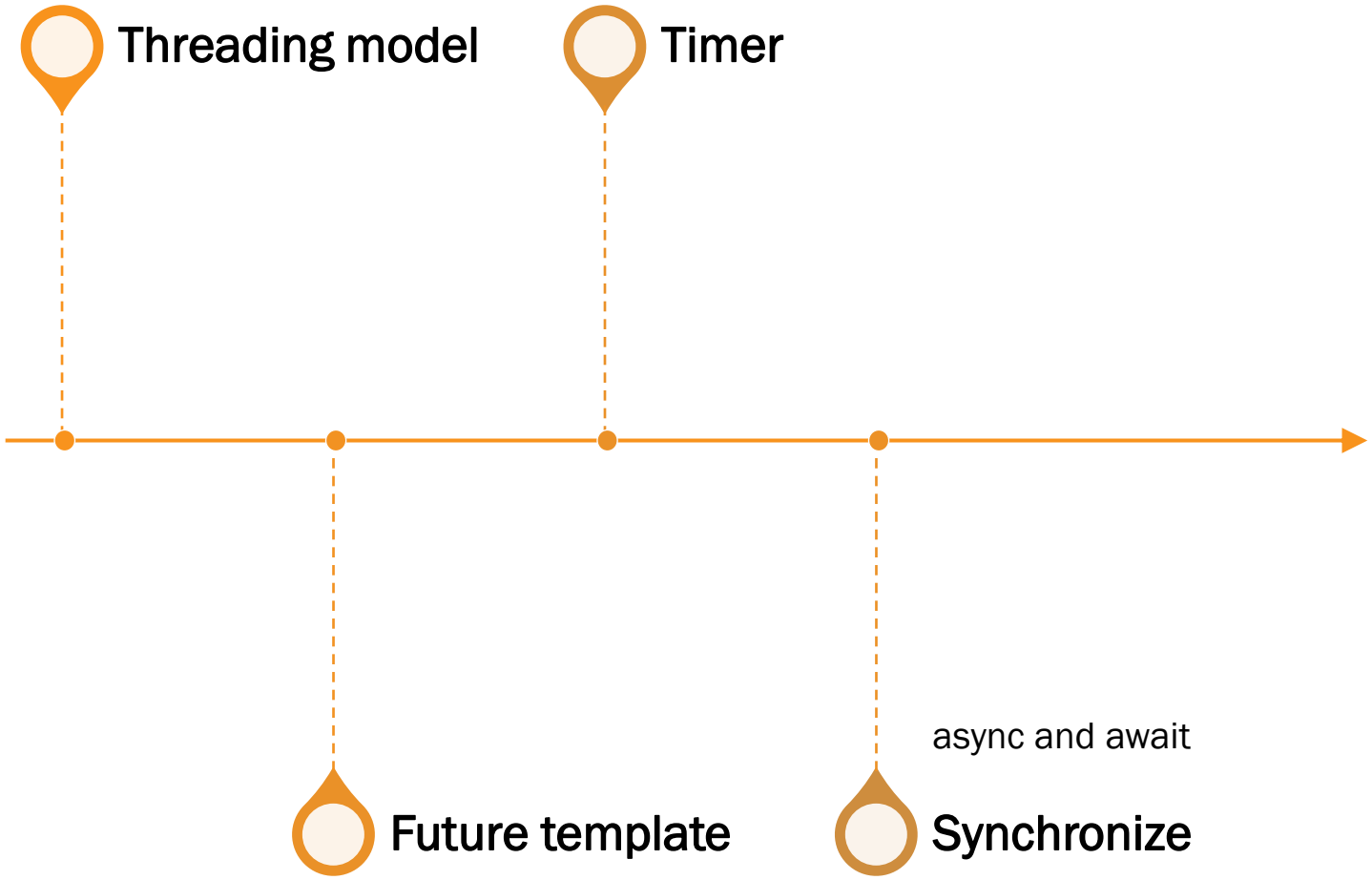
DART Language

COURSE 10 (REV 3)

GAVRILUT DRAGOS

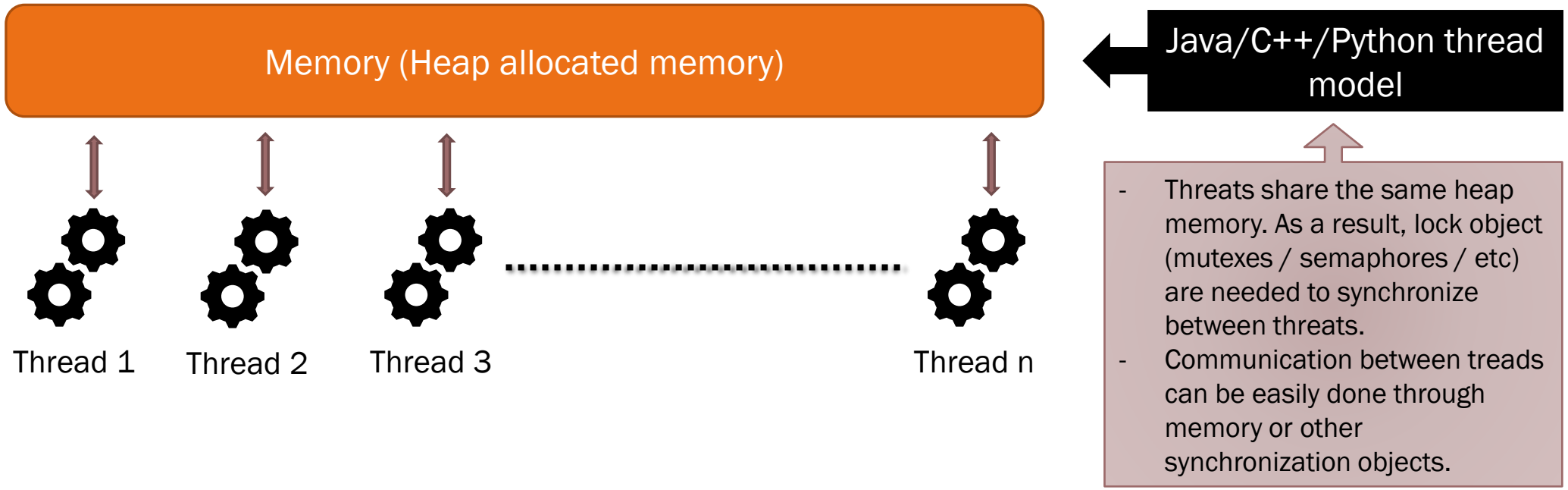
Agenda


Threading model
Timer
Future template
async and await
Synchronize

Threading model

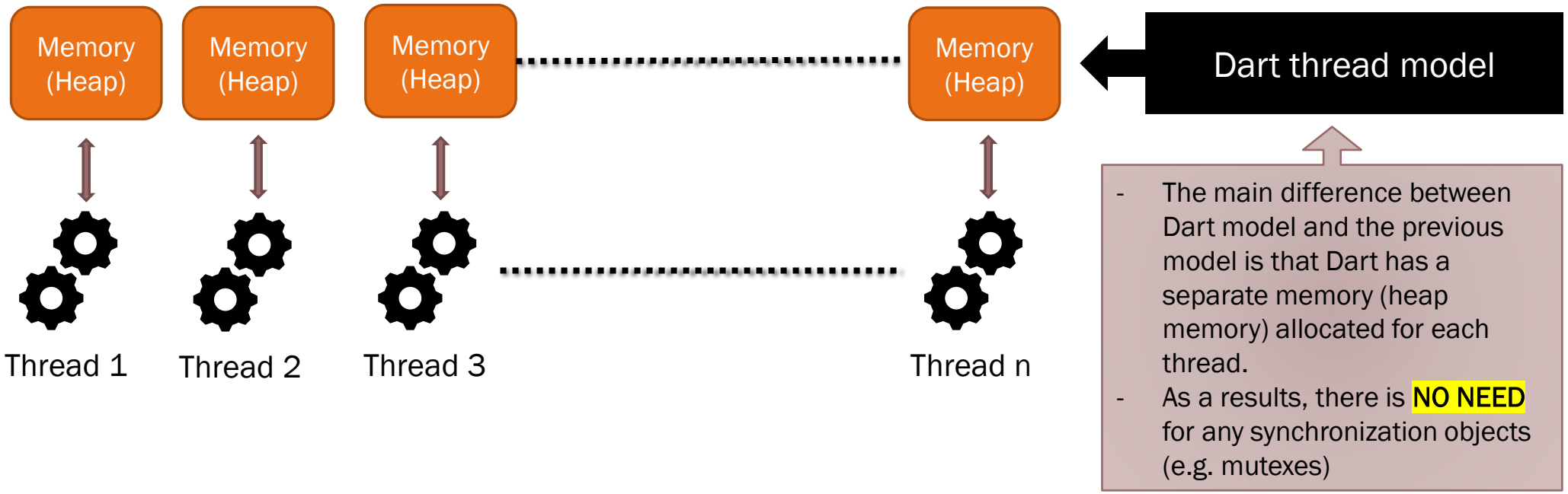
Threading model

Generically, most programming languages have a multi-threating support. Dart has one too, but with some differences.



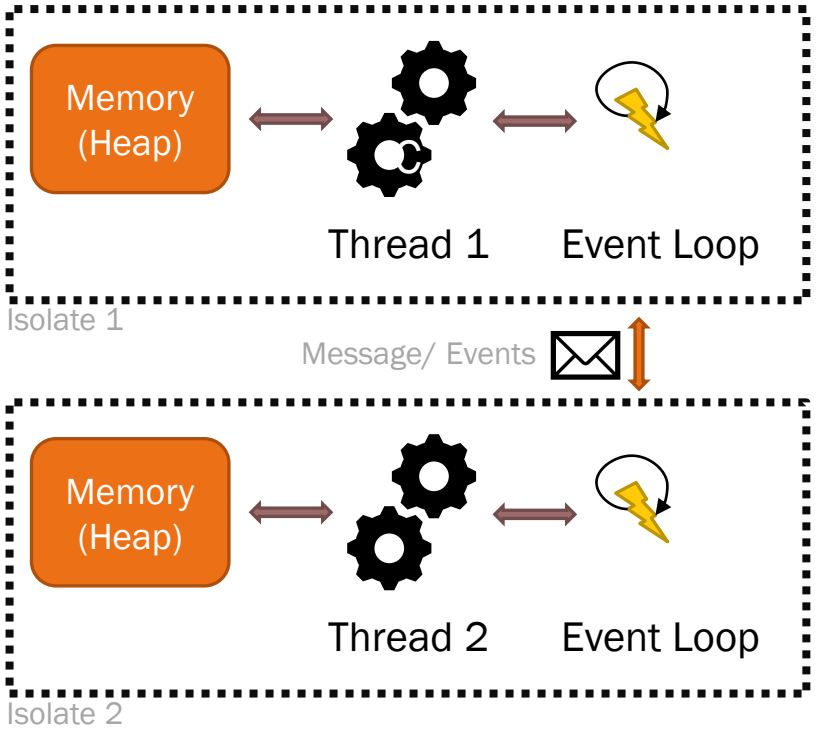
Threading model

Generically, most programming languages have a multi-threating support. Dart has one too, but with some differences.



Threading model

Generically, most programming languages have a multi-threating support. Dart has one too, but with some differences.



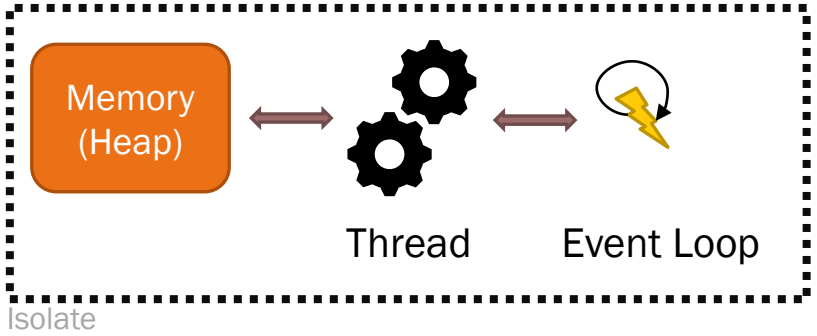
An Isolate

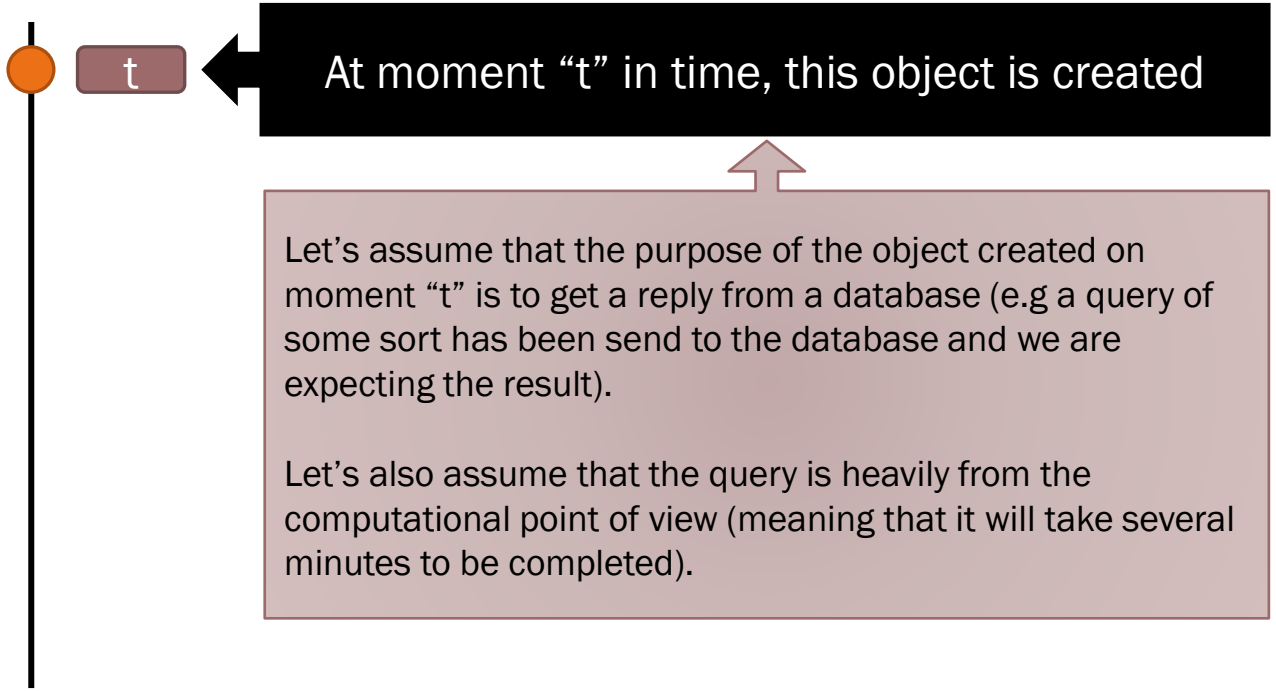
- An isolate uses an event loop to process messages or I/O events
- Multiple isolates can communicate between them in a similar manner (by sending an event to the event loop).
- The memory used by an isolate is his own (meaning that another isolate can not access any object created by another isolate).
- Any DART program has at least (an most of the time) only one isolate (the one that represent the function main).

Future object

Futures

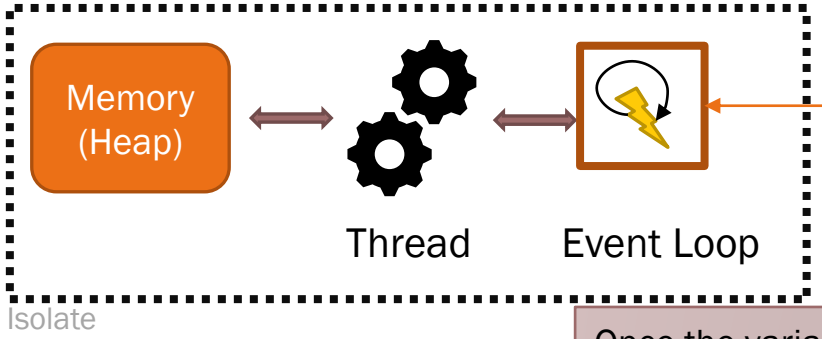
A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.





Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



t

At moment “t” in time, this object is created

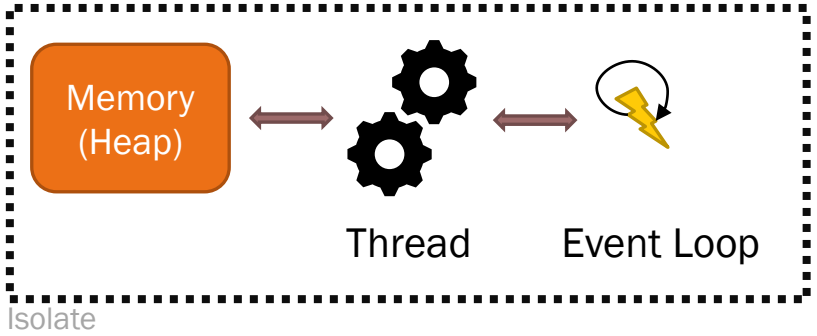
```
var myFuture = SomeFunctionThatReturnsAFuture();
```

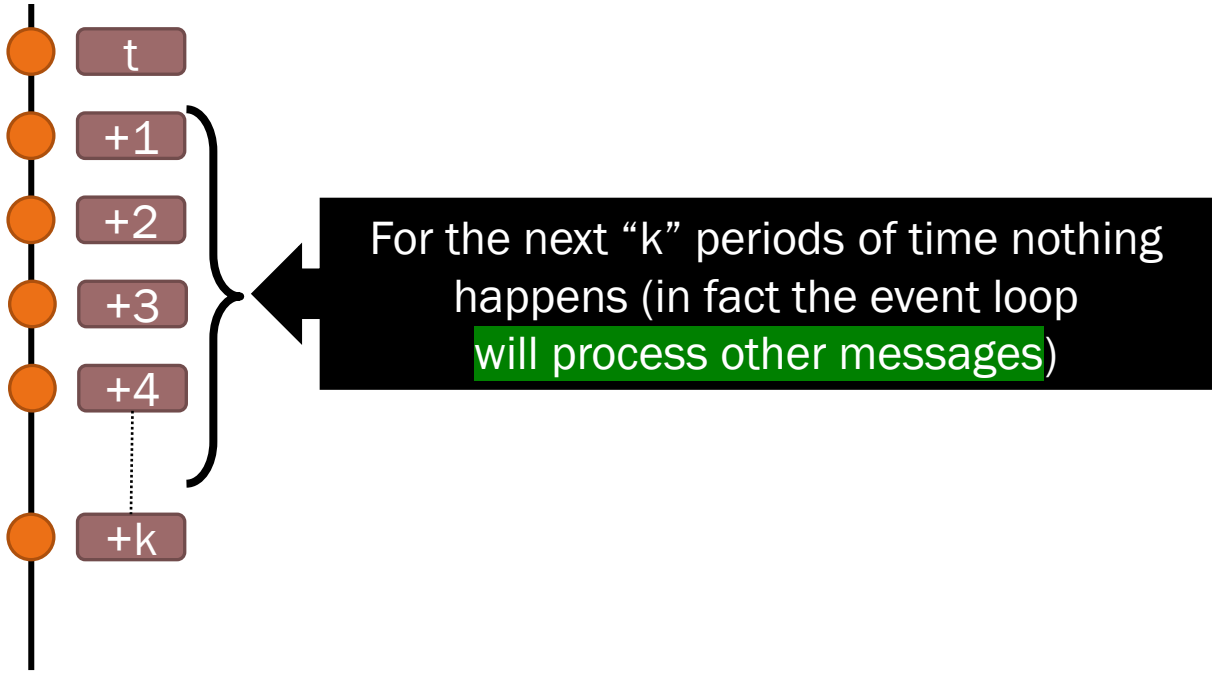
Once the variable is created (in our case myFyuture) the event loop will be notified that there is a possibility in the future for this variable to trigger an event (e.g. upon receiving the data from the database query).

This is important as the event loop will not close until this future is completed.

Futures

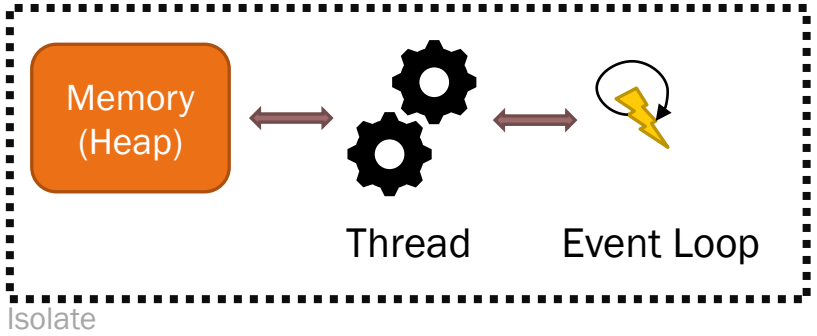
A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.

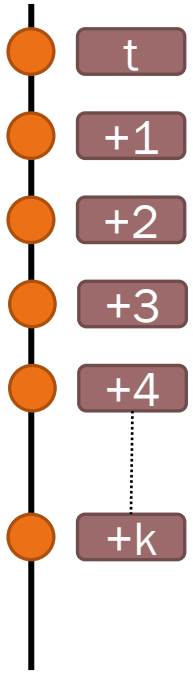




Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.

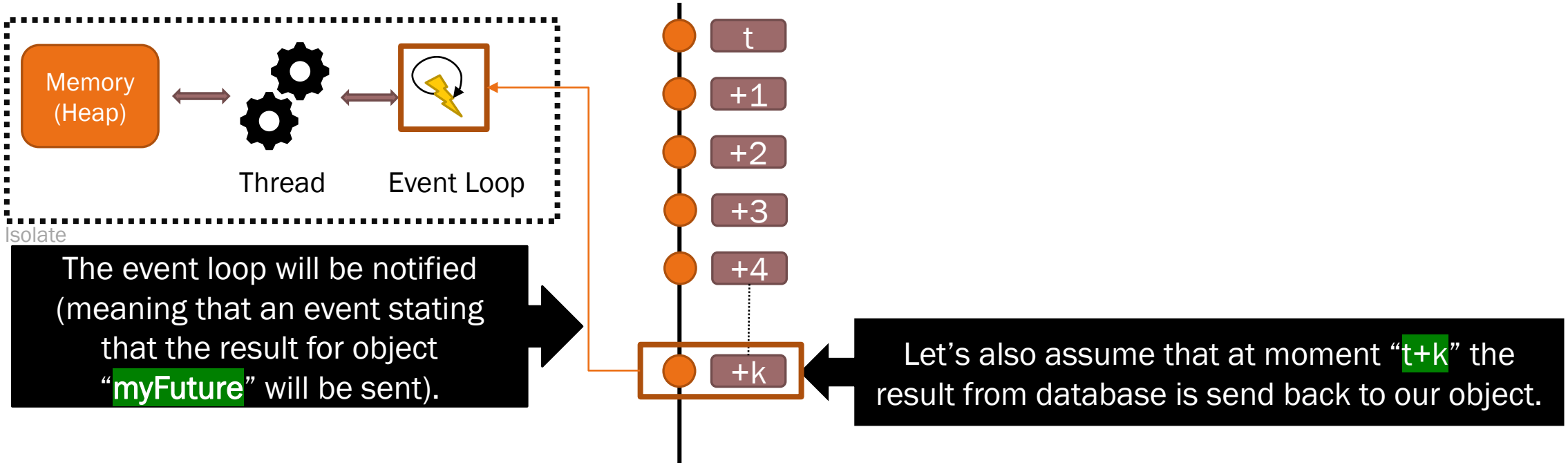




Let's also assume that at moment "t+k" the result from database is send back to our object.

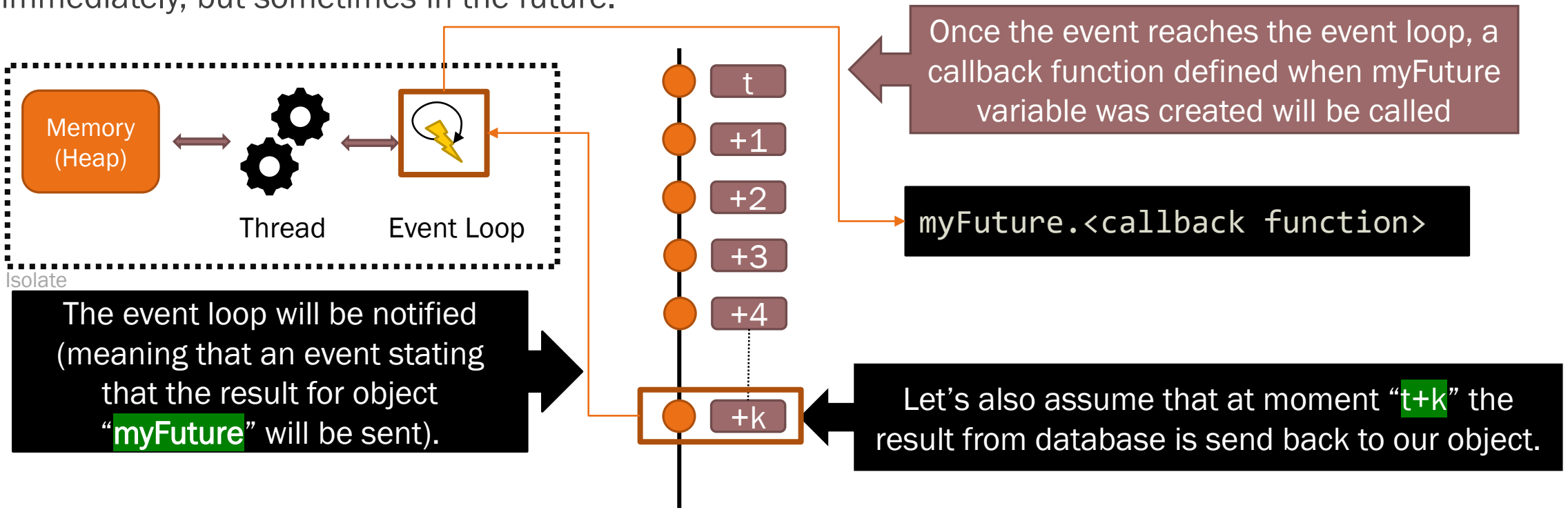
Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



Futures

A future object is a place-holder (a container) that will have that object you are requestion not immediately, but sometimes in the future.



Futures

A future object is a template and can be constructed in the following way:

```
Future<T> (FutureOr<T> computation())
Future<T>.delayed (Duration duration, [FutureOr<T> computation()])
Future<T>.synk (FutureOr<T> computation())
Future<T>.value ([FutureOr<T>? value])
```

Where *FutureOr<T>* can be either a value of type T or an object of type Future<T>.

Dart compiler will throw an error if any class tries to extend / implement or mix a FutureOr<> class.

Out of the above constructors, Future<T>.delayed implies that the computation code will be executed after a specific period of time.

To use/create a future object import “dart:async” library.

Futures

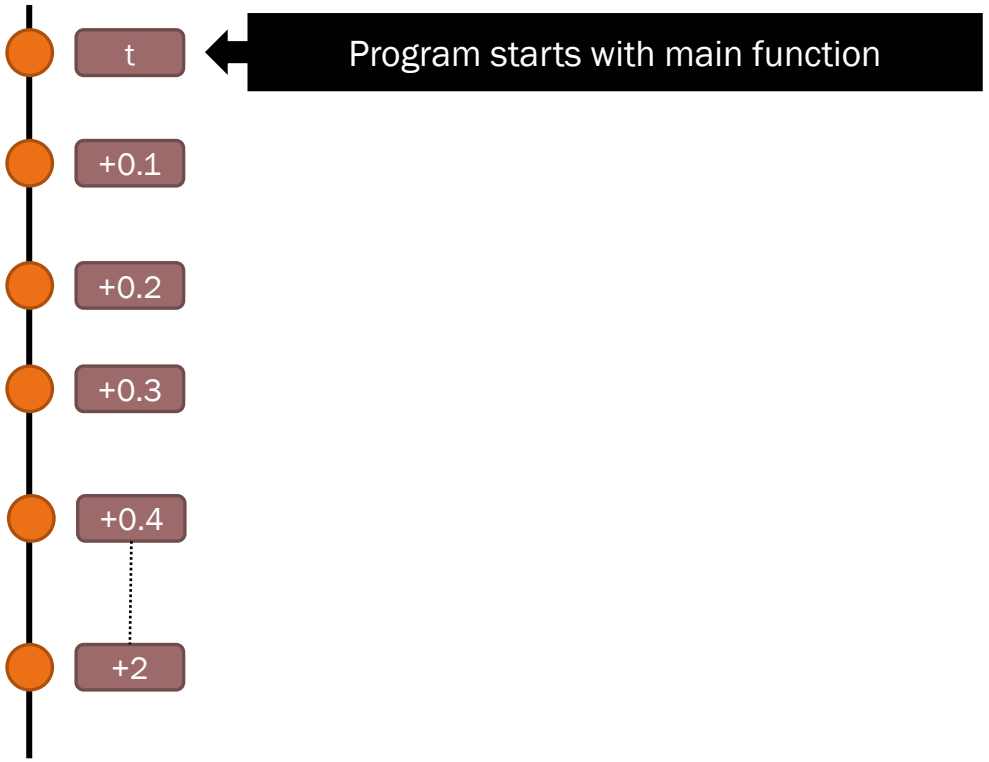
A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```

Futures

A Future example:

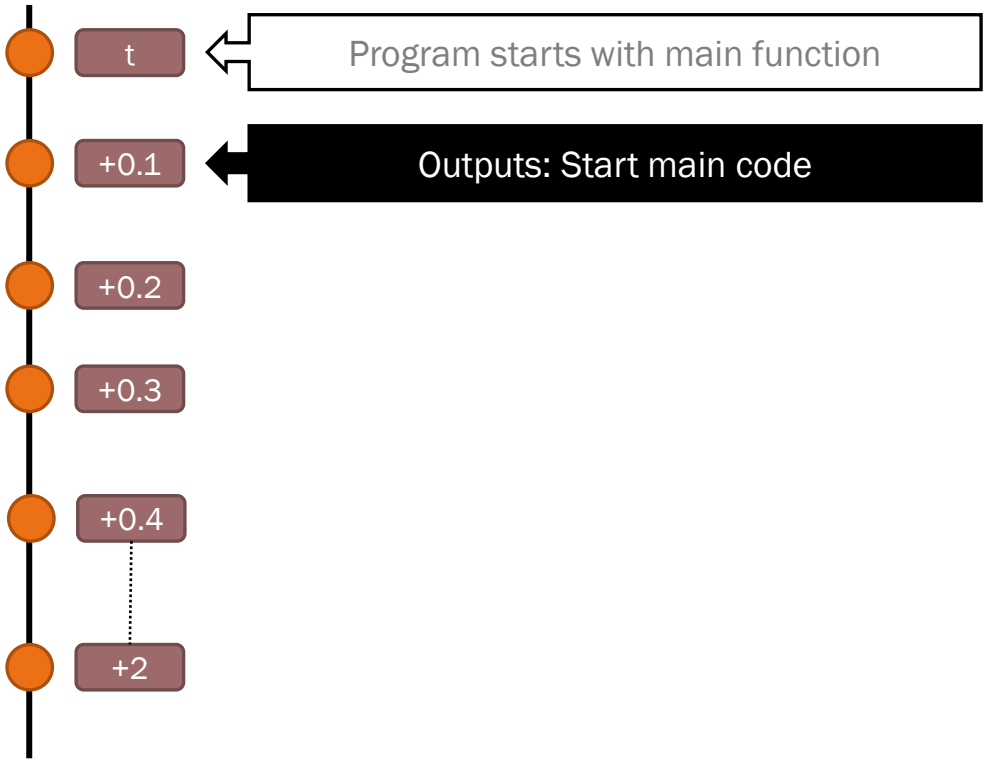
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

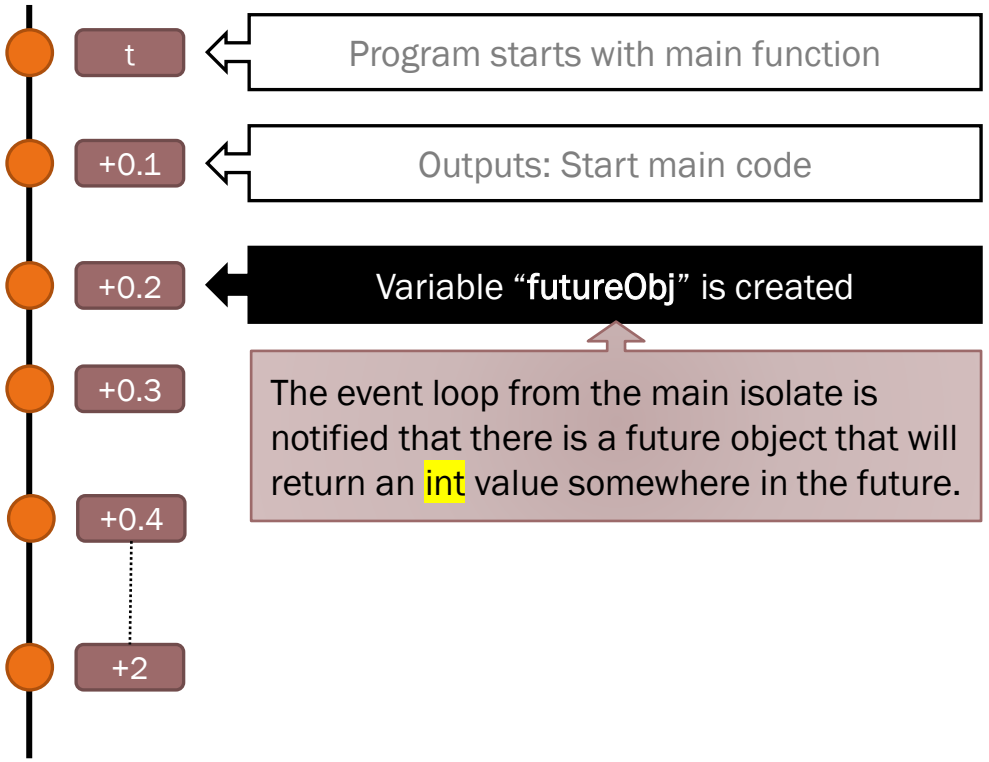
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

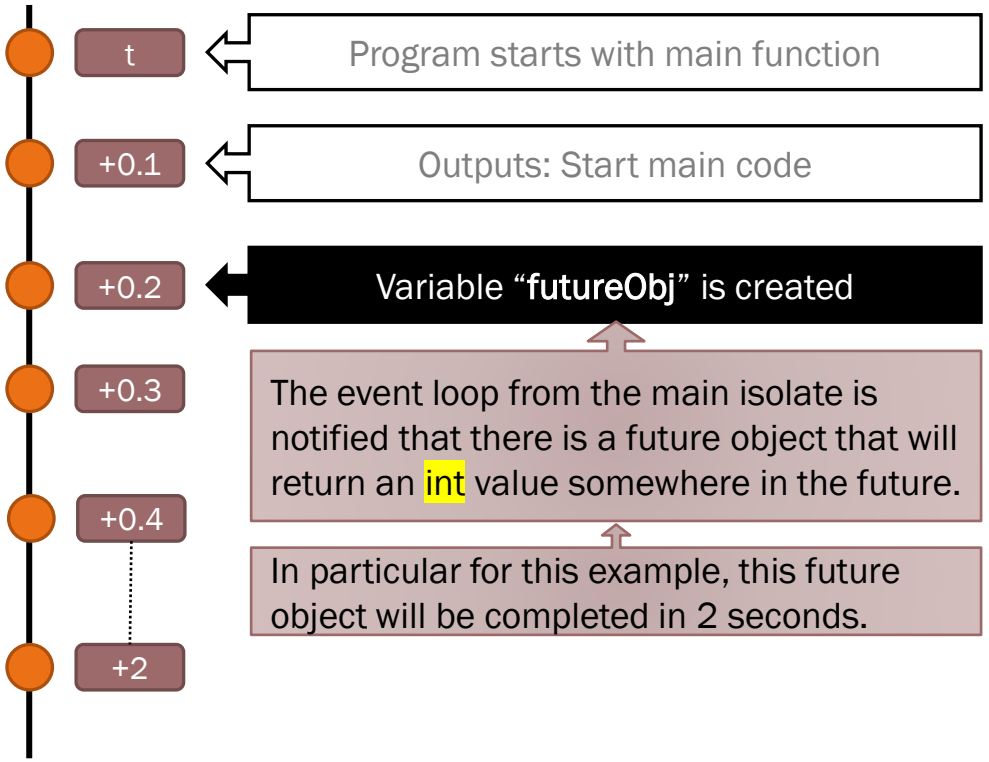
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                        Duration(seconds: 2),
                        CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

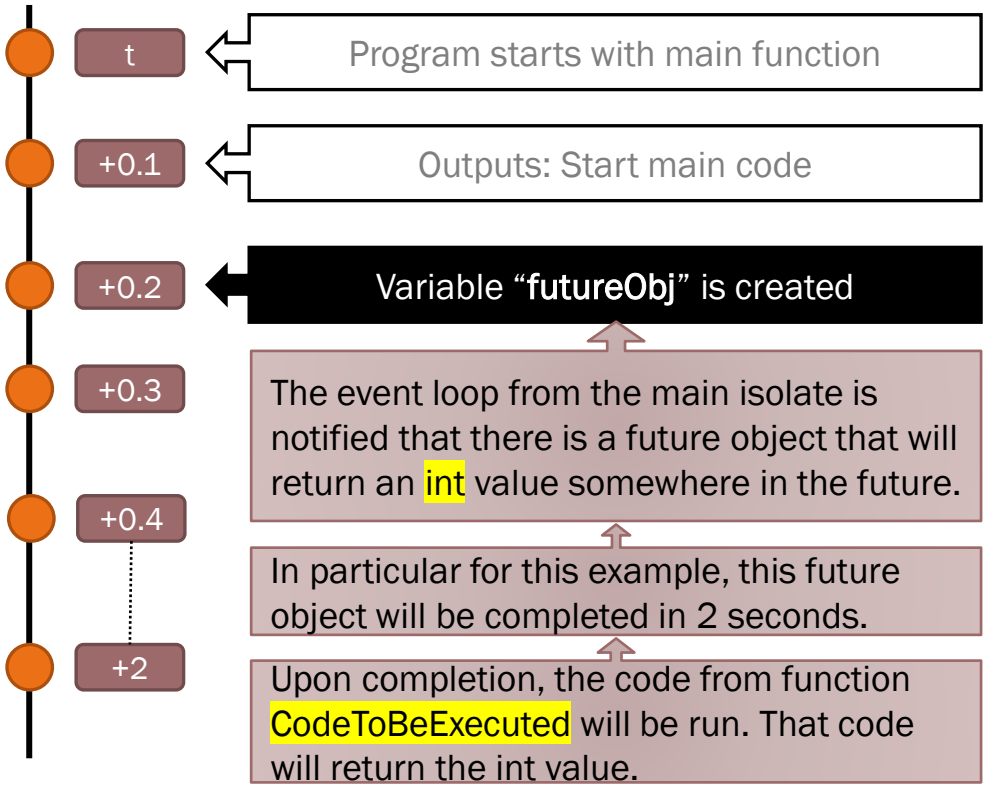
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                    Duration(seconds: 2),
                    CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

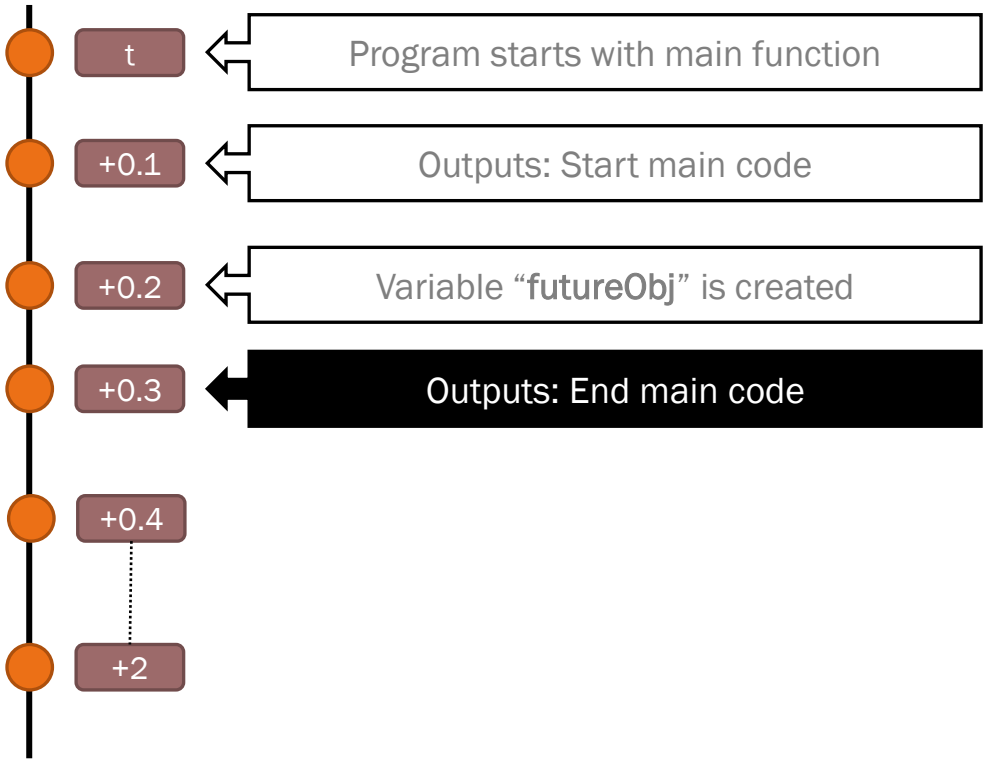
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                    Duration(seconds: 2),
                    CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

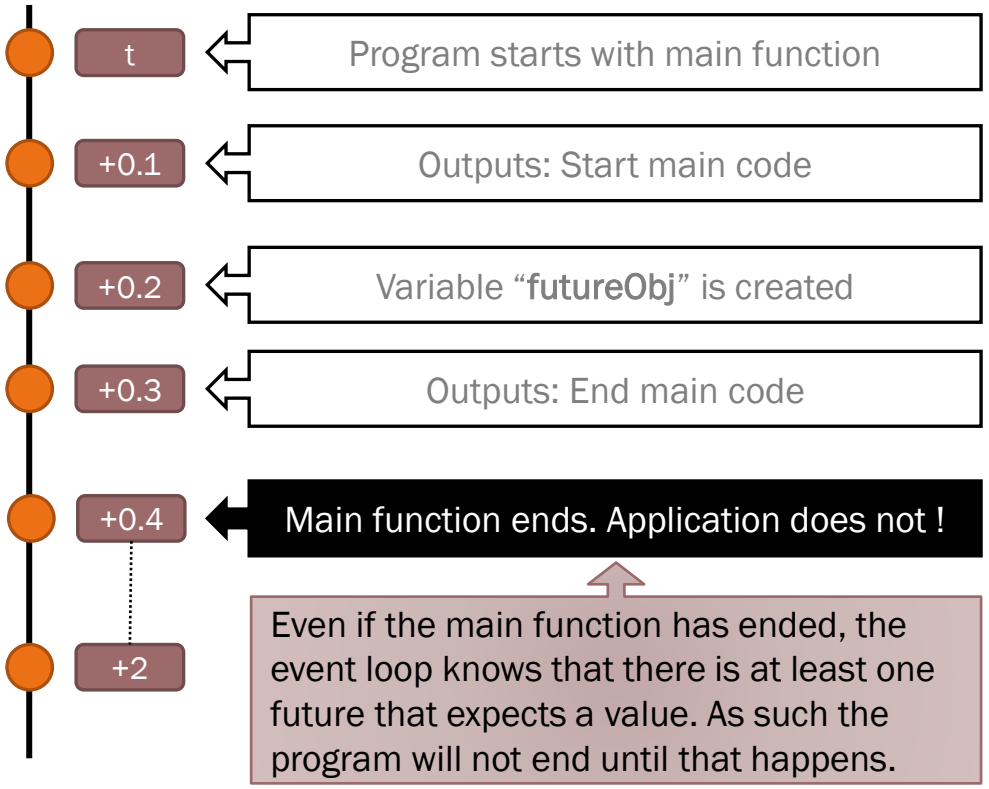
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

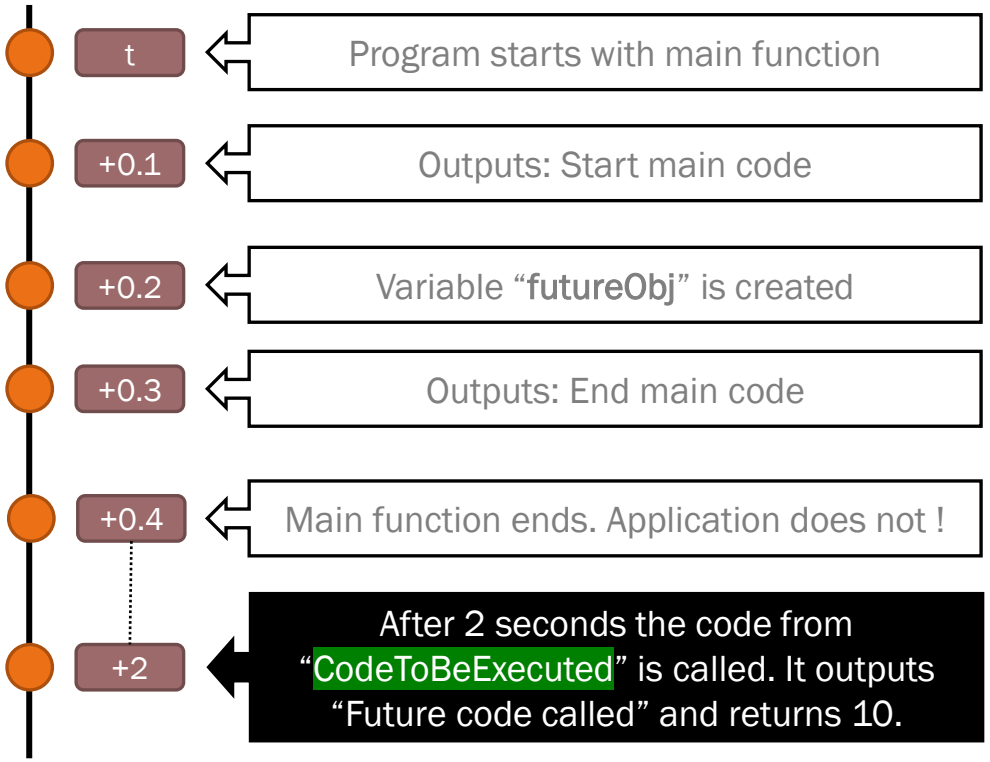
```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```



Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
  print("Future code called");
  return 10;
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(
                      Duration(seconds: 2),
                      CodeToBeExecuted);
  print("End main code");
}
```



Futures

To get a callback/be notified when a futures completes, use the method **then**

```
Future<T> then<T> (FutureOr<T> onValue(T value), {Function? onError})
```

This method will be called when the future object completes and the value it return is passed on to the **onValue** callback.

```
import "dart:async";
void processValue(int value) {
  print("Value received is ${value}");
}
void main() {
  print("Start main code");
  var futureObj = Future<int>.delayed(Duration(seconds: 2),()=>10);
  futureObj.then(processValue);
  print("End main code");
}
```

Output:
Start main code
End main code
Value received is 10

Futures

Method .then(...) can be used to linked a future with another one.

```
import "dart:async";
FutureOr<int> SecondFuture(int value) {
   print("Second future -> value=${value}");
   return 0;
}
FutureOr<int> FirstFuture(int value) {
   print("First future -> value=${value}");
   return Future<int>.delayed(Duration(seconds:value),()=>2)
                     .then(SecondFuture);
}
void main() {
   print("Start");
   Future<int>.delayed(Duration(seconds: 2),()=>4).then(FirstFuture);
}
```

Output:

```
Start
First future -> value=4
Second future -> value=2
```

Futures

Dart Future<T>.value named constructor can be used to return a value (but after the current execution ends).

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  for (var i=0;i<10;i++) {
      Future<int>.value(i).then((value) => print(value));
  }
  stdin.readLineSync();
  print("End");
}
```

Output:

```
Start
End
0
1
2
3
4
5
6
7
8
9
```

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
            .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen

2. Wait for a line to be typed from the keyboard followed by the ENTER key

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen

2. Wait for a line to be typed from the keyboard followed by the ENTER key

3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen

2. Wait for a line to be typed from the keyboard followed by the ENTER key

3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.

4. Wait for ENTER to be pressed (more than 10 seconds).

In theory, those 2 seconds from that Future object would pass, and the value 2 should be printed ! In practice nothing happens.

Futures

Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.
4. Wait for ENTER to be pressed (more than 10 seconds).
5. End is outputted to the screen

Futures

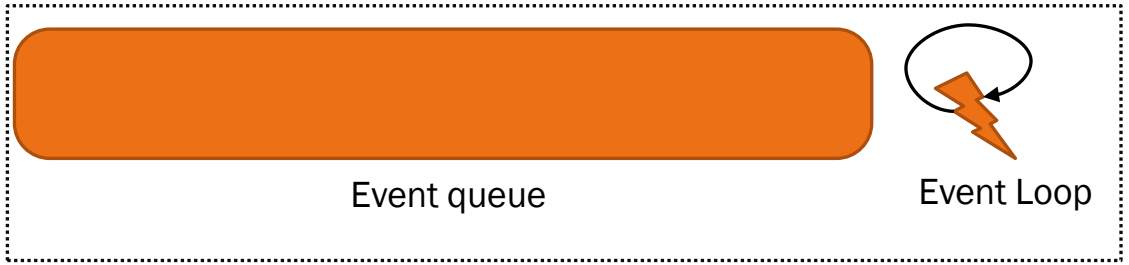
Let's analyze the following piece of code and work with the assumption that on each stdin.readLineSync() we will wait 10 seconds before we introduce a string and press Enter. The following code was compiled and tested with: Dart SDK version: 2.15.1 (stable)

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2),()=>2)
              .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.
4. Wait for ENTER to be pressed (more than 10 seconds).
5. End is outputted to the screen
6. Main code ends. The event loop checks the future, 2 seconds have passed, and it prints 2

Futures

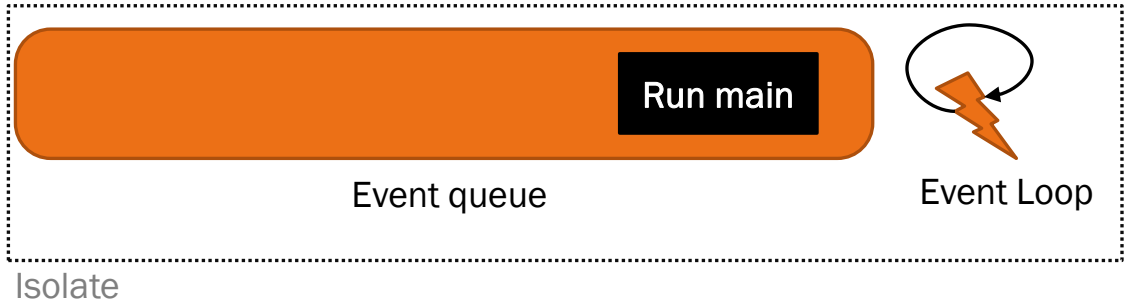
Let’s see how this work:



Isolate

Futures

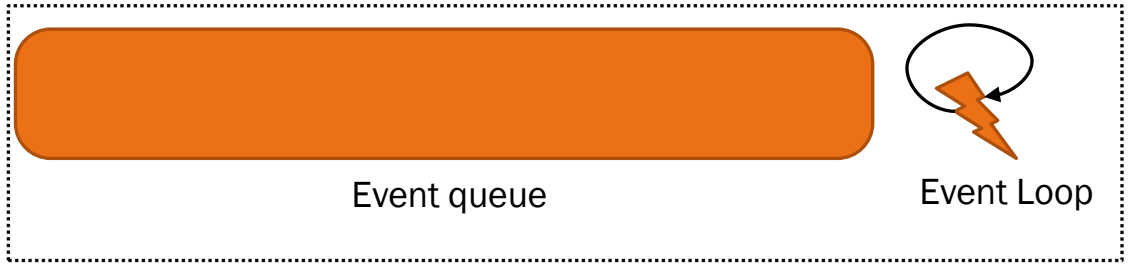
Let’s see how this work:



1. The first event in the list is to run the code from main

Futures

Let’s see how this work:



Isolate

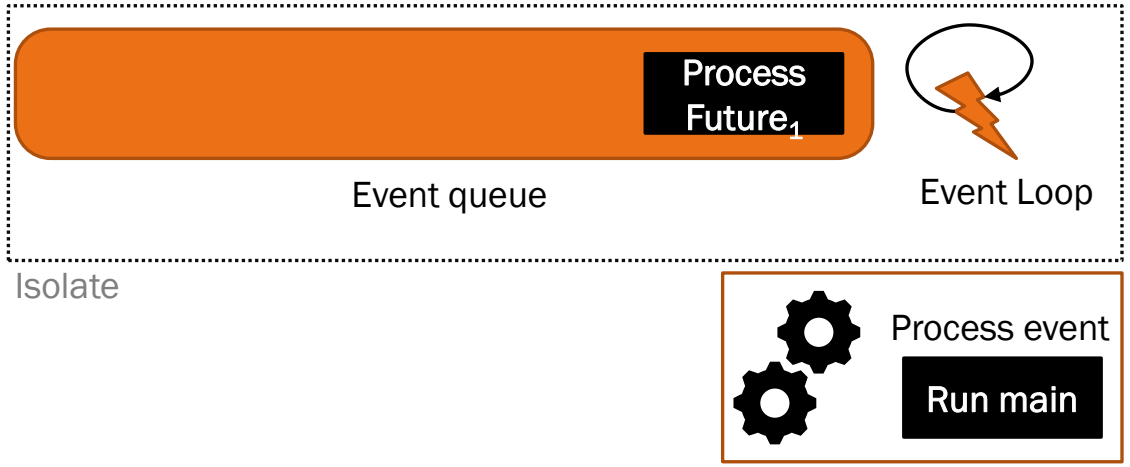
The first event from the queue is process. In this case, it means running the code from function main.

1. The first event in the list is to run the code from main

2. The event loop takes the first available event and processes it.

Futures

Let's see how this work:

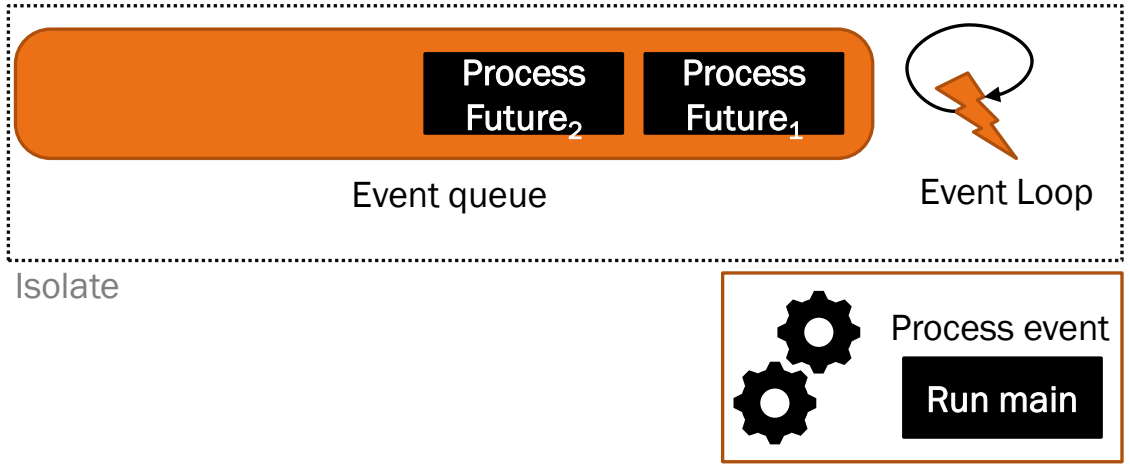


1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created ➔ lets call it Future$_1$

As a result, an event to process that Future (to test if it is completed will be created and pushed in the queue)..

Futures

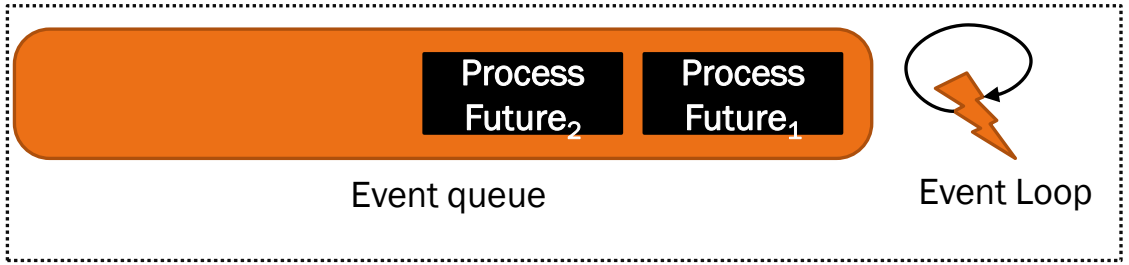
Let's see how this work:



1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created ➔ lets call it Future$_1$
4. After a while, another future object (let's call it Future$_2$) is created by the code from main

Futures

Let's see how this work:

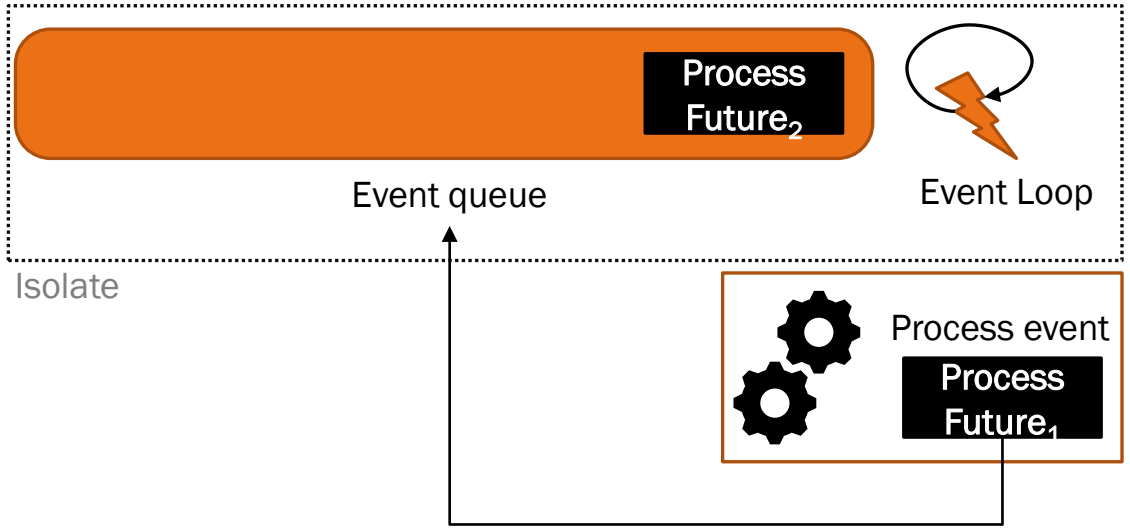


Isolate

1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created ➔ lets call it Future$_1$
4. After a while, another uture object (let's call it Future$_2$) is created by the code from main
5. The code from main ends (meaning that the first event "Run main" was completed).

Futures

Let's see how this work:



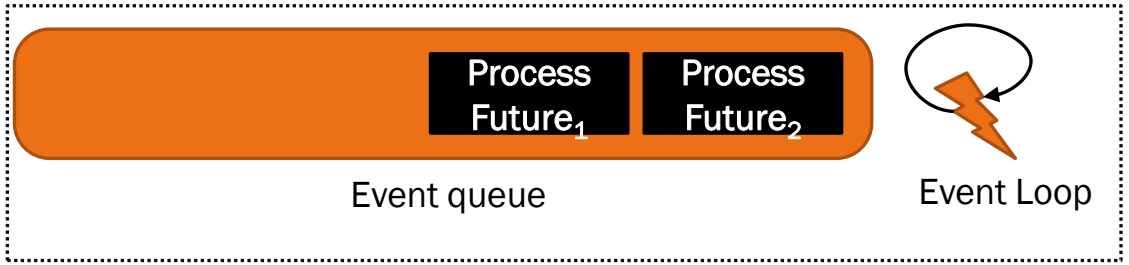
In this case the event will be resend to the event queue to be process at a later time.

1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created ➔ lets call it Future$_1$
4. After a while, another future object (let's call it Future$_2$) is created by the code from main
5. The code from main ends (meaning that the first event "Run main" was completed).
6. The event loop extracts the next event from the queue and start processing it.

Let's assume that the event was not completed (e.g. the URL content was not read, the time that should have passed for a Future<T>,delayed did not pass, etc)

Futures

Let's see how this work:

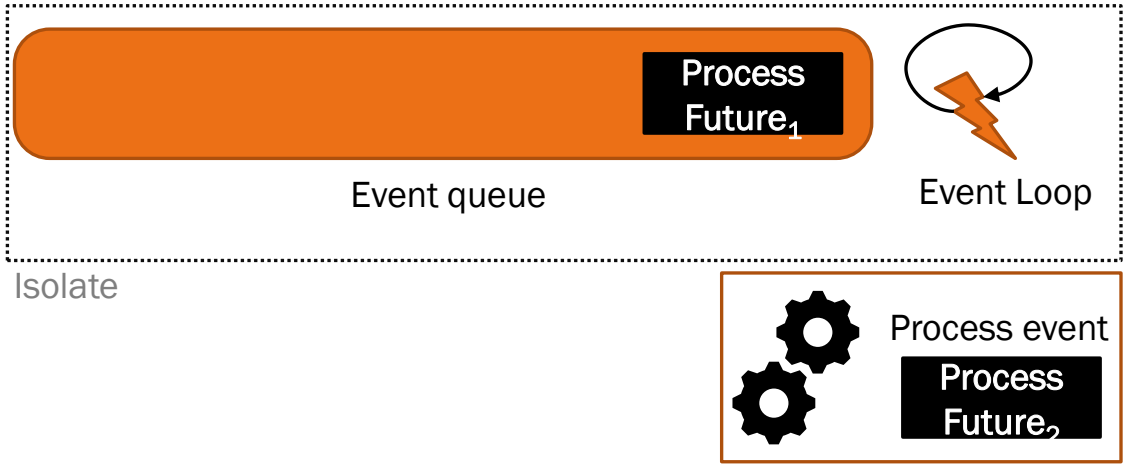


Isolate

1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created ➔ lets call it Future$_1$
4. After a while, another future object (let's call it Future$_2$) is created by the code from main
5. The code from main ends (meaning that the first event "Run main" was completed).
6. The event loop extracts the next event from the queue and start processing it.
7. The event loop now extracts the next event from the queue and starts processing it.

Futures

Let's see how this work:



1. The first event in the list is to run the code from main

7. The event loop now extracts the next event from the queue and starts processing it.

8. The next event refers to Future$_2$. **Event loop will start process this one.**

Assuming that Future$_2$ is completed, the callback that was set through Future<T>.then(...) method will be called.

Futures

Let's see how this work:

Isolate

1. The first event in the list is to run the code from main

7. The event loop now extracts the next event from the queue and starts processing it.

8. The next event refers to Future$_2$. **Event loop will start process this one.**

9. After the previous future has ended, the event is removed from the event ques and the process is restarted and the next event is process.

OBS: This is an approximation behavior (in reality the code behind the event loop si more complex and subject to change from version to version.

Futures

Other methods that can be used for a Future.

`Stream<T> Future<T>.asStream()`	Creates a stream based on a future
`Future<T>.catchError(Function onError, {bool test(Object error)?})`	Set a callback to be used when an error is raised.
`Future<T>.whenComplete(FutureOr<void> action())`	Call when a future is complete (regardless of the future outcome – complete or error).
`Future<T>.timeout(Duration timeLimit, {FutureOr<T> onTimeout()?})`	Sets a timeout for a Future object

Futures

An example using catchError and whenComplete.

```
import "dart:async";

int runCode() {
  print("Run code");
  throw "some exception";
}
int onErrorCallback(error) { print("Error: ${error}"); return 0; }

void main() {
  Future<int>.sync(runCode)
            .catchError(onErrorCallback)
            .then((value) => print("Value:${value}"))
            .whenComplete(() => print("Done"));
}
```

Output:

```
Run code
Error: some exception
Value: 0
Done
```

Futures

A simple example that uses a http package to download our website (info.uaic.ro) and find out the name of our dean.

```
import 'package:http/http.dart' as http;
import 'dart:io';

main() {
  http.get("https://www.info.uaic.ro/conducere/").then((response) {
    var s = r'Decan</strong>:[\w\s\.]*<span class="wikilink">([\w\s]*)</span>';
    var r = RegExp(s);
    var m = r.firstMatch(response.body);
    print(m[1]);
  });
}
```

Futures

To make this code work, the following steps must be performed:

1. Create a **pubspec.yaml** file in the same folder where your dart file is located. Add the following content to pubspec.yaml file

```
name: my_example
version: 1.2.3
dependencies:
    http: ^0.12.1
environment:
    sdk: '>=2.10.0 <3.0.0'
```

2. Run the following command: `**dart pub get**` ➔ where 'dart' referes to the dart executable (e.g <install_folder>\dart-sdk\bin\dart.exe

3. Compile `dart compile exe <file_name>.dart`

Timer

Timer

Dart also has an object (Timer) that works like an event that is being called after a specific period of time.

Constructors:

`Timer (Duration timeLimit, void callback() )`	Creates a timer that will be triggered only once
`Timer.periodic (Duration timePeriod, void callback(Timer timer));`	Creates a timer that will be triggered periodically

Properties:

`int Timer.tick`	Number of timePeriods that have passed
`bool Timer.isActive`	True if the timer is active, false otherwise

The un-named constructor is similar to a Future<T>.delayed object.

Timer

A simple example that uses a timer that will be triggered after 2 seconds and will print a message when those 2 seconds have passed.

```
import 'dart:async';

main() {
  print("start");
  Timer(Duration(seconds: 2), ()=>print("Timer was triggered"));
  print("end");
}
```

Output:
start
end
Timer was triggered

Timer

A timer also has a method (.cancel()) that can be used to stop it. This is useful for periodic timers.

```
import 'dart:async';

void timerCallback(Timer t) {
  if (t.tick>=4)
    t.cancel();
  print("Timer called: Tick = ${t.tick}");
}
main() {
  print("start");
  Timer.periodic(Duration(seconds: 2), timerCallback);
  print("end");
}
```

Output:

```
start
end
Timer called: Tick = 1
Timer called: Tick = 2
Timer called: Tick = 3
Timer called: Tick = 4
```

Synchronization (await/async)

Synchronization

Let's analyze the following code:

```
import "dart:async";

void main() {
  print("start");
  var f = Future<int>.delayed(Duration(seconds: 3),
                              ()=>5)
                       .then((value)=>print("received: ${value}"));
  print("end");
}
```

Output:
start
End
Received: 5

We know that the output will be start , end and received: 5 (as a result on how event loop works) → meaning that main code gets executed first, then the future code is executed.

But what if we want to wait until "f" completes and only then move to the next event code from main ?

Synchronization

First there are some observations to be made here:

1. We need a way to tell the event loop that the current method / function that the event loop is execution can be stop until a future object completes (or to be more precise, we need a way to tell the event loop that current functions works asynchronously).
2. We need a way to specify at what point (location/locations) of the code from we should stop executing the code and cease control to the event loop until the future object we are interested in completes.

Synchronization

Dart has introduced 2 keywords to resolve the previously described problems: await and async

- "async" ➔ should be used for any function that can be interrupted so that its execution waits for one or multiple futures to be completed.
- "await" ➔ can only be used int functions that are defined using "async" keyword and specify the future we want to wait to complete.

```
import "dart:async";

Future<returnType> MyFunction(...) async
{
    // some code
    // a future object (let's call it my_future) is created/obtained
    await my_future;
    // some other code
}
```

This tells the event loop that *MyFunction* should be treated asynchronously (meaning that the execution can be paused until a future completes).

This command pauses the execution of *MyFunction* and wait for *my_future* object to be completed.

Synchronization

Dart has introduced 2 keywords to resolve the previously described problems: await and async

- "async" ➔ should be used for any function that can be interrupted so that its execution waits for one or multiple futures to be completed.
- "await" ➔ can only be used int functions that are defined using "async" keyword and specify the future we want to wait to complete.

```
import "dart:async";

Future<returnType> MyFunction(...) async
{
    // some code
    // a future object (let's call it my_future) is created/obtained
    var res = await my_future; // res will be of type returnType
    // some other code
}
```

The return type should be a Future or void for any async functions.

Synchronization

Let's analyze the previous code with async/await:

```
import "dart:async";

void main() async {
  print("start");
  var f = Future<int>.delayed(Duration(seconds: 5),
                              ()=>5)
                      .then((value)=>print("received: ${value}"));
  await f;
  print("end");
}
```

Output:
Start
received: 5
end

Now the output will be start , received: 5 and end (as a result on how event loop works) → meaning that main code gets executed first, then the future code is executed.

Also, the code will wait for 5 seconds until "f" is complete and only then will run "print("end");" command

Synchronization

await keywork can be use to get the value that a future returns.

```
import "dart:async";

void main() async {
  print("start");
  var f = Future<int>.value(123);
  var result = await f;
  print(result);
  print("end");
}
```

Output:
start
123
end

As a general concept: "var <variable_name> = await <future>" will copy the value returned by a future into a new variable.

Synchronization

This technique can be used to sync multiple future objects that are linked together. Let's assume that we have $Future_1$ that returns a value that will be used by $Future_2$ to compute something.

```
import "dart:async";
void main() {
       Future<int>.value(10)
                  .then((val) => Future<int>.value(val*val)
                                                .then((value) => print(value)));
}
```

or we can write it with async and await like this:

```
import "dart:async";
void main() async {
     int x = await Future<int>.value(10);
     print(await Future<int>.value(x*x));
}
```

Synchronization

You can also use a try...catch or try...catch...finally block for cases where a Future could not complete with a value, but with an error of some sort (e.g. we are waiting to download some content from the internet and the server we are downloading from suddenly stop responding).

```
import "dart:async";

void main() async {
  try {
        int x = await Future<int>.sync(() => throw "My error");
  }
  catch (err) {
        print("Exception: ${err}"); // will print Exception: My error
  }
}
```

Synchronization

Because any function that is created with async keyword returns a Future object (except for void functions), this can be another way to create a Future object.

```
import 'dart:async';

Future<int> GetAFuture(int value) async => value * value;
main() {
  print("start");
  GetAFuture(10).then((value) => print(value));
  print("end");
}
```

Output:
start
end
100

Q & A