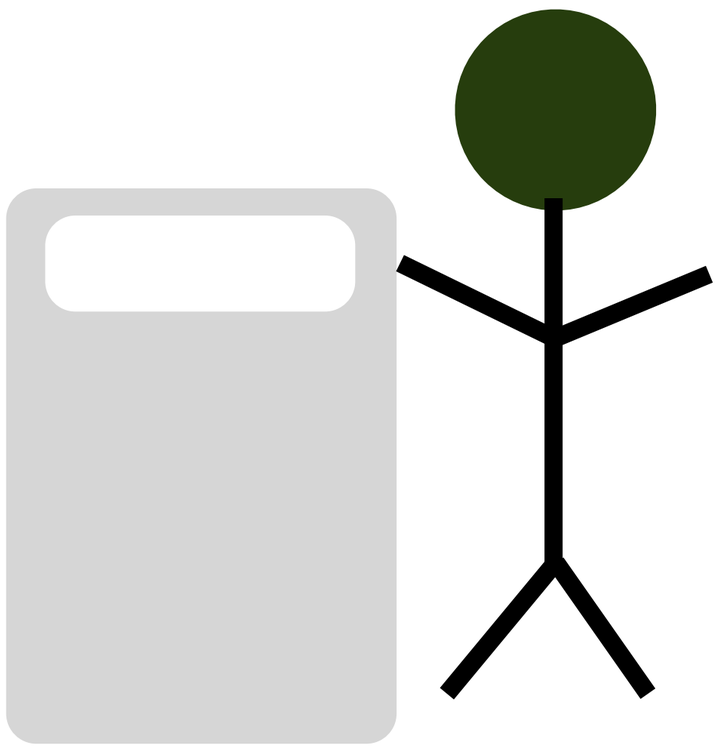


Method 2: **Consume upon purchase**

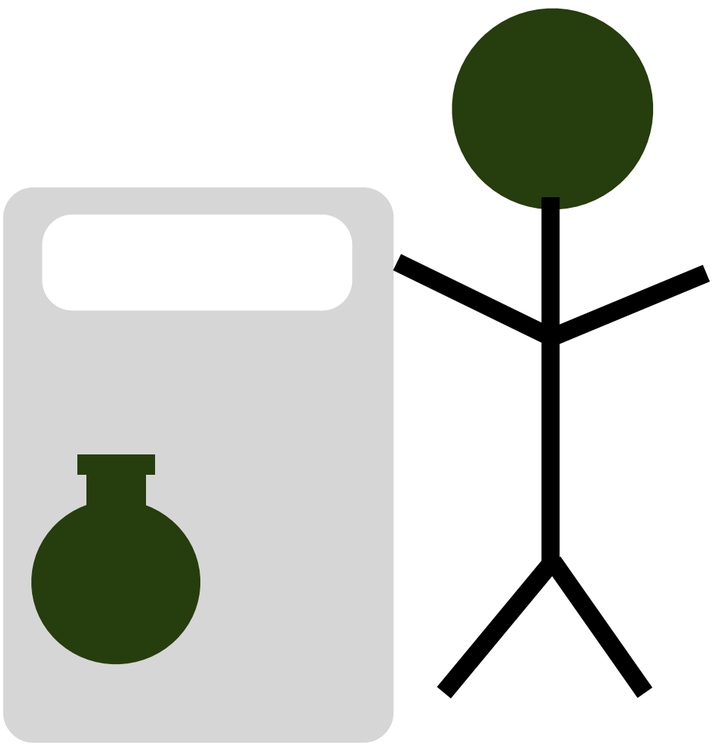


BRUNO'S STUFF

Method 2: **Consume upon purchase**

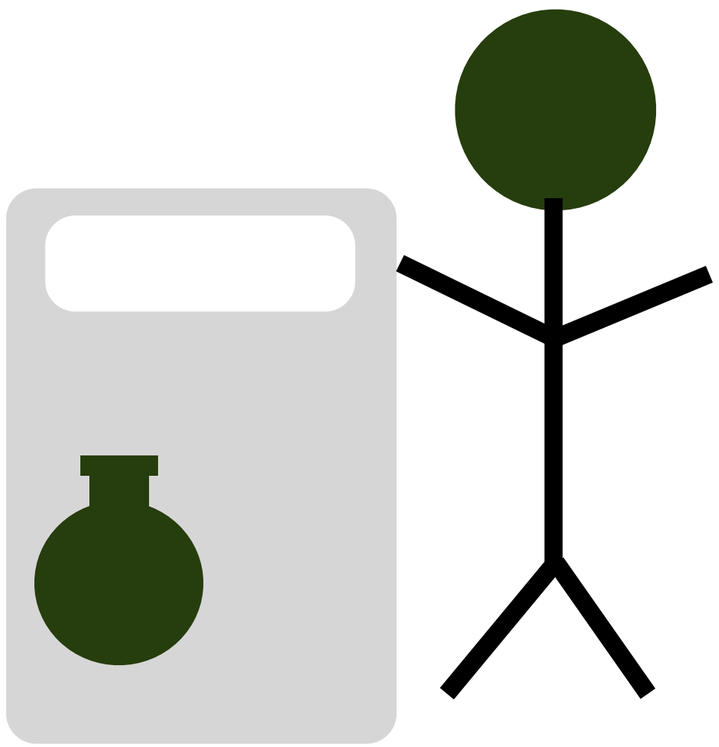


Method 2: **Consume upon purchase**

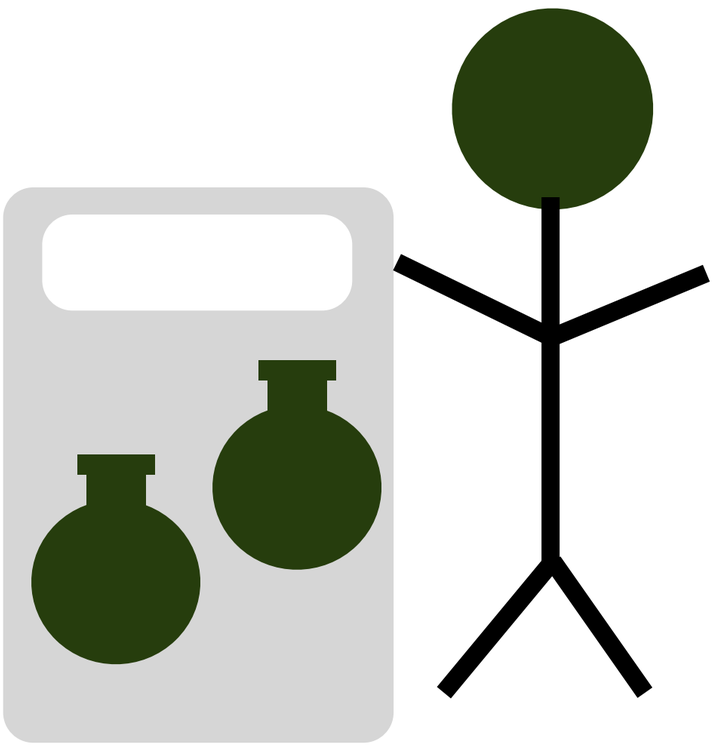


BRUNO'S STUFF

Method 2: **Consume upon purchase**



Method 2: **Consume upon purchase**



BRUNO'S STUFF

Also:

Method 2

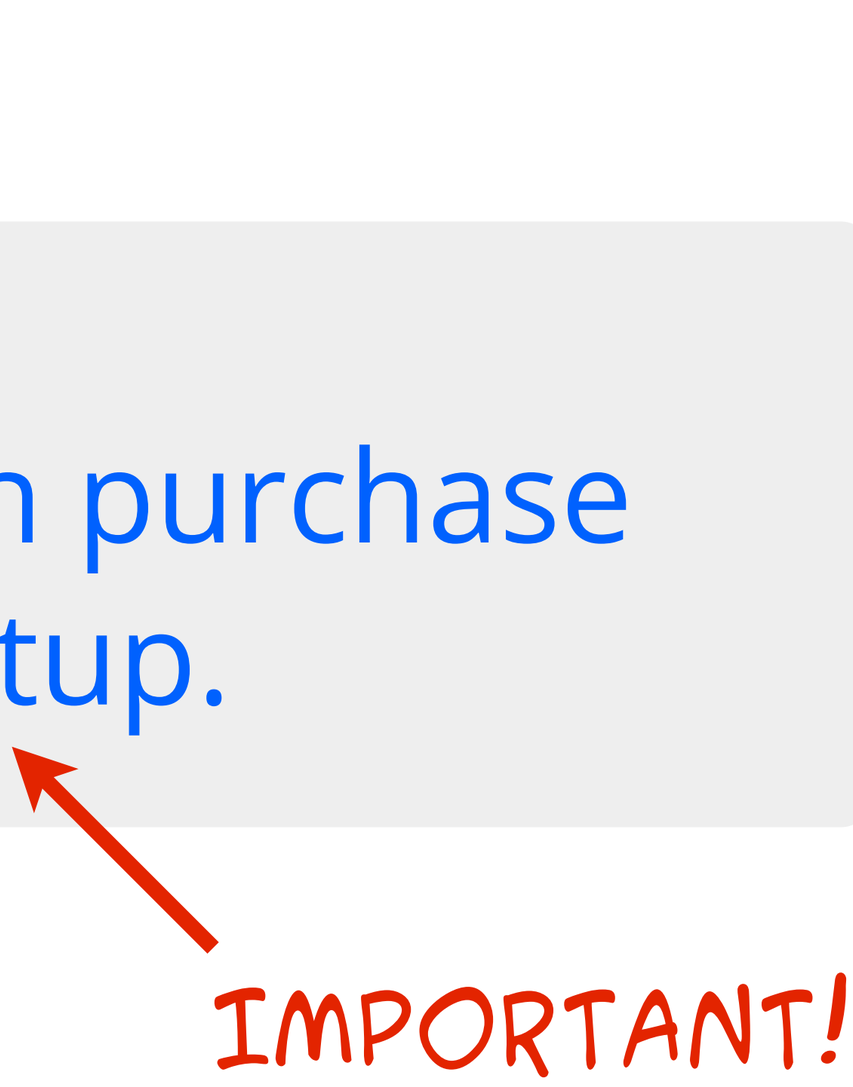
immediately upon purchase

Also:

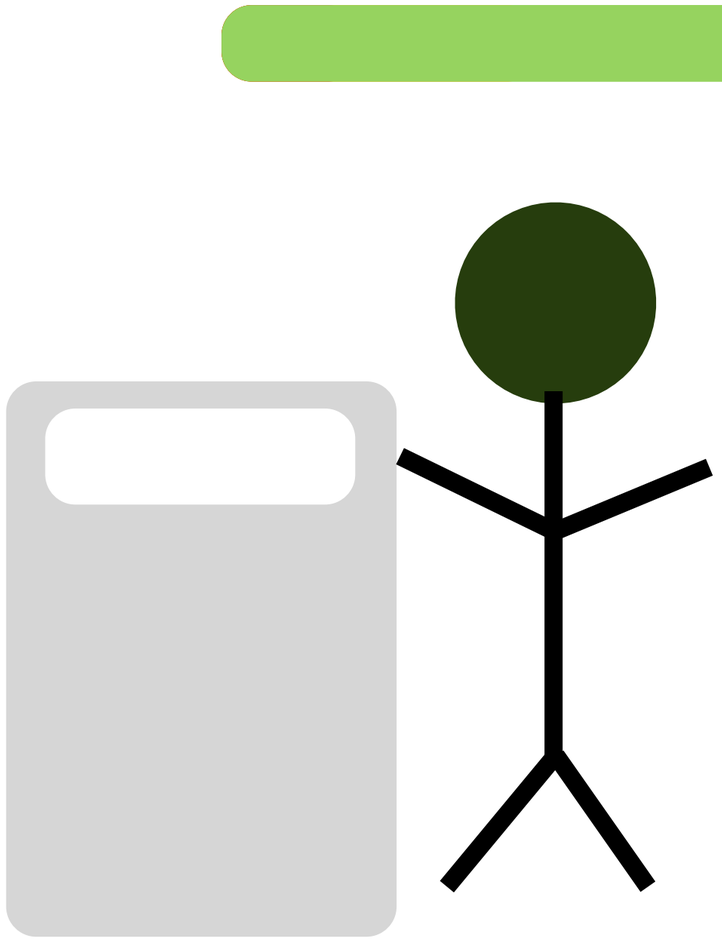
Method 2

immediately upon purchase

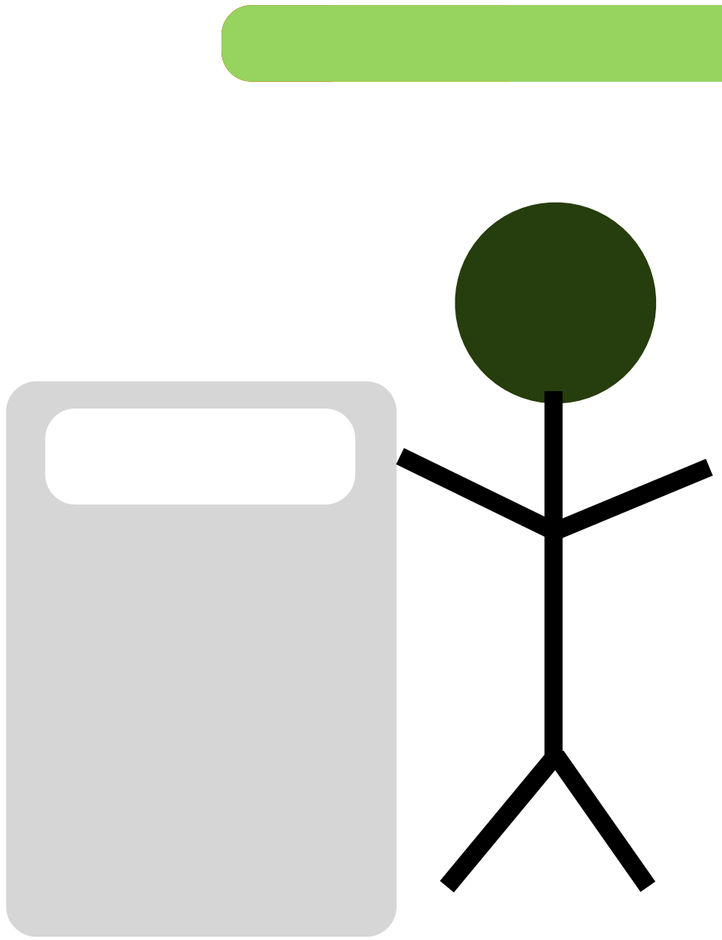
...and also on startup.



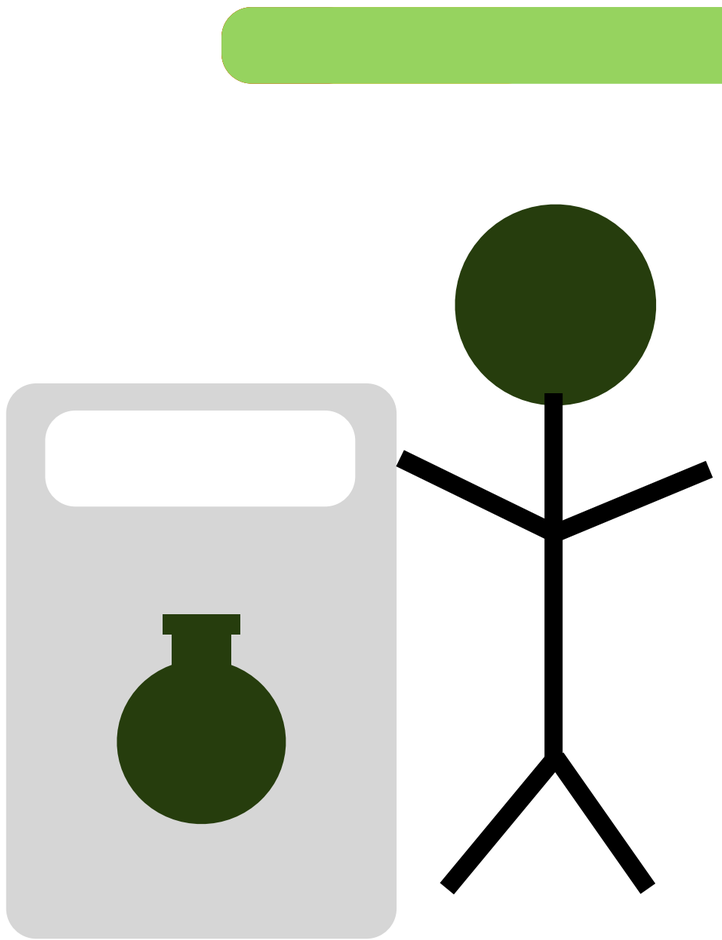
Method 2: Consume upon purchase and startup



Method 2: Consume upon purchase and startup



Method 2: Consume upon purchase and startup



BRUNO'S STUFF

Summarizing

on startup:
getPurchases()
if has potion, consume()

when user wants to purchase:
getBuyIntent(), launch the Intent.

onActivityResult:
if purchase successful,
  consume()

consume():
If consume successful,
  add potion to inventory.

Get product details

Query SKU details

```
Bundle getSkuDetails(
    int apiVersion,
    String packageName,
    String type,
    Bundle skusBundle
);
```

```
{
  “productId”: “xyz123”
  “type”: “inapp” | “subs”,
  “price”: “$1.99”
  “title”: “100 coins”
  “description”: “...”
}
```

```
{
  “productId”: “xyz123”
  “type”: “inapp” | “subs”,
  “price”: “$1.99”
  “title”: “100 coins”
  “description”: “...”
}
```

Note: Price is for **display** not **parsing**!

Subscriptions

Are subscriptions supported?

```
public void onServiceConnected(
                ComponentName name, IBinder service) {

    mService = IInAppBillingService.Stub.asInterface(service);
    int response = mService.isBillingSupported(3,
                getPackageName(), “subs”);
    if (response == BILLING_RESPONSE_RESULT_OK) {
        // has V3 subscriptions!
    }
    else {
        // no V3 subscriptions...
    }
}
```

Launch the purchase flow (subscription)

Launch the purchase flow (subscription)

```
Bundle bundle = mService.getBuyIntent(3, "com.example.myapp",
    MY_SKU, “subs”, developerPayload);

PendingIntent pendingIntent =
    bundle.getParcelable(RESPONSE_BUY_INTENT);

if (bundle.getInt(RESPONSE_CODE) ==
                    BILLING_RESPONSE_RESULT_OK) {
    startIntentSenderForResult(pendingIntent, RC_BUY,
        new Intent(), Integer.valueOf(0), Integer.valueOf(0),
        Integer.valueOf(0));
}
```

RESULT COMES BACK ON
onActivityResult()

```
Bundle b = mService.consumePurchase(
    3,                          // API version
    “com.example.xyz”,  // package name
    token                       // purchase token
);
```

A subscription can’t be consumed.

Active Subcriptions?

Active Subcriptions?

```
Bundle bundle = mService.getPurchases(
        3, mContext.getPackageName(), ITEM_TYPE_SUBS);

if (bundle.getInt(RESPONSE_CODE) == BILLING_RESPONSE_RESULT_OK) {

    // same as before...

}
```

trial subscriptions

active subscriptions

expired subscriptions

trial subscriptions

active subscriptions

appear in getPurchases()

expired subscriptions

trial subscriptions

active subscriptions

appear in getPurchases()

expired subscriptions

don't appear in getPurchases()

trial subscriptions

active subscriptions

cancelled subscriptions

** until end of paid billing period*

expired subscriptions

appear in getPurchases()

don't appear in getPurchases()

Local cache?

Local cache?

Google Play app

Refresh: 24h (normally)

Careful with the UI thread!

Car ad!

This slide isn't responding.

Do you want to close it?

Wait | OK

Safe for UI thread

`isBillingSupported`

`getBuyIntent`

Safe for UI thread

`isBillingSupported`

`getBuyIntent`

Don't call on UI thread

`getPurchases`

`consumePurchase`

`getSkuDetails`

Server side subscription API

subscription

Is it valid?

When does it expire?

Will it auto-renew?

subscription

Cancel it.

```
{
    “orderId”: ...,
    “packageName”: ...,
    “productId”: ...,
    “purchaseTime”: ...,
    “purchaseState”: ...,
    “developerPayload”, ...,
    “purchaseToken”: ...
}
```

```
GET https://www.googleapis.com/
androidpublisher/v1/applications/
{packageName}/subscriptions/{subscriptionId}/
purchases/{token}
```

```
{
  "kind":
     "androidpublisher#subscriptionPurchase",

  "initiationTimestampMsec": ....,
  "validUntilTimestampMsec": ....,
  "autoRenewing": ....
}
```

More at:

https://developers.google.com/android-publisher/v1/

Security

Is this purchase legitimate?

Is this purchase legitimate?

arrrrrrr!
of courrrse!

Risk Model

audience

likelihood of piracy

item value

Risk Model

observed # of fake purchases

total legitimate purchases

technical difficulty

total purchases

Making life hard for pirates

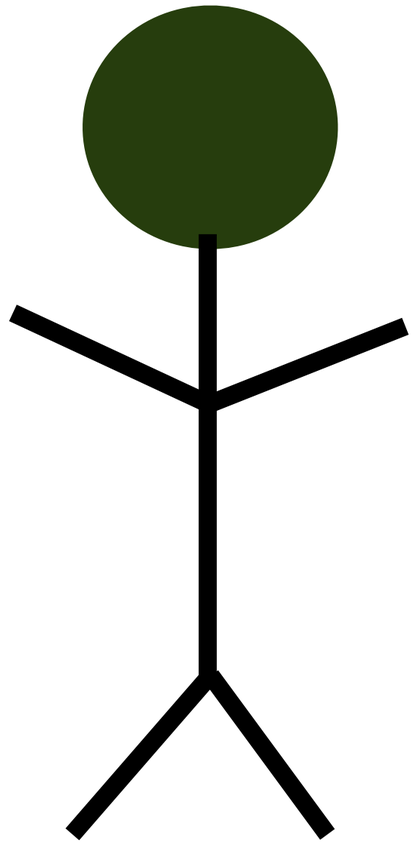
Your defenses:

1. developer payload
2. signature verification
3. server-side validation

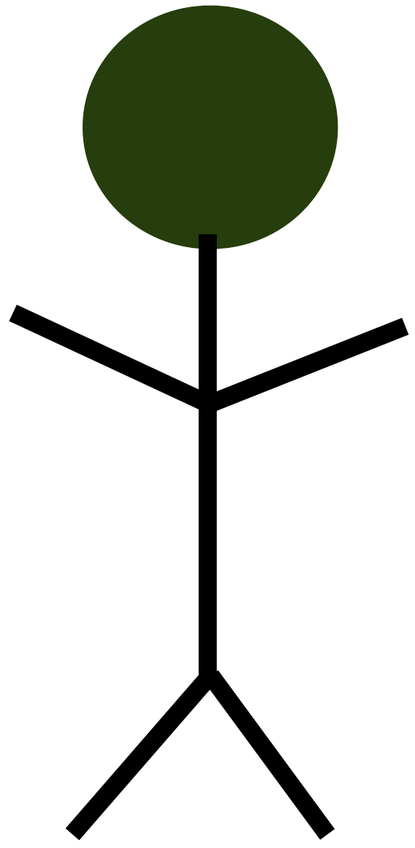
Developer Payload

Developer Payload

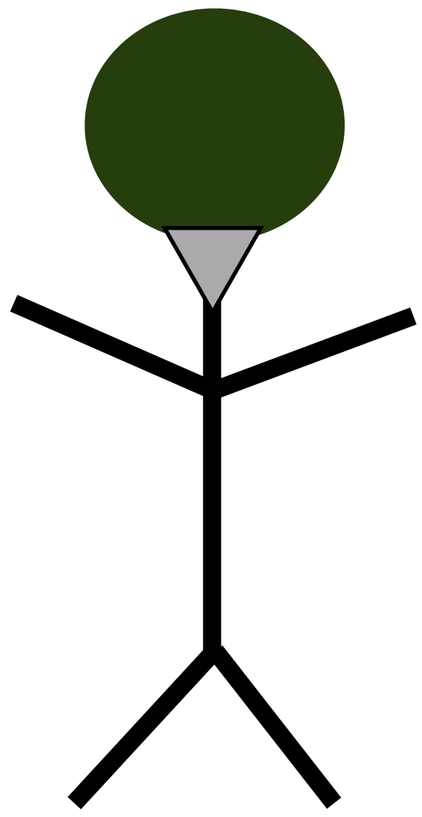
Bruno



Bruno



Onurb



Developer Payload

Developer Payload

Signature Verification

Your app has a public key

Google Play signs purchases

Always check signature!

Signature Verification

```
PublicKey publicKey = ...;
String signedData = ...;
String signature = ...;

Signature sig = Signature.getInstance(“SHA1withRSA”);
sig.initVerify(publicKey);
sig.update(signedData.getBytes());
if (!sig.verify(Base64.decode(signature))) {
    // failed!
}
return true;
```

Is client-side security enough?

Server-Side Validation

check signature

don't trust the client

check order number

valid? duplicate?

secure the handshake

Summary	MITM	Purchase replay	Framework compromise

Summary

	MITM	Purchase replay	Framework compromise
Wishing really hard	X	X	X

Summary

	MITM	Purchase replay	Framework compromise
Wishing really hard	X	X	X
Client-side signature verification		X	X

Summary

	MITM	Purchase replay	Framework compromise
Wishing really hard	X	X	X
Client-side signature verification		X	X
+ unique developer payload			X

Summary

	MITM	Purchase replay	Framework compromise
Wishing really hard	X	X	X
Client-side signature verification	✓	X	X
+ unique developer payload	✓	✓	X
+ Server-side verification	✓	✓	✓

KEEP

CALM

AND

USE BEST

PRACTICES

KEEP
CALM
AND
USE BEST
PRACTICES

developer payload

signature verification

server-side validation

Sandbox

Before

Now (Sandbox)

Before

mock product
mock purchase flow
mock result

Now (Sandbox)

Before

mock product
mock purchase flow
mock result

real product
real credit card
real purchase flow
real charge

Now (Sandbox)

Before

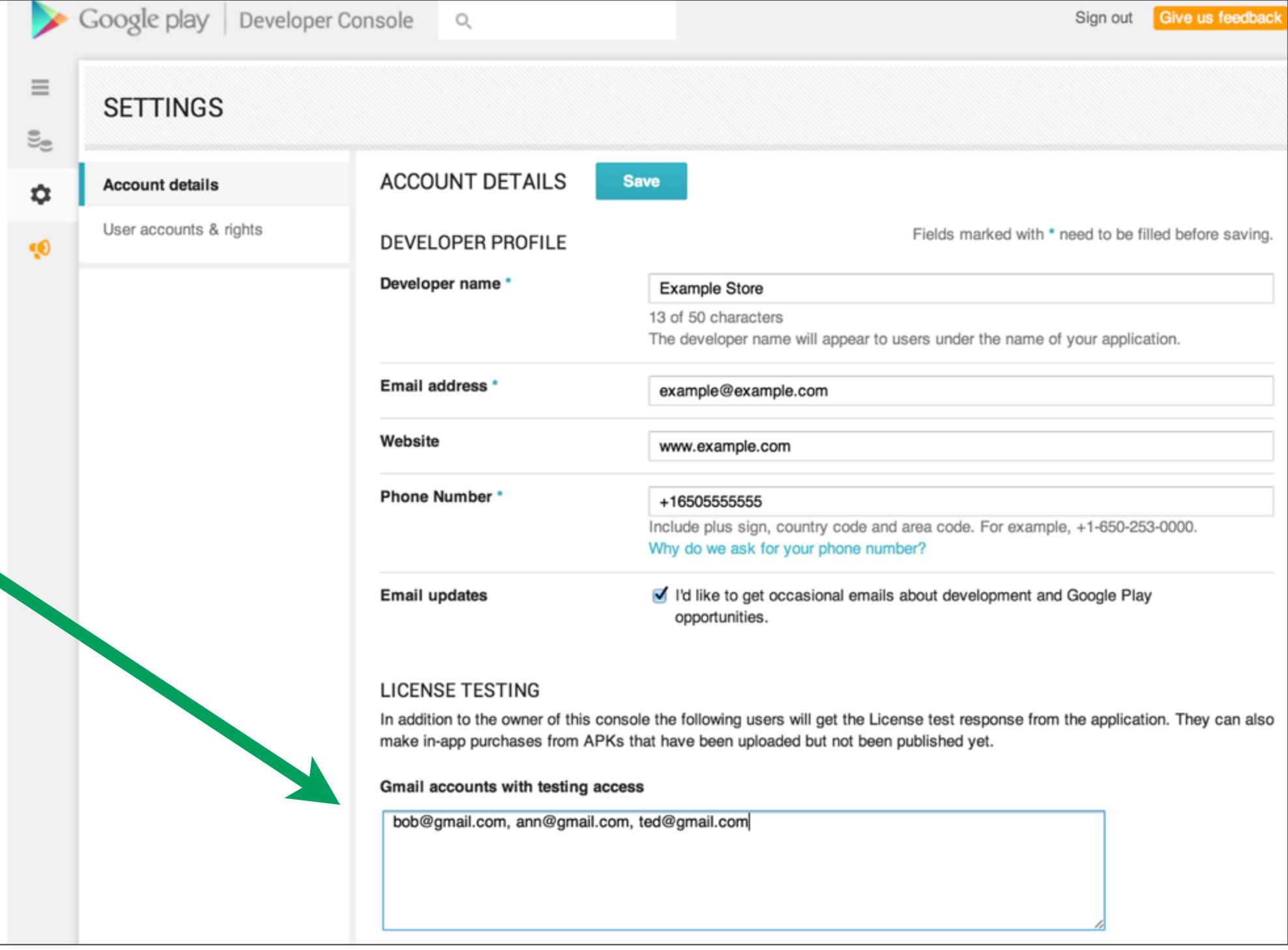
mock product
mock purchase flow
mock result

real product
real credit card
real purchase flow
real charge

Now (Sandbox)

real product
real credit card
real purchase flow
real result
no charge

purchase
test accounts



SINGLE PLAYER

1,000 coins (Nostalgic Racer) $0.99

MasterCard- 0541

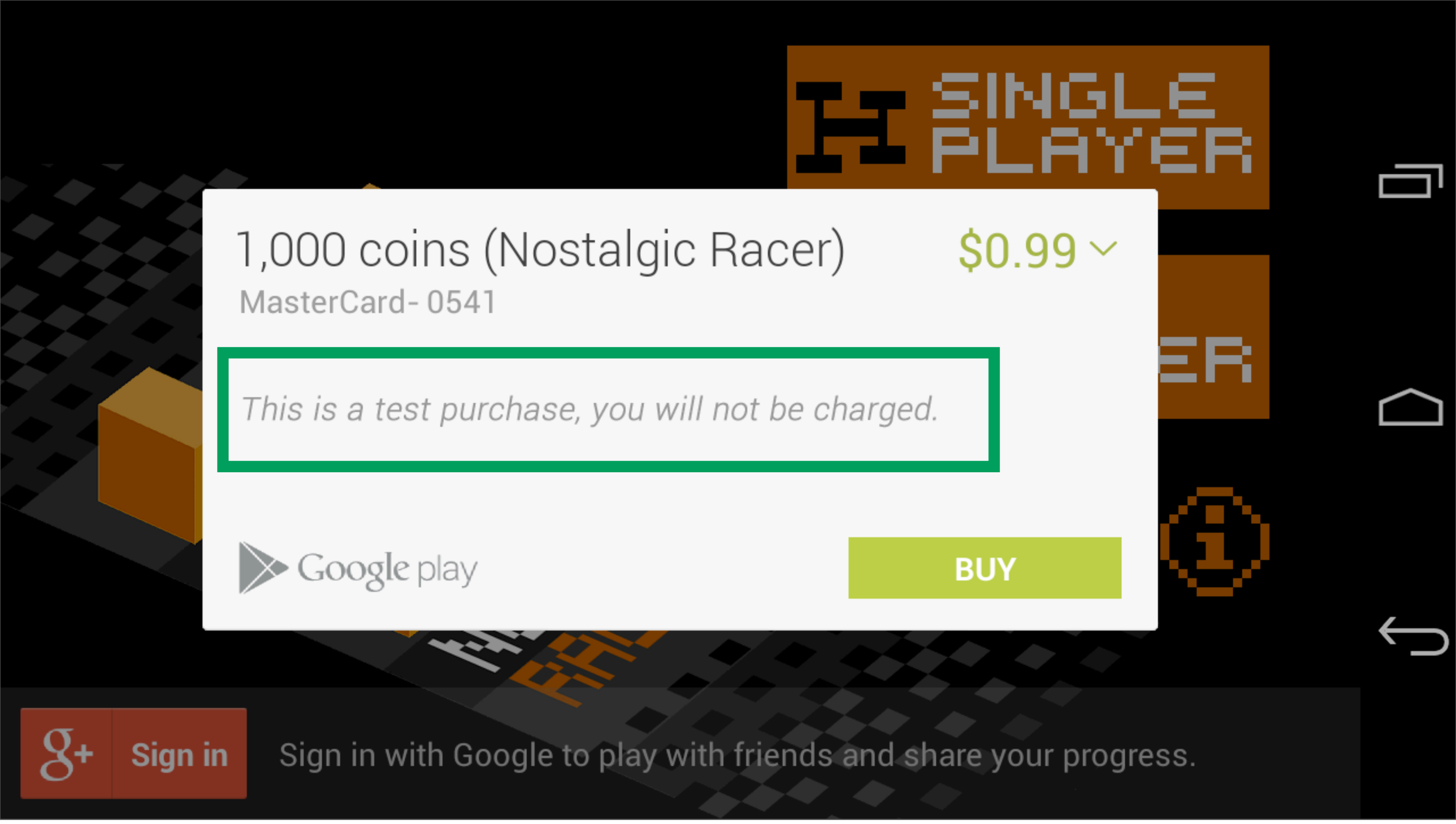
This is a test purchase, you will not be charged.

Google play

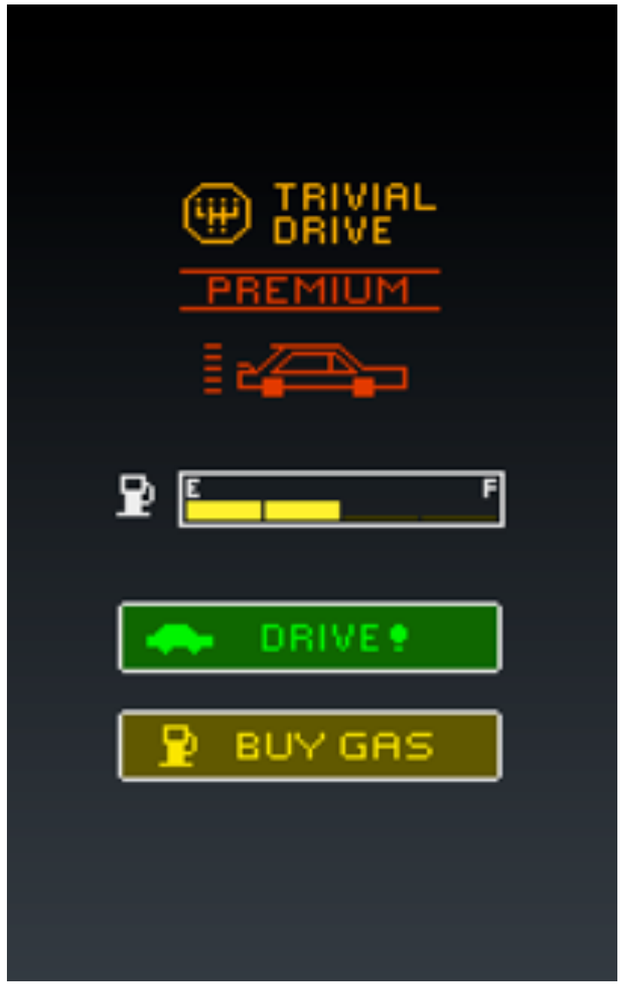
BUY

Sign in

Sign in with Google to play with friends and share your progress.


SINGLE PLAYER
1,000 coins (Nostalgic Racer)
$0.99
MasterCard- 0541
This is a test purchase, you will not be charged.
Google play
BUY
Sign in
Sign in with Google to play with friends and share your progress.

Sample: TrivalDrive



SDK manager

code.google.com/p/marketbilling

IAB v2

IAB v2

IAB v3

IAB v2

IAB v3

subscriptions

IAB v2

IAB v3

subscriptions

consumption

IAB v2

IAB v3

subscriptions

consumption

product details

IAB v2

IAB v3

server API

subscriptions

consumption

product details

IAB v2

IAB v3

server API

subscriptions

consumption

security

product details

IAB v2

IAB v3

server API

subscriptions

consumption

security

product details

sandbox

developer experience

developer experience

user experience

Monday, May 20,

<Thank You!>

plus.google.com/+BrunoOliveira

Bruno Oliveira


Google
Developers