Efficient Android Threading

Anders Göransson

anders.goransson@jayway.com

www.jayway.com
www.jayway.com/blog
www.oredev.org

ØREDEV

JAYWAY

Agenda

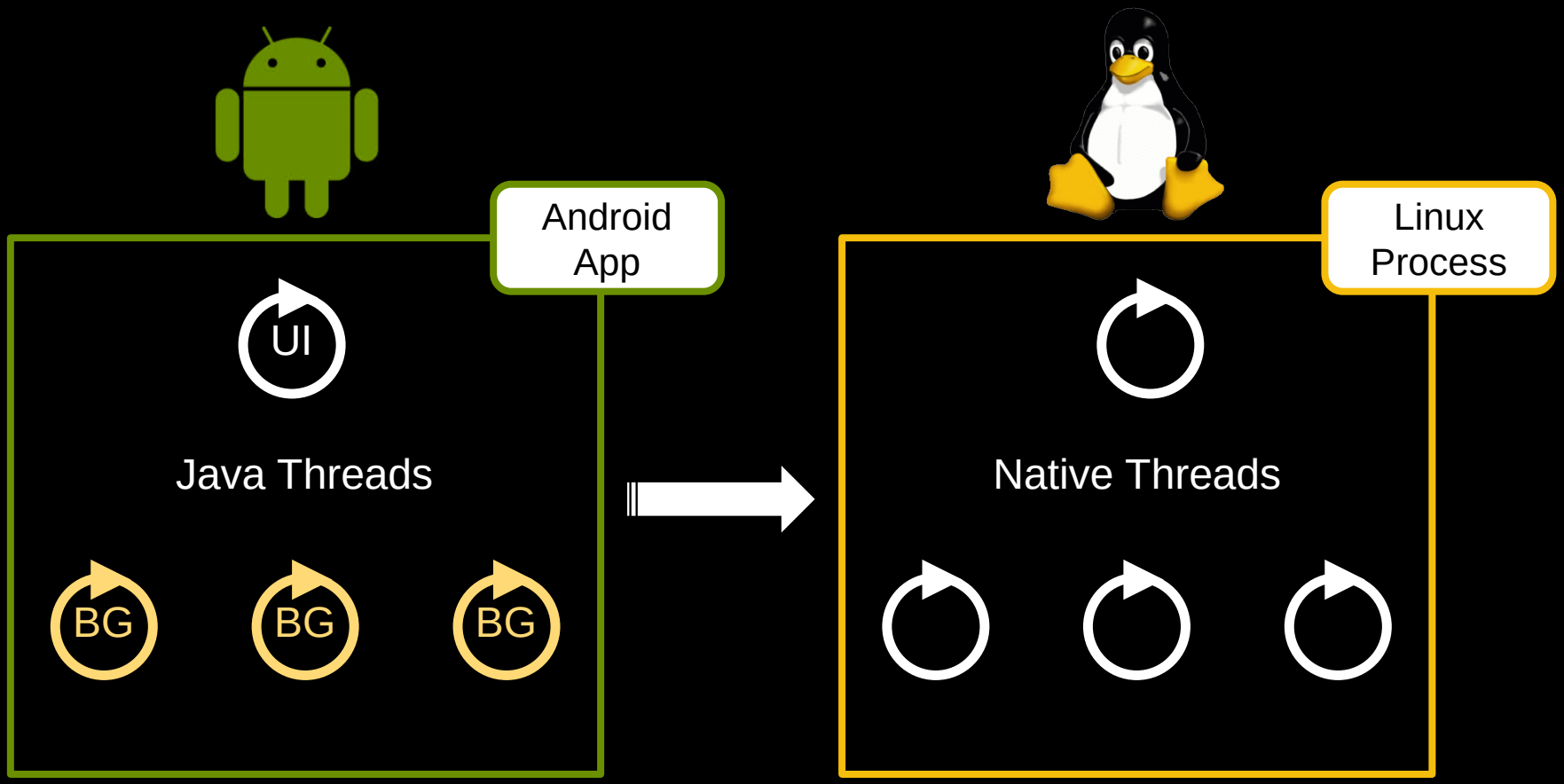
Optimize UI thread execution

Threads on Android

Asynchronous Techniques

JAYWAY

Threads on Android



JAYWAY

Android Scheduling

Process level:

1. Foreground
2. Visible
3. Service
4. Background

Android Scheduling

Process level:

1. Foreground
2. Visible
3. Service
4. Background

App A

App B

Foreground Thread Group

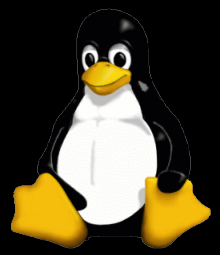
App A

> 90%

Background Thread Group

App B

< 10%



JAYWAY

Android Scheduling

Process level:

1. Foreground
2. Visible
3. Service
4. Background

App A

App B

Foreground Thread Group

App A

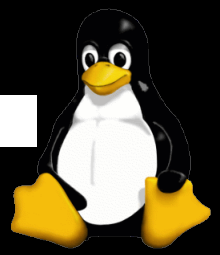
> 90%

```
Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
```

Background Thread Group

App B

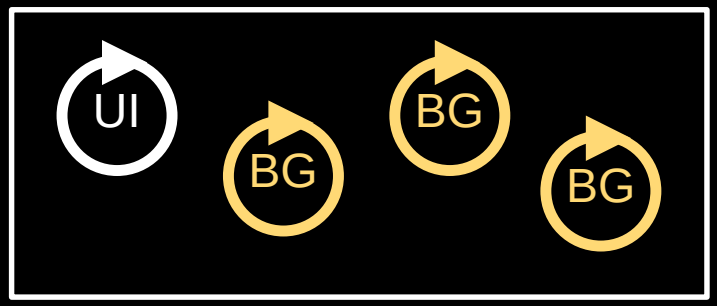
< 10%



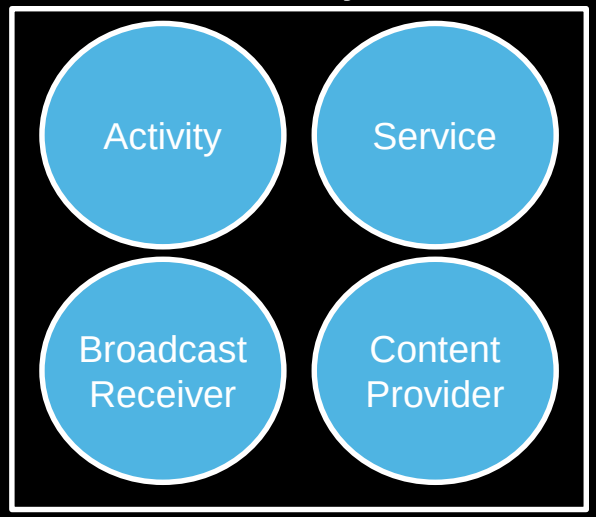
JAYWAY

Lifecycles

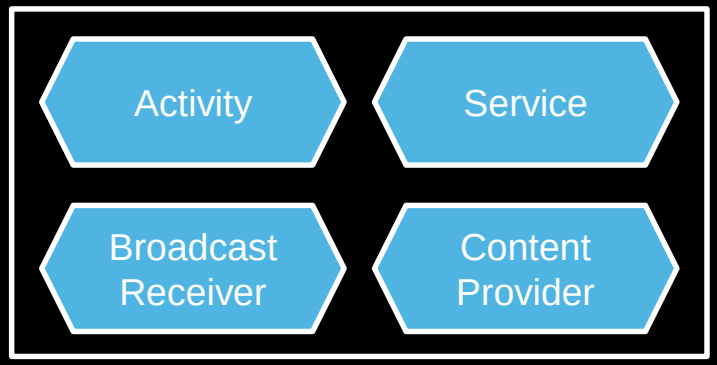
Threads



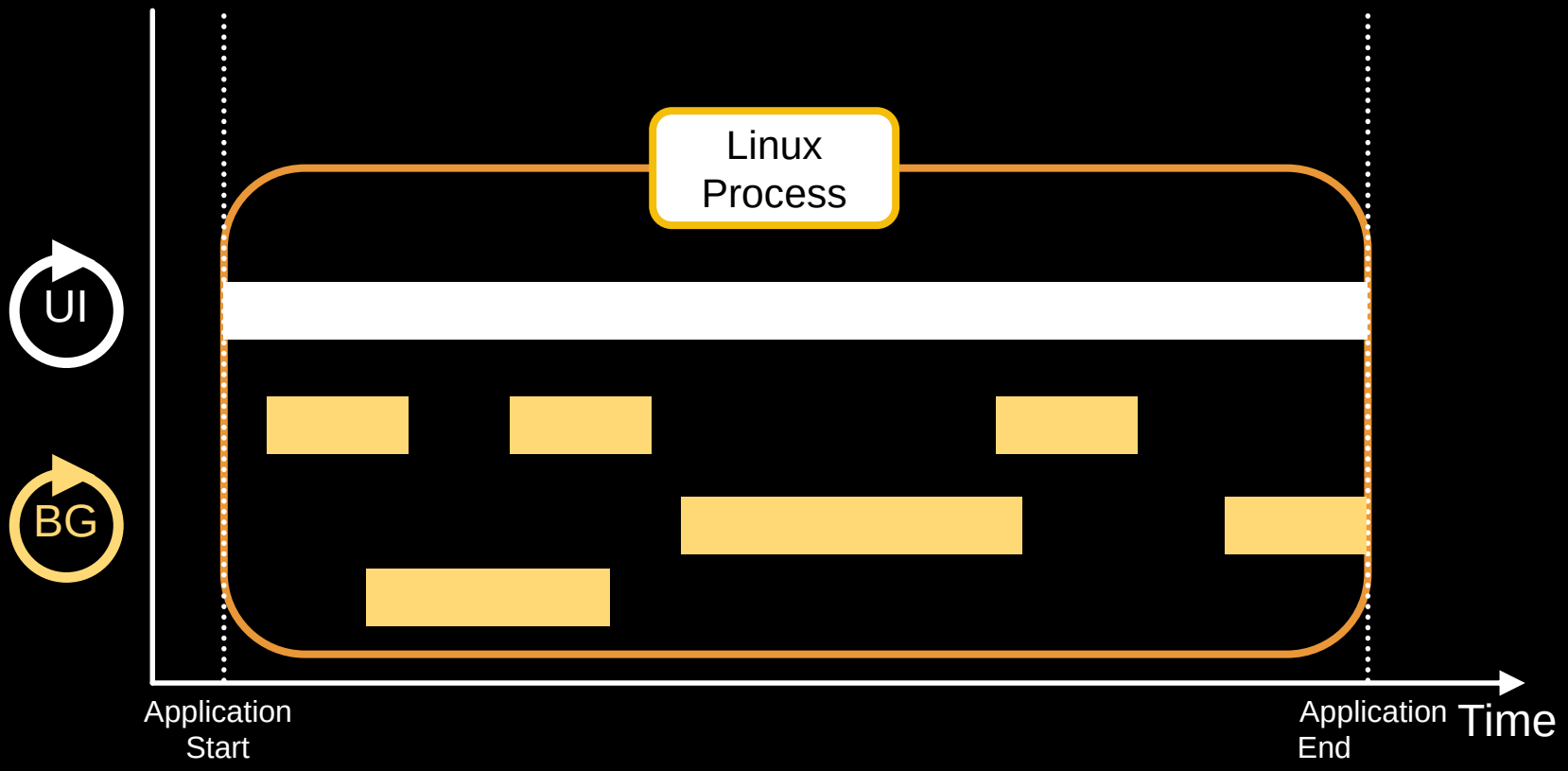
Java Objects



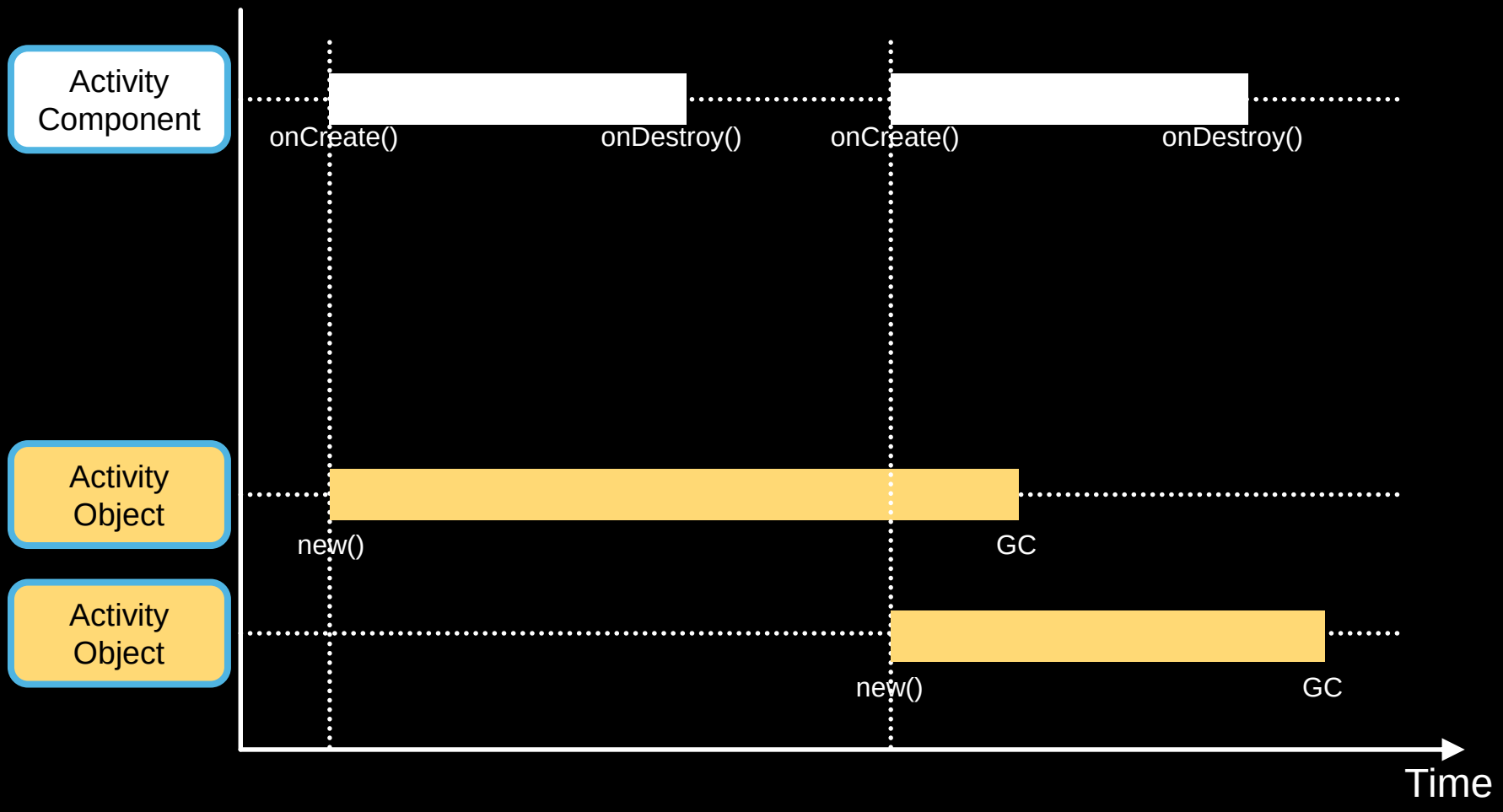
Android Components



Native Thread Lifecycle

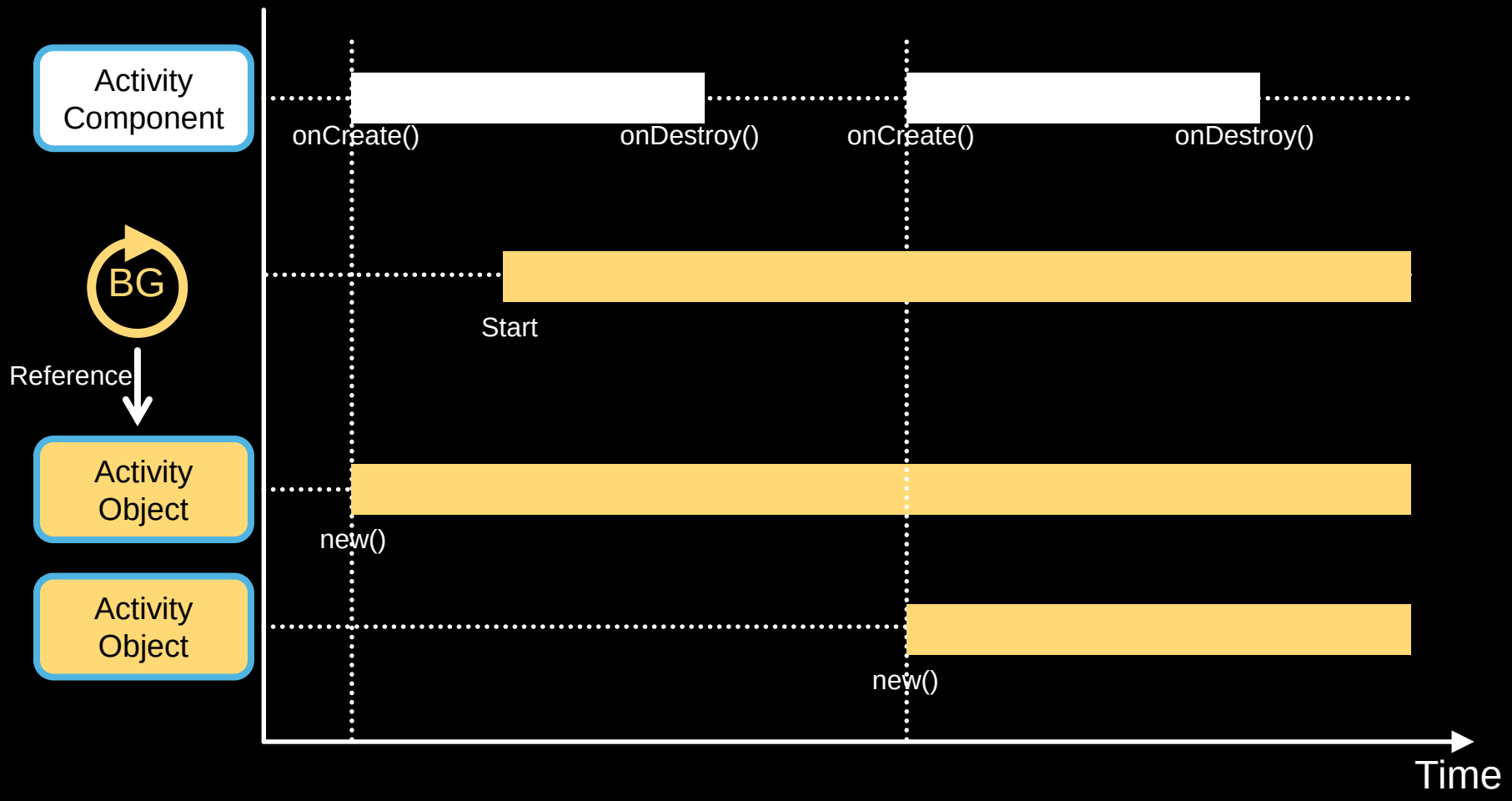


Example: Activity Lifecycle



JAYWAY

Example: Activity Lifecycle

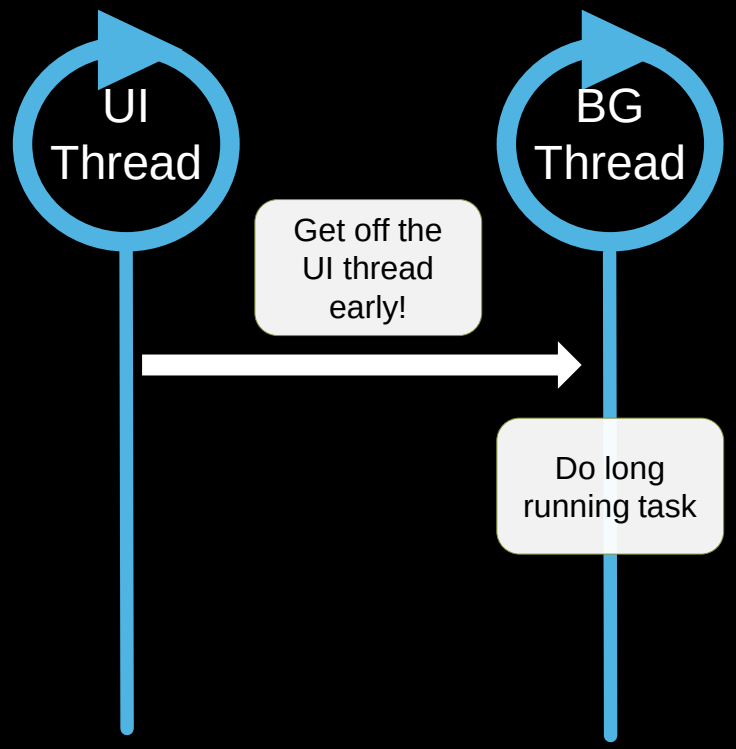


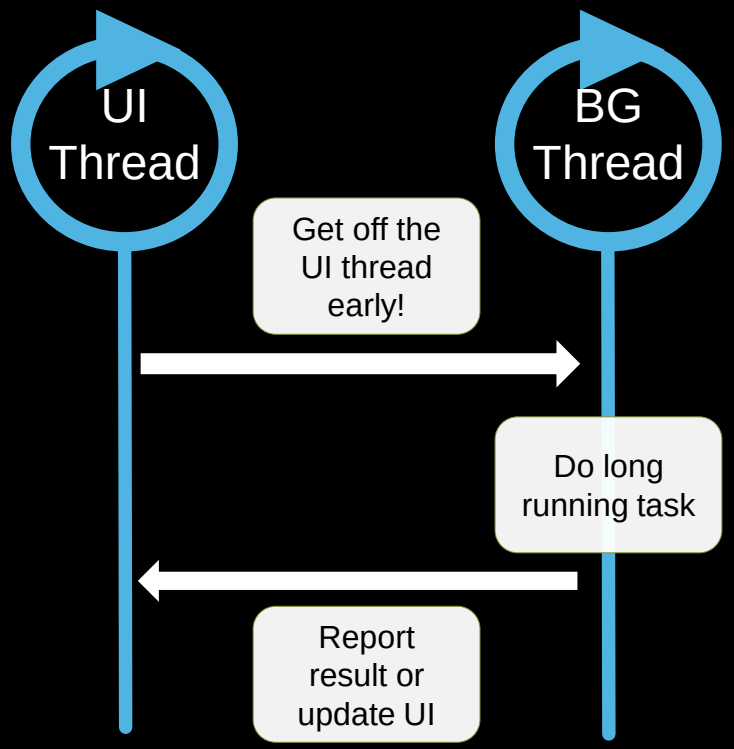
JAYWAY

Background Threads

- A thread is a GC root
- Implement cancellation policy for your background threads!

Use Cases





JAYWAY

Goal

What technique shall I use for my background thread execution?

JAYWAY

Asynchronous Techniques

- Thread
- Executor
- HandlerThread
- AsyncTask
- Service
- IntentService
- AsyncQueryHandler
- Loader

JAYWAY

Thread

- Plain old Java Thread

JAYWAY

Creation and Start

Anonymous Inner Class

```
new Thread(new Runnable() {
        public void run() {
                // Do long task
        }
}).start();
```

External Class

```
Thread t = new MyThread();
t.start();
```

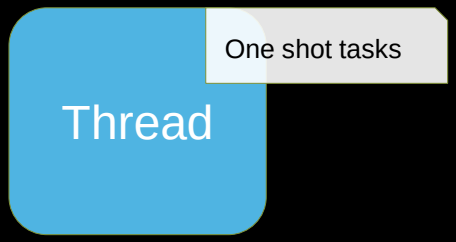
Implicit reference to outer class

Pitfalls

- Non-retained threads
- Missing cancellation policy
- Starting over and over again

Good Use Cases

- One-shot tasks
  - Post messages to the UI thread at end.

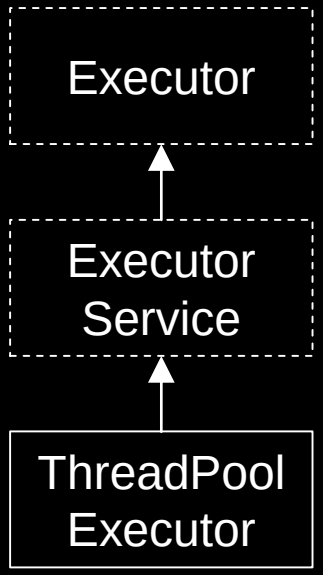


JAYWAY

Executor

- Powerful task execution framework.

```
public interface Executor {

        public void execute(Runnable r);

}
```



JAYWAY

ExecutorService

Task submission:

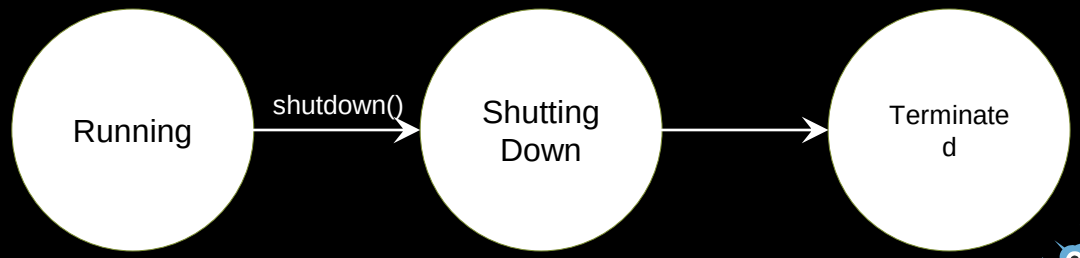
```
executorService.submit(MyTask);
executorService.invokeAll(Collection<Tasks>);
executorService.invokeAny(Collection<Tasks>);
```

Lifecycle management:

```
executorService.shutdown();
executorService.shutdownNow();
```

Lifecycle observation:

```
executorService.isShutdown();
executorService.isTerminated();
executorService.awaitTermination();
```



Task / Execution Environment

Task:
Independent unit of work executed anywhere

Runnable
run()

Callable
call()

Task manager/observer:

Future
isDone()
isCancelled()
cancel()

```
Future future = executorService.submit(Callable);
```

Execution Environment:
Technique used to execute the task

Executor
execute(Runnable)

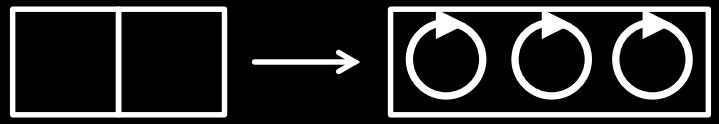
Thread Pools

- Executor managing a pool of threads and a work queue.
- Reduces overhead of thread creation

Thread Pools

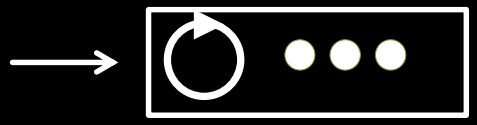
Fixed Thread Pool

```
Executors.newFixedThreadPool(3)
```



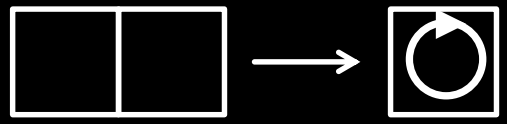
Cached Thread Pool

```
Executors.newCachedThreadPool()
```



Single Thread Pool

```
Executors.newSingleThreadExecutor()
```



Custom Thread Pool

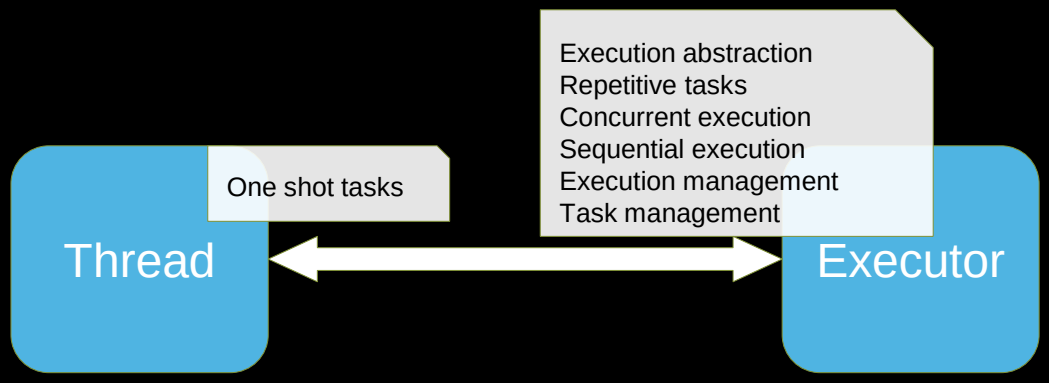
```
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, aliveTime, unit, workQueue)
```

Pitfalls

- Lost thread safety when switching from sequential to concurrent execution

Good Use Cases

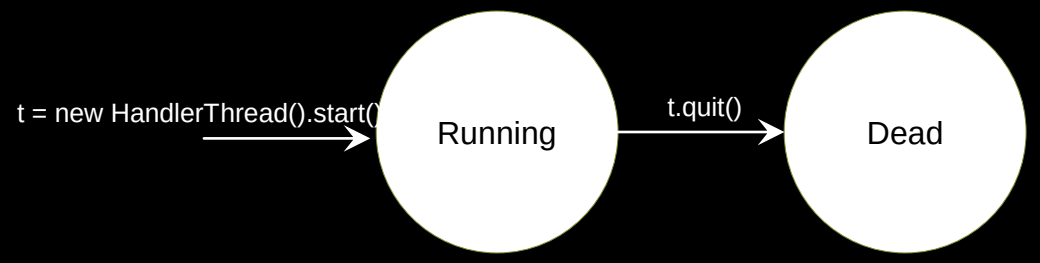
- Execute tasks concurrently
  - Multiple Http requests
  - Concurrent image processing
  - Use cases gaining performance from concurrent execution.
- Lifecycle management and observation of task execution.
- Maximum platform utilization


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor

JAYWAY

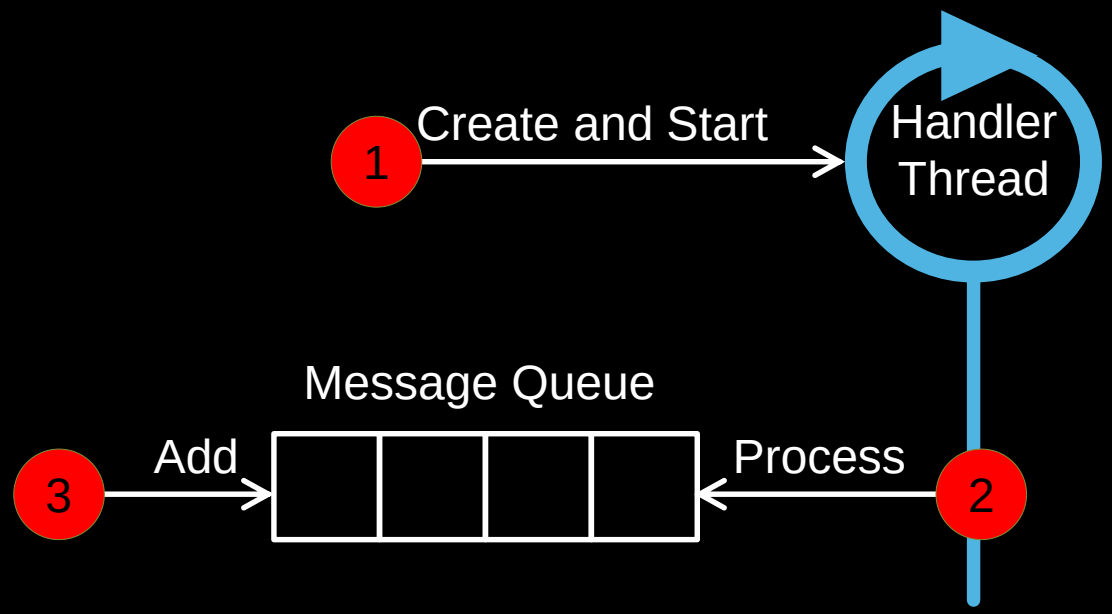
HandlerThread

- Inherits from Thread and encapsulates a Looper-object.
- Thread with a message queue and processing loop.
- Handles both Message and Runnable.



JAYWAY

How it works



1

```
t = new HandlerThread("BgThread");
t.start();
```

3

```
h.sendEmptyMessage(42);
```

2

```
Handler h = new Handler(t.getLooper()) {
        @Override
        public void handleMessage(Message msg) {
                //Process message
        }
};
```

JAYWAY

Handler

Runnable/Message submission:

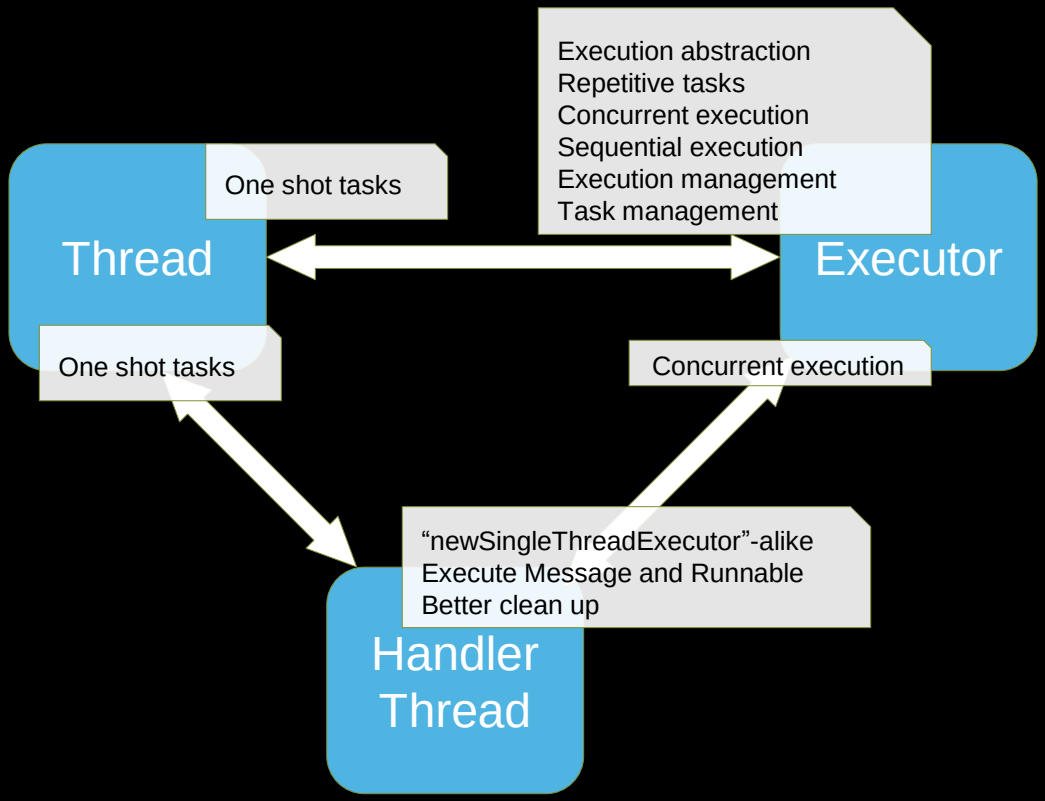
```
post(Runnable);
postDelayed(Runnable);
postAtTime(Runnable)
postAtFrontOfQueue(Runnable);
sendMessage(Message);
sendMessageDelayed(Message);
sendMessageAtTime(Message);
sendMessageAtFrontOfQueue(Message);
```

Runnable/Message removal:

```
removeCallbacks(Runnable);
removeMessages(Message)
```

Good Use Cases

- Keep a thread alive
- Sequential execution of messages
  - Avoid concurrent execution on multiple button clicks
  - State machine
  - Detailed control of message processing.


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
One shot tasks
Concurrent execution
“newSingleThreadExecutor”-alike
Execute Message and Runnable
Better clean up
Handler Thread

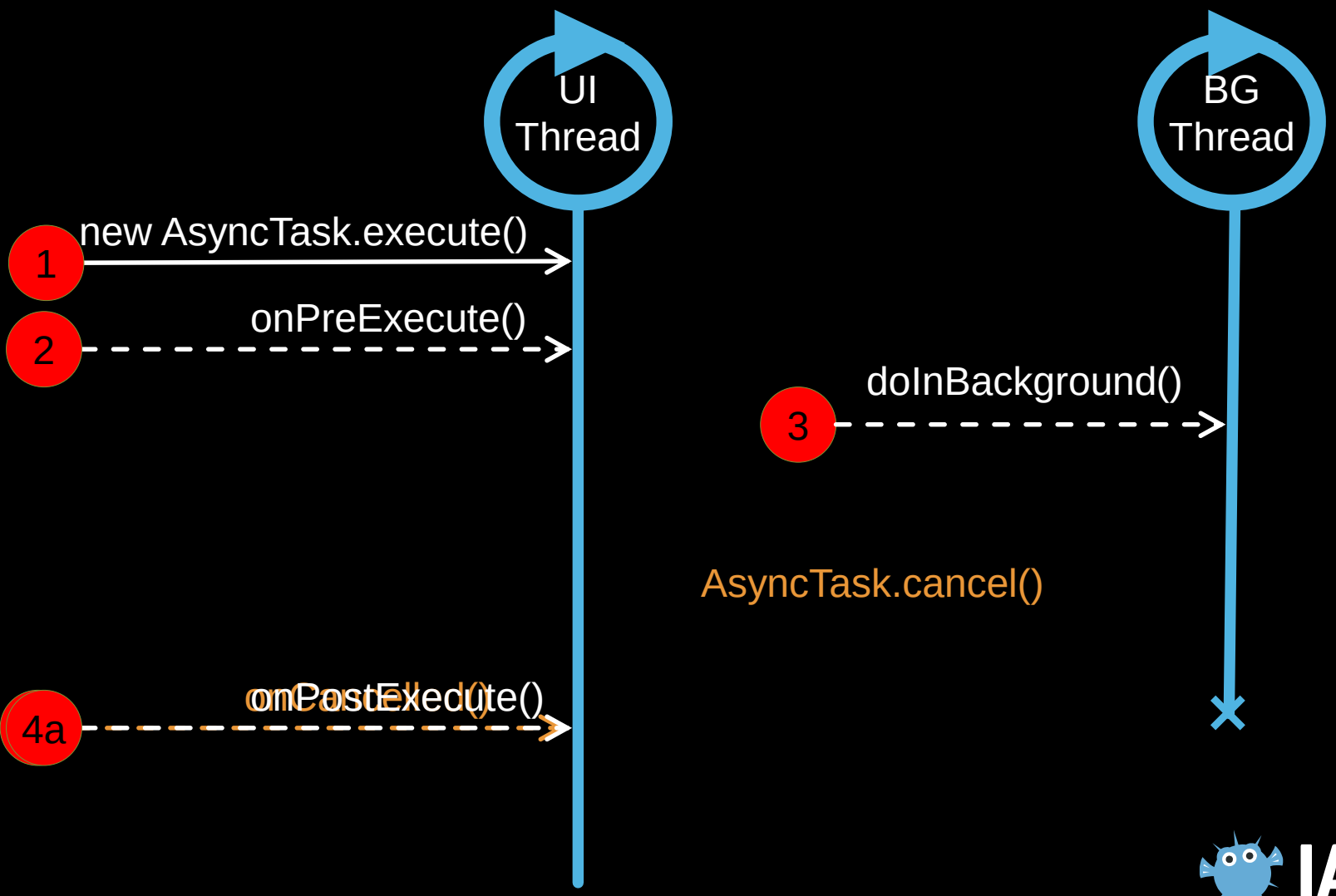
JAYWAY

AsyncTask

- Wraps Handler/Looper thread communication.
- Utilizes the Executor framework.
- Callbacks for UI operations and Background operation.

JAYWAY

How it works



JAYWAY

How it works

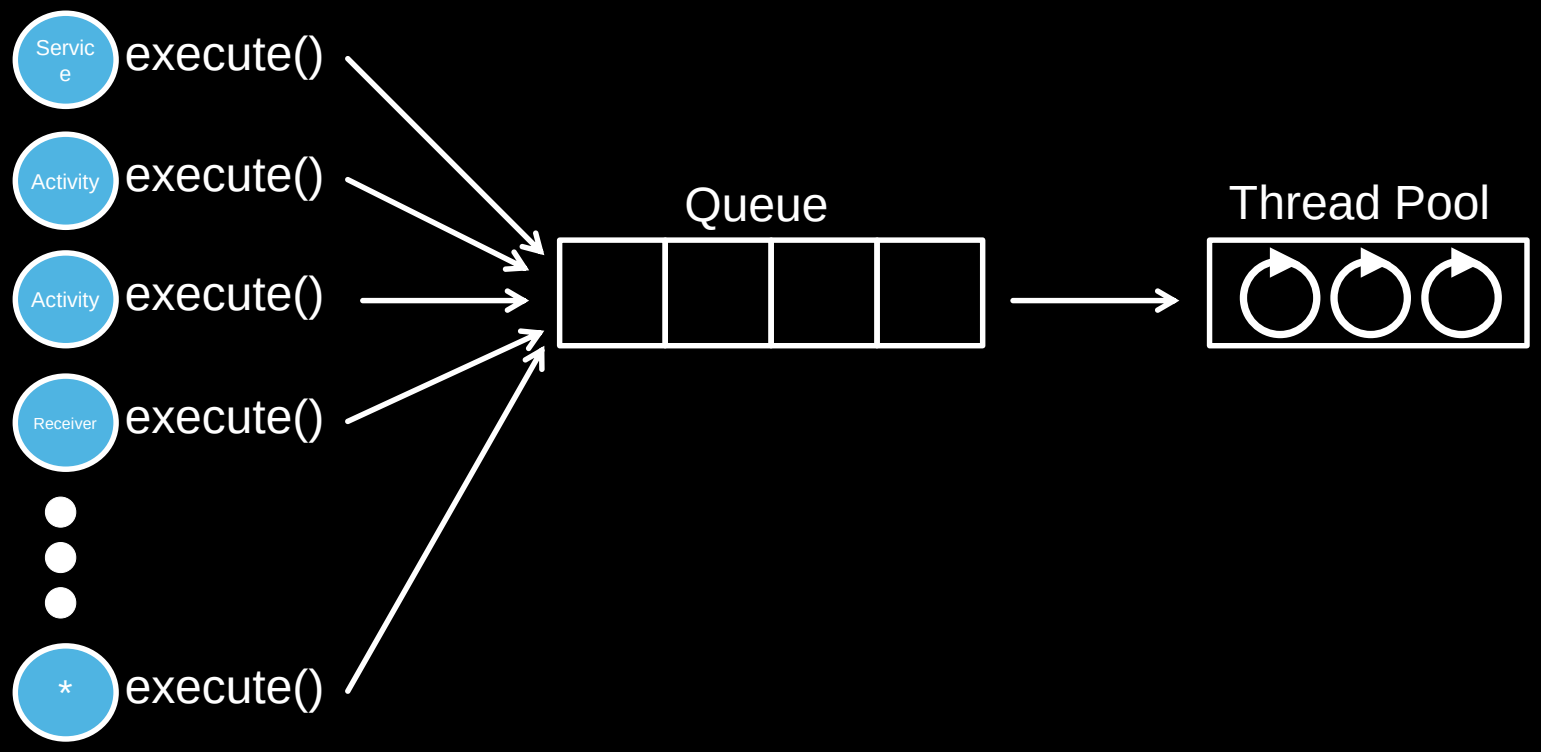
```
public class MyActivity extends Activity {

        public void onButtonClicked(View v) {
                new MyAsyncTask().execute("SomeInputString");
        }
}
```

```
public class MyAsyncTask extends AsyncTask<String, Void, Integer> {

        @Override
        protected void onPreExecute() {
        }

        @Override
        protected Integer doInBackground(String... params) {
                return null;
        }

        @Override
        protected void onCancelled(Integer result) {
        }

        @Override
        protected void onPostExecute(Integer result) {
        }
}
```

Pitfalls

- Application Global Behavior
- execute()
- Cancellation
- Creation and Start

Application Global Behavior



JAYWAY

execute()

Execution behavior has changed over time

< Donut: execute() → [ ][ ][ ][ ] → [ ]

< Honeycomb: execute() → [ ][ ][ ][ ] → [ ][ ][ ]

>= Honeycomb:

execute() → [ ][ ][ ][ ] → [ ]

executeOnExecutor(Executor) → [ ][ ][ ][ ] → [ ][ ][ ]

execute()

"So, if I call AsyncTask.execute on Honeycomb and later my tasks will run sequentially, right?"

"Right?!?"

"Eh, it depends…"

```
<uses-sdk  android:targetSdkVersion="12" />
```

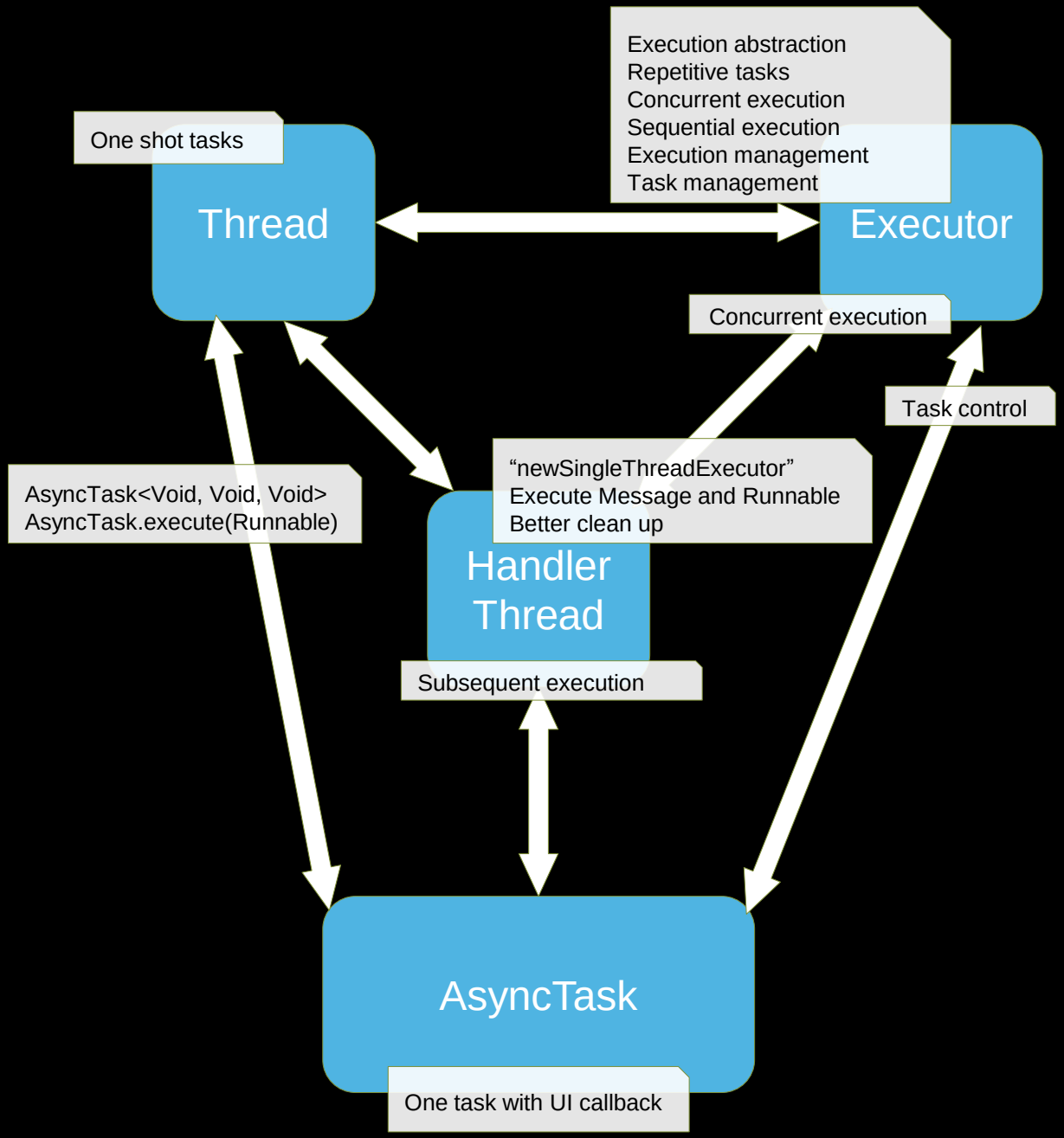
```
<uses-sdk  android:targetSdkVersion="13" />
```

Cancellation

- Cancel AsyncTask when component finishes lifecycle.
  - Avoid UI updates on unavailable component.
- cancel(true) == cancel(false) + interrupt()
- Implement cancellation policy.

Creation and Start

- at = new AsyncTask()
  - The first creation decides callback thread
    - onPostExecute()
    - onPublishProgress()
    - onCancelled()
- at.execute()
  - Callback thread of onPreExecute()


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
Concurrent execution
Task control
AsyncTask<Void, Void, Void>
AsyncTask.execute(Runnable)
"newSingleThreadExecutor"
Execute Message and Runnable
Better clean up
Handler Thread
Subsequent execution
AsyncTask
One task with UI callback

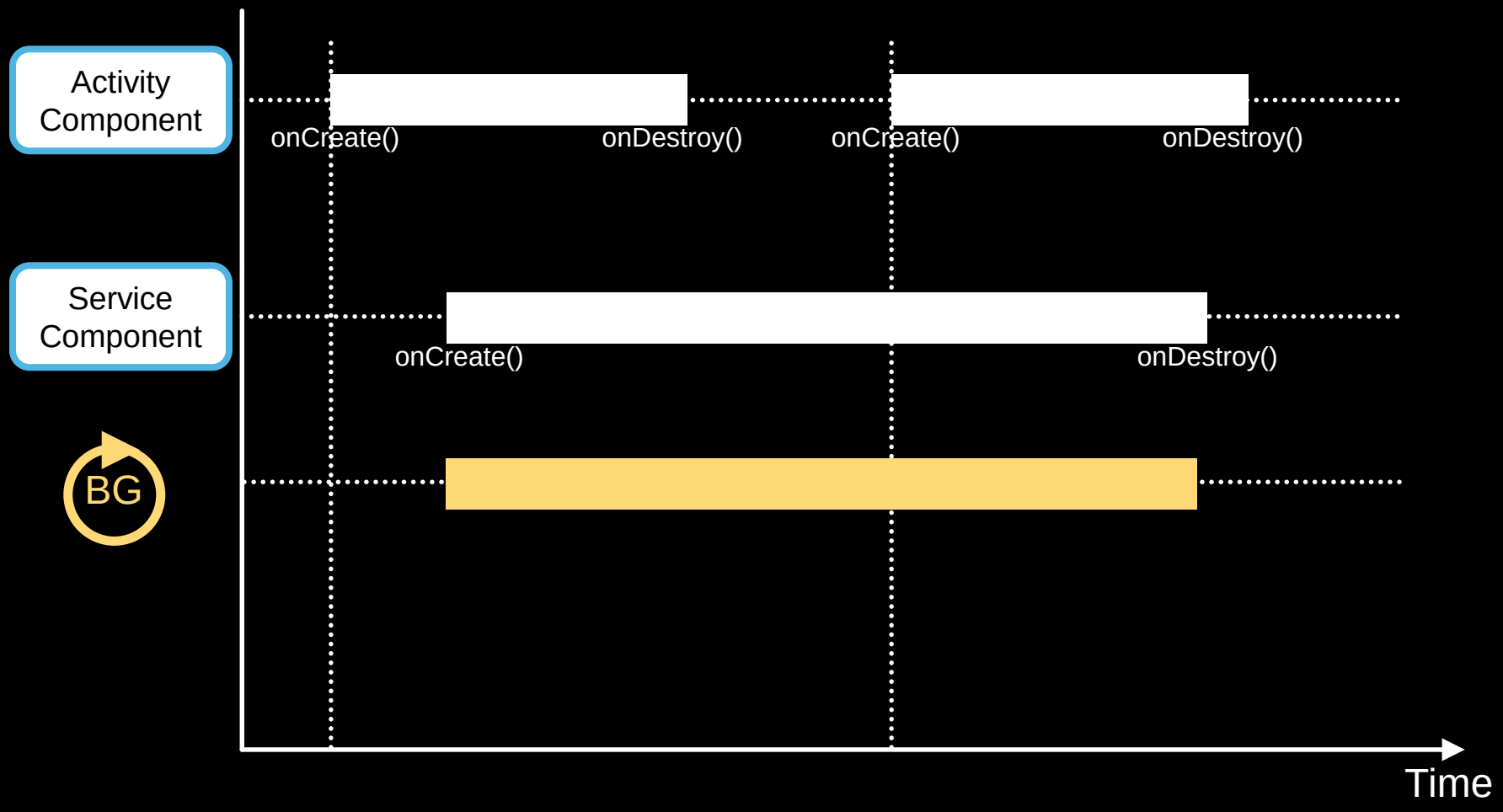
JAYWAY

Service

- Background execution on the UI thread
- Decouple background threads with other lifecycles, typically the Activity lifecycle.

JAYWAY

Example: Lifecycle Decoupling



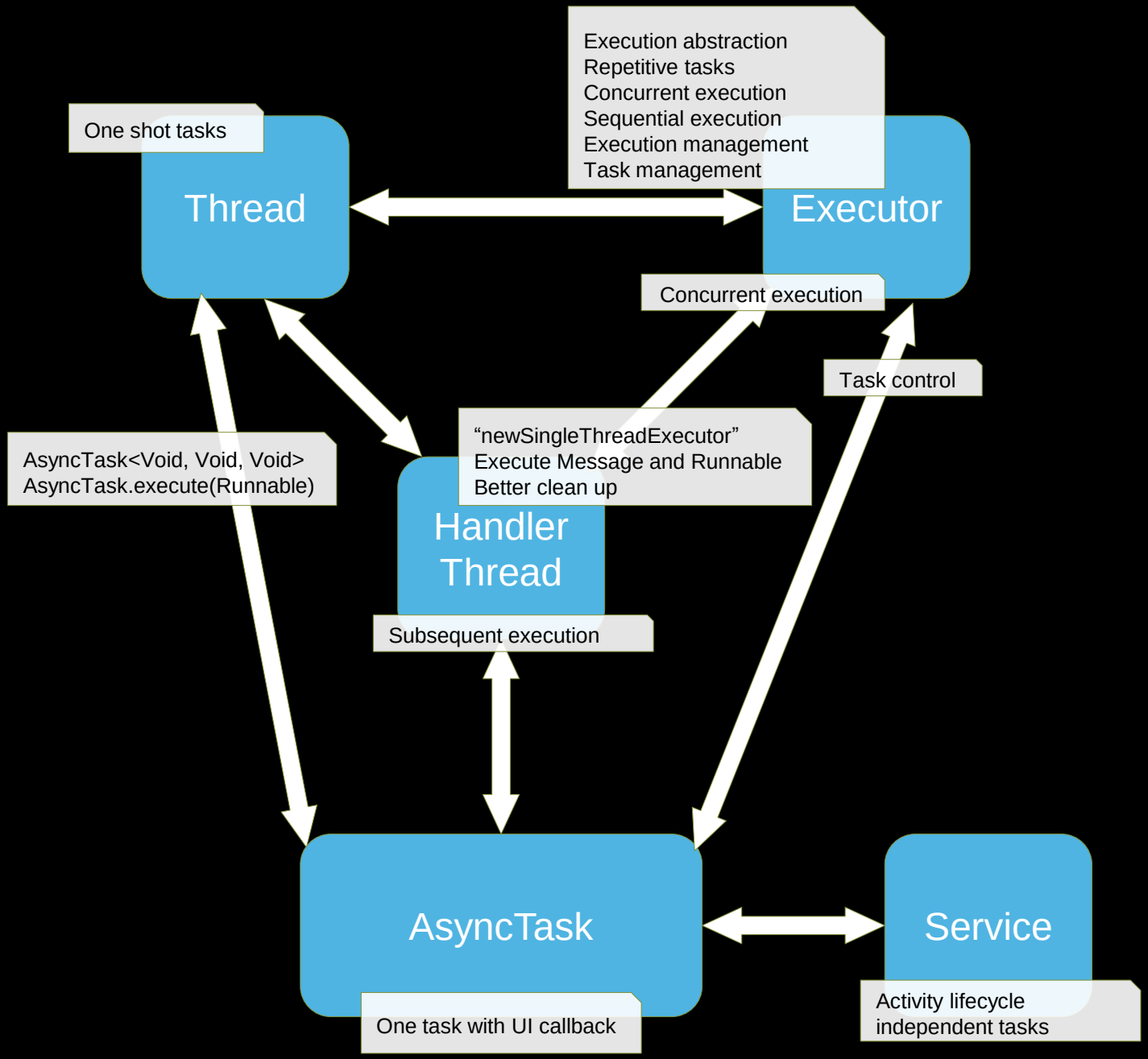
JAYWAY

Good Use Cases

- Tasks executed independently of user interaction.
- Tasks executing over several Activity lifecycles or configuration changes.

Pitfalls

- Hidden AsyncTask.execute() with serial execution.


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
Concurrent execution
Task control
AsyncTask<Void, Void, Void>
AsyncTask.execute(Runnable)
"newSingleThreadExecutor"
Execute Message and Runnable
Better clean up
Handler Thread
Subsequent execution
AsyncTask
Service
One task with UI callback
Activity lifecycle
independent tasks

JAYWAY

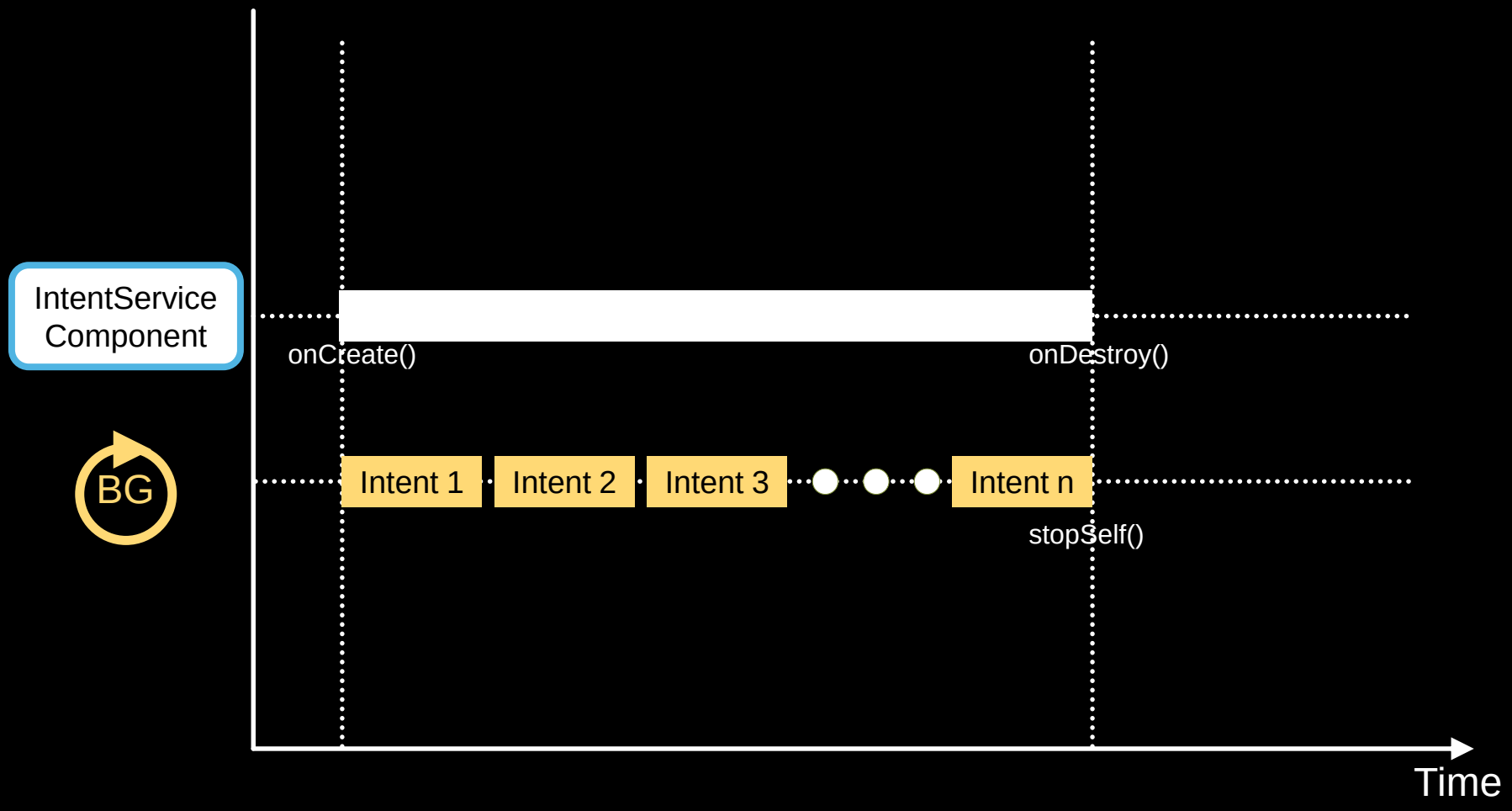
IntentService

- Service with a worker thread.
- On demand intents.

```
public class MyIntentService extends IntentService {

        @Override
        protected void onHandleIntent(Intent intent) {
        }

}
```

JAYWAY

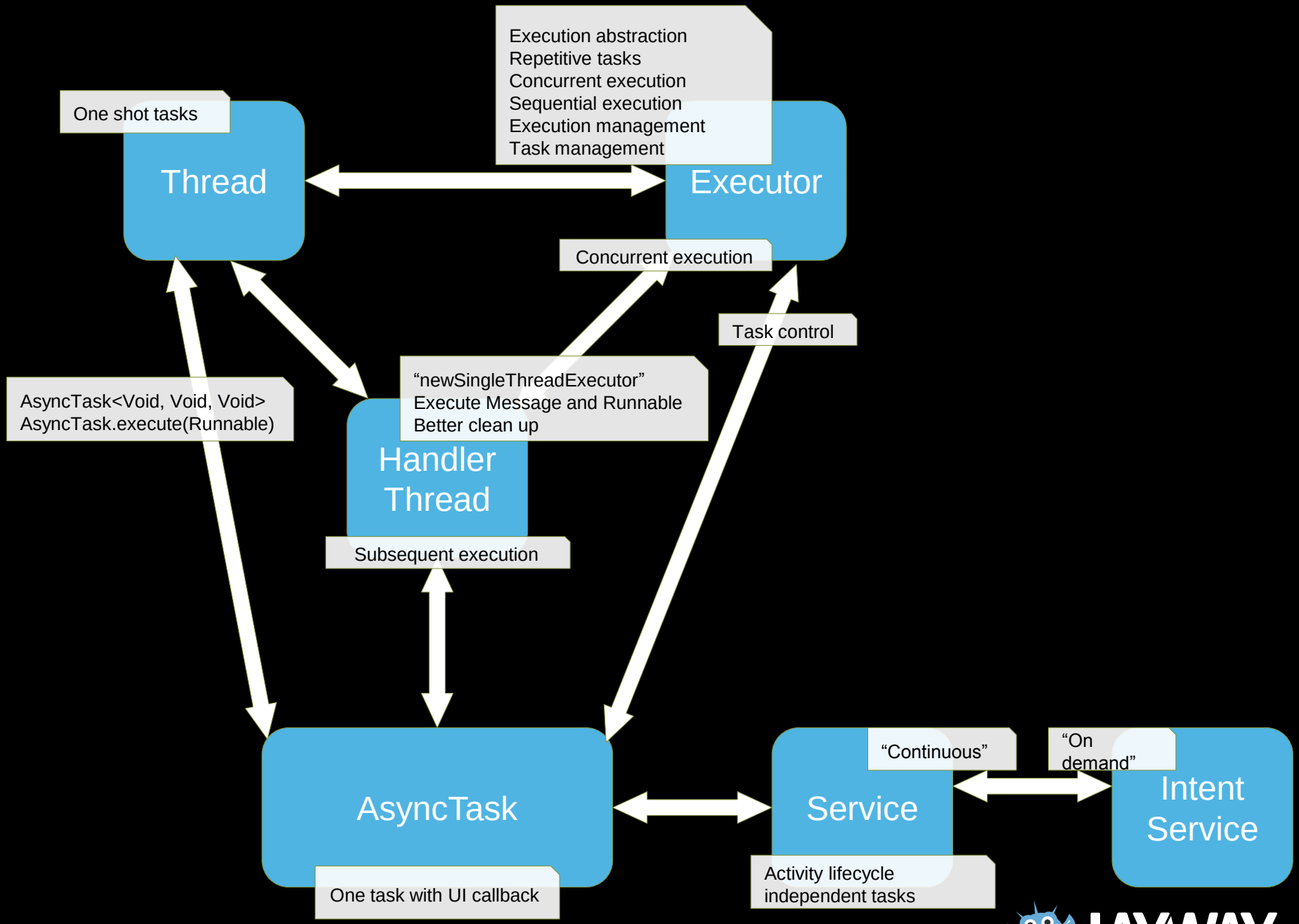
Lifecycle



JAYWAY

Good Use Cases

- Serially executed tasks decoupled from other component lifecycles.
- Off-load UI thread from BroadcastReceiver.
- REST client (ResultReceiver as callback)


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
Concurrent execution
Task control
AsyncTask<Void, Void, Void>
AsyncTask.execute(Runnable)
"newSingleThreadExecutor"
Execute Message and Runnable
Better clean up
Handler Thread
Subsequent execution
AsyncTask
One task with UI callback
"Continuous"
"On demand"
Service
Intent Service
Activity lifecycle independent tasks

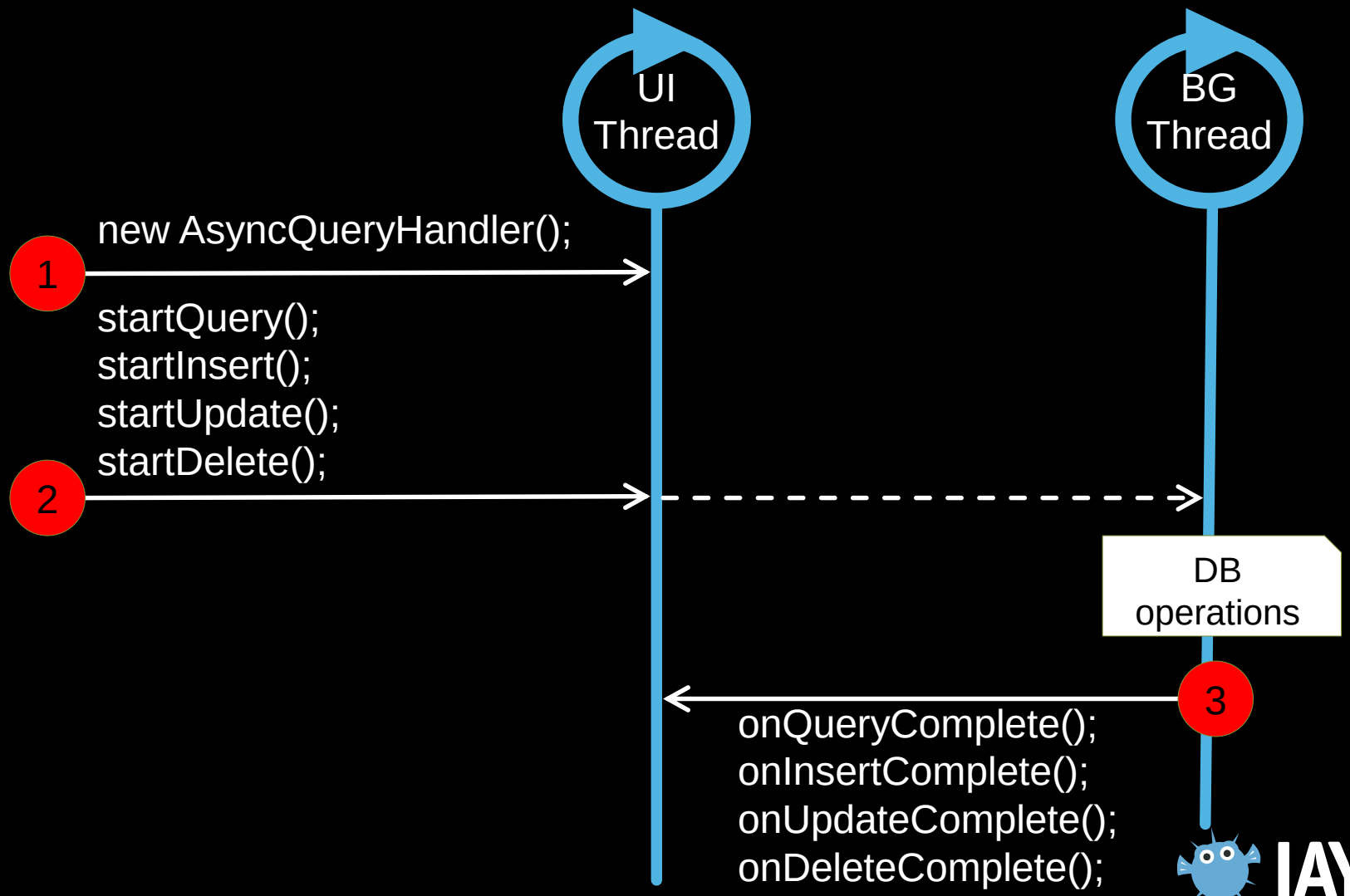
JAYWAY

AsyncQueryHandler

- API Level 1
- Asynchronous operations on a ContentResolver
  - Query
  - Insert
  - Delete
  - Update
- Wraps a HandlerThread

JAYWAY

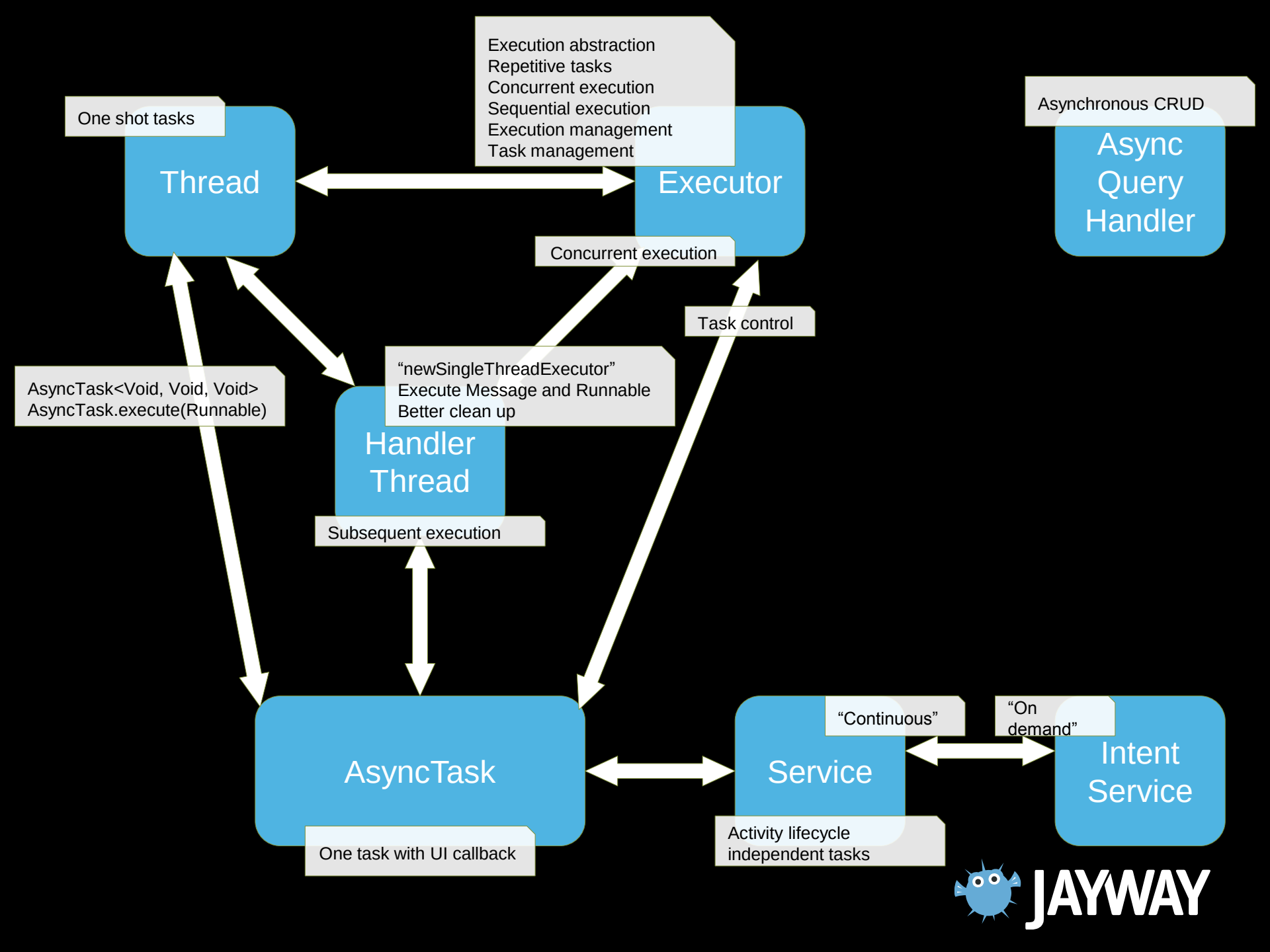
How it works



JAYWAY

Cons

- No cursor management
- No content observation
- No data retention on configuration changes
- Background thread can’t be forced to quit

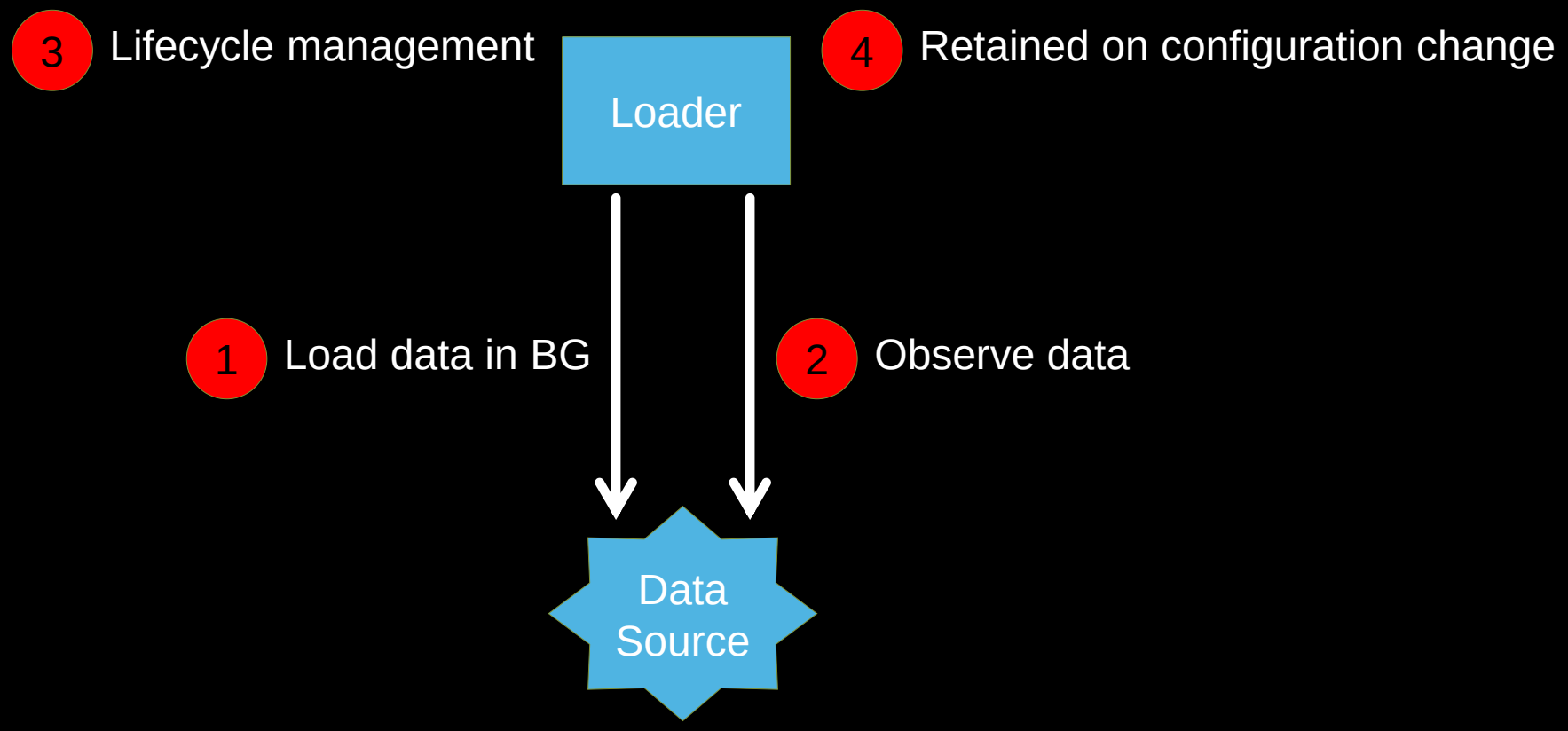

Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
Asynchronous CRUD
Async Query Handler
Concurrent execution
Task control
AsyncTask<Void, Void, Void>
AsyncTask.execute(Runnable)
"newSingleThreadExecutor"
Execute Message and Runnable
Better clean up
Handler Thread
Subsequent execution
AsyncTask
One task with UI callback
"Continuous"
"On demand"
Service
Intent Service
Activity lifecycle independent tasks
JAYWAY

Loader

- API added in Honeycomb
- Available in compatibility package
- Load data in a background thread
- Observes data changes
- Retained on configuration changes
- Connected to the Activity and Fragment lifecycles

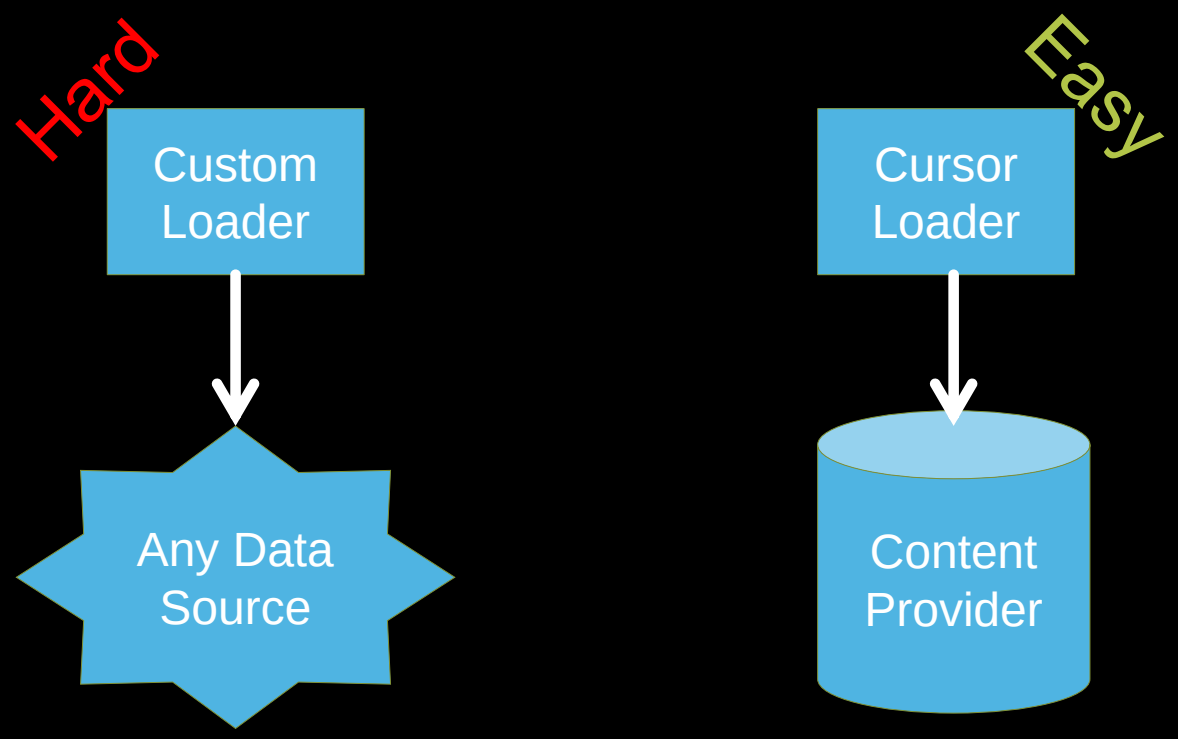
JAYWAY

Basics



JAYWAY

Data Sources



JAYWAY

How It Works

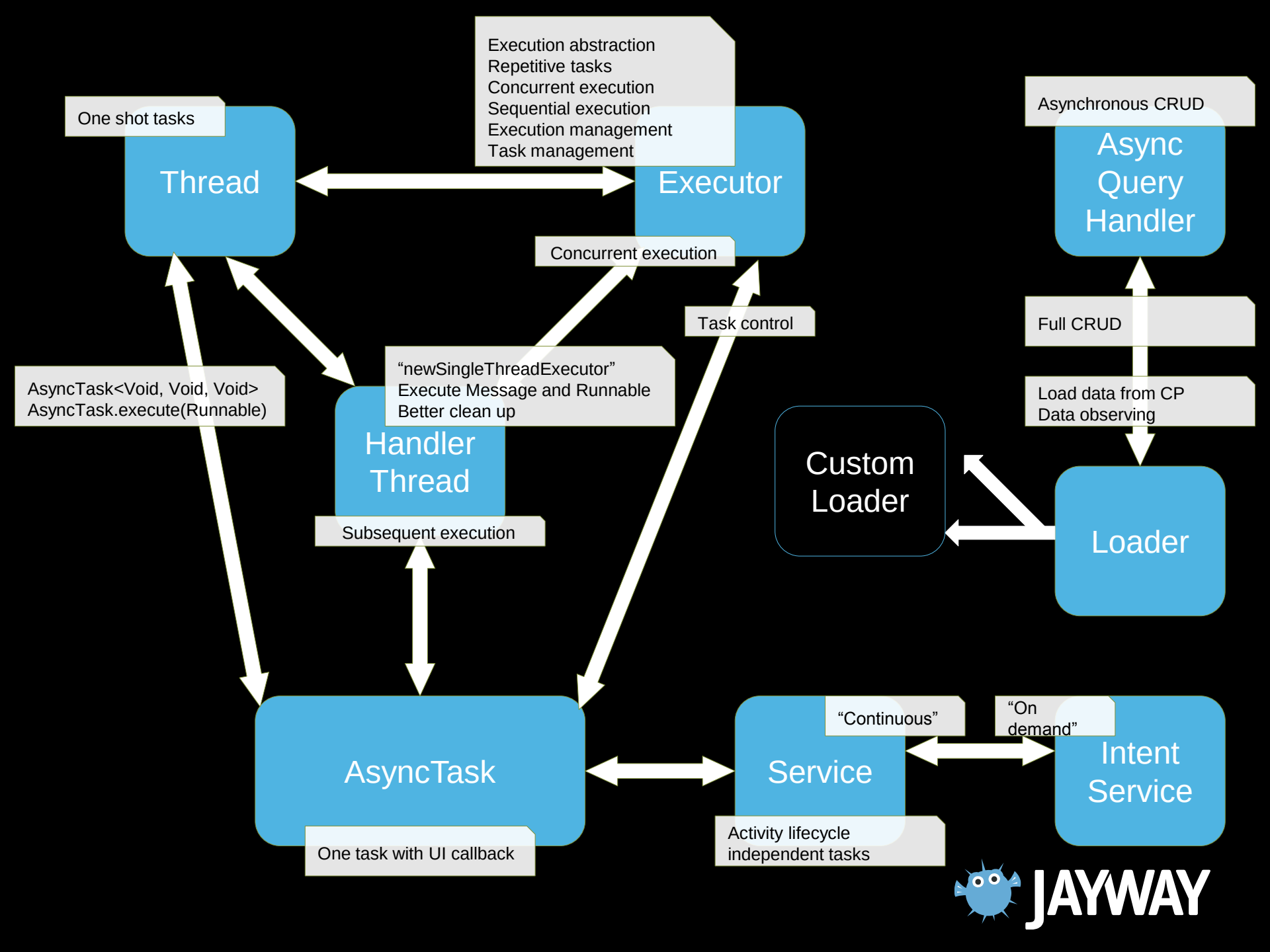
```
public class AndroidLoaderActivity extends ListActivity implements LoaderCallbacks<Cursor>{

        SimpleCursorAdapter mAdapter;

        public void onCreate(Bundle savedInstanceState) {
                getLoaderManager().initLoader(0, null, this);
        }

        @Override
        public Loader<Cursor> onCreateLoader(int id, Bundle args) {
                return new CursorLoader(..., CONTENT_URI, ...);
        }

        @Override
        public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
                mAdapter.swapCursor(c);
        }

        @Override
        public void onLoaderReset(Loader<Cursor> arg0) {
                mAdapter.swapCursor(null);
        }
}
```

Pitfalls

- Data loading will stop when Activity/Fragment is destroyed
  - Http requests not finished

Good Use Cases

- Loading data from Content Providers.


Execution abstraction
Repetitive tasks
Concurrent execution
Sequential execution
Execution management
Task management
One shot tasks
Thread
Executor
Asynchronous CRUD
Async Query Handler
Concurrent execution
Task control
Full CRUD
AsyncTask<Void, Void, Void>
AsyncTask.execute(Runnable)
"newSingleThreadExecutor"
Execute Message and Runnable
Better clean up
Handler Thread
Load data from CP
Data observing
Custom Loader
Loader
Subsequent execution
AsyncTask
One task with UI callback
"Continuous"
"On demand"
Service
Intent Service
Activity lifecycle
independent tasks
JAYWAY

Thank you for listening!

Questions?

JAYWAY