CS193P - Lecture 3

iPhone Application Development

Custom Classes	クラスをつくる
Object Lifecycle	オブジェクトの一生
Autorelease	オートリリース
Properties	プロパティ

日本語解説
小嶋 秀樹（宮城大学）
xkozima@myu.ac.jp

Announcements お知らせ

宿題 締切

- Assignments 1A and 1B **due Wednesday 1/13 at 11:59 PM**
  - Enrolled Stanford students can email cs193p@cs.stanford.edu with any questions
  - 提出
  - Submit early! Instructions on the website...
    - **Delete the "build" directory manually, Xcode won't do it**

Announcements お知らせ

宿題 締切

- Assignments 2A and 2B **due Wednesday 1/20 at 11:59 PM**
  - 2A: Continuation of Foundation tool
    - Add custom class
    - Basic memory management
  - 2B: Beginning of first iPhone application
    - Topics to be covered on Thursday, 1/14
    - Assignment contains extensive walkthrough

履修学生　　iTunes U（聴講生）

Enrolled students & iTunes U

レクチャーは iTunes U で公開され始めた

- Lectures have begun showing up on iTunes U
- Lead time is longer than last year （公開まで）去年より時間がかかる
- **Come to class!!** 授業に来てね
  - Lectures may not post in time for assignments
    宿題やるまでにレクチャが（iTune U に）ポストされないかも

Office Hours

- Paul's office hours: Thursday 2-4, Gates B26B
- David's office hours: Mondays 4-6pm: Gates 360

Today's Topics 今日のトピックス

- Questions from Assignment 1A or 1B?
- Creating Custom Classes カスタムクラスをつくる
- Object Lifecycle オブジェクトが生まれてから死ぬまで
- Autorelease 「オートリリース」という概念
- Objective-C Properties Objective-C の「プロパティ」

Custom Classes カスタムクラスをつくる

Design Phase デザイン（設計）フェーズ

- Create a class　クラスを作る（たとえば Person クラス）
  - Person
- Determine the superclass
  - NSObject (in this case)
- What properties should it have?
  - Name, age, whether they can vote
- What actions can it perform?
  - Cast a ballot

親クラスを決める
（ここでは NSObject）

持つべき属性（変数）を考える
（Name, Age, 投票できるか）

可能な行為（メソッド）を考える
（投票する）

クラスを定義する

Defining a class

A public header and a private implementation

公開する「ヘッダ」とプライベートな「実装」

Header File

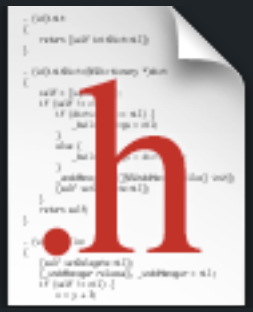
Implementation File

クラスを定義する

Defining a class

A public header and a private implementation

公開する「ヘッダ」とプライベートな「実装」



Header File

Implementation File

ヘッダファイル
（インタフェース）

ヘッダファイルで宣言されたクラスインタフェース

Class interface declared in header file

NS等で始まるオブジェクトを使うためのオマジナイ

```
#import <Foundation/Foundation.h>
```

クラス名 : 親クラス名

```
@interface Person  : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;

- (BOOL)canLegallyVote;
- (void)castBallot;

@end
```

インスタンス変数

メソッドの宣言（本体は .m ファイル）

- (返値型) セレクタ名;
- (返値型) セレクタ名: (引数型) 引数名;

法的に投票できるか

投票する

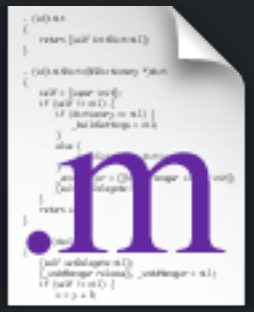
クラスを定義する

Defining a class

A public header and a private implementation

公開する「ヘッダ」とプライベートな「実装」

Header File



Implementation File

インプリメンテーション ファイル
（メソッドの実装）

カスタムクラスを実装する

Implementing custom class

- Implement setter/getter methods セッタ・ゲッタ メソッドを実装
- Implement action methods （その他の）アクション メソッドを実装

クラスの実装

Class Implementation

```
#import "Person.h"
```
ヘッダファイルを読み込む

クラス名

```
@implementation Person
```

(返値型) セレクタ名

```
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}

//... and other methods

@end
```

(引数型) 引数名

メソッド（ゲッタ）の実装
この age はインスタンス変数

メソッド（セッタ）の実装
この age はインスタンス変数

自分で自分のメソッドを呼び出すには…

Calling your own methods

```
#import "Person.h"

@implementation Person

- (BOOL)canLegallyVote {   投票権があるか？

}

- (void)castBallot {       投票する




}

@end
```

自分で自分のメソッドを呼び出すには…

Calling your own methods

```
#import "Person.h"

@implementation Person

- (BOOL)canLegallyVote {
    return ([self age] >= 18); 自分のゲッタを呼び出して年齢をゲット
}

- (void)castBallot {




}

@end
```

自分で自分のメソッドを呼び出すには…

Calling your own methods

```
#import "Person.h"

@implementation Person

- (BOOL)canLegallyVote {
    return ([self age] >= 18);
}

- (void)castBallot {
    if ([self canLegallyVote]) {
        // do voting stuff
    } else {
        NSLog (@“I’m not allowed to vote!”);
    }
}

@end
```

自分のゲッタを呼び出して
自分の年齢をゲット

自分のメソッドを呼び出して
投票権の有無をチェック

投票の中身を実行（省略）

私は投票を許されていない！

親クラスのメソッド

Superclass methods

暗黙の変数　　自分

- As we just saw, objects have an implicit variable named “self”
  - Like “this” in Java and C++

親

- Can also invoke superclass methods using “super”

```
- (void)doSomething {
  // Call superclass implementation first
  [super doSomething];
  // Then do our custom behavior
  int foo = bar;
  // ...
}
```

まずは親の同名メソッドを呼び出す
（親に「お膳立て」をしてもらう）

つぎに自分のカスタム動作を実行する
（自分らしく「飾る・改造する」）

注）self も super も「自分自身＝インスタンス」を意味するが，
メソッドの検索方法が異なる．
１．self は，インスタンスが直接所属するクラスからメソッドを検索．
２．super は，その "super" が出現したクラスの親クラスから
メソッドを検索．

Object Lifecycle

オブジェクトの一生
（生まれてから死ぬまで）

Object Lifecycle オブジェクトの一生

- Creating objects オブジェクトの生成
- Memory management メモリ管理
- Destroying objects オブジェクトの廃棄
  破壊する

Object Creation オブジェクトの生成

- Two step process ２段プロセス
  - allocate memory to store the object
  - initialize object state

１．オブジェクトを格納するメモリ（領域）を確保
２．オブジェクトの状態を初期化

`+ alloc` 必要なメモリを確保（インスタンスを生成）するクラスメソッド
  - Class method that knows how much memory is needed

`- init` 変数の初期化などを行なうインスタンスメソッド
  - Instance method to set initial values, perform other setup

（インスタンス）生成＝メモリ確保＋初期化

Create = Allocate + Initialize

クラス名

```
Person *person = nil;
```

*person インスタンスへのポインタ
（初期値 nil を与えるのには理由があります）

```
person = [[Person alloc] init];
```

メモリ確保 ＋ 初期化

このように書いてもよい：

```
person = [Person alloc];
[person init];
```

クラスメソッド alloc の呼び出し
インスタンスメソッド init の呼び出し

段階ごとに説明すると…

```
[Person alloc];
```

Personクラスのインスタンスを格納するために必要なメモリ領域を確保しそのアドレスを返す.

```
[person init];
```

person で参照されるメモリ領域（つまりインスタンス）を初期化する.
自分自身のアドレスを返す.

あなた独自の -init メソッドを実装する

Implementing your own -init method

```
#import "Person.h"

@implementation Person

- (id)init {
  // allow superclass to initialize its state first
  if (self = [super init]) {
      age = 0;
      name = @“Bob”;

      // do other initialization...
  }

  return self;
}

@end
```

-init は id型（無名オブジェクトへのポインタ）を返す

親クラス（たとえば NSObject）の -init メソッドで初期化し，のちに Person クラス独自の初期化．

-init は self を返すようにする

複数の -init メソッド

Multiple init methods

複数の -init メソッドを定義してもよい

セレクタが異なれば
メソッドも異なる
init
initWithName:
initWithName:age:

- Classes may define multiple init methods

```
- (id)init;
- (id)initWithName:(NSString *)name;
- (id)initWithName:(NSString *)name age:(int)age;
```

- Less specific ones typically call more specific with default values

より一般的なメソッドが
より特殊化された
メソッドを（デフォルト値を
与えて）呼び出すことが多い

一般
```
- (id)init {
    return [self initWithName:@“No Name”];
}
```

特殊
```
- (id)initWithName:(NSString *)name {
    return [self initWithName:name age:0];
}
```

Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];
```

Person クラスの
オブジェクト（インスタンス）を
生成する

内部変数を設定
（名前と年齢）

投票する
何か別のことをする（ビール飲む）

オブジェクトを使いきる

Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

// What do we do with person when we’re done?
```

Person クラスの
オブジェクト（インスタンス）を
生成する

内部変数を設定
（名前と年齢）

投票する
何か別のことをする（ビール飲む）

さて，終わったら，
この person をどうしようか？

Memory Management メモリ管理

	確保 Allocation	破壊 Destruction
C	`malloc`	`free`
Objective-C	`+alloc` 生成	`-dealloc` 廃棄

- Calls must be balanced これらの呼び出し（回数）をバランスさせる
  - Otherwise your program may leak or crash さもなくばメモリ漏れ あるいはクラッシュ
- However, you'll **never** call -dealloc directly
  - One exception, we'll see in a bit...

-dealloc を直接呼び出すことは絶対ない
ひとつだけ例外（後述）があるけれど…

Reference Counting 参照数をカウントする

すべてのオブジェクトは retain count をもつ

- Every object has a **retain count**
  - Defined on NSObject
  - As long as retain count is > 0, object is alive and valid

0 より大きければオブジェクトは生きている

生成
- `+alloc` and `-copy` create objects with retain count == 1 生成時は１

保持
- `-retain` increments retain count -retain により１だけ増える

解放
- `-release` decrements retain count -release により１だけ減る

- When retain count reaches 0, **object is destroyed**

廃棄
- `-dealloc` method invoked automatically
  - One-way street, once you're in -dealloc there's no turning back

retain count が０に達するとオブジェクトは廃棄される

-dealloc が自動的に呼ばれる

一度 -dealloc したら元に戻すことはできない（一方通行）

Balanced Calls alloc/release のバランス

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

// When we’re done with person, release it
[person release];   // person will be destroyed here
```

person の retain count は１

person の retain count が１だけ減り０となる．
すぐに -dealloc が呼び出され，
person オブジェクトは廃棄される．

Reference counting in action 参照数カウントの動き

```
Person *person = [[Person alloc] init];
```

Retain count begins at 1 with +alloc

```
[person retain];
```

増える

Retain count increases to 2 with -retain

```
[person release];
```

減る

Retain count decreases to 1 with -release

```
[person release];
```

減る

-dealloc が自動的に呼び出される

Retain count decreases to 0, -dealloc automatically called

廃棄されたオブジェクトにメッセージを送ったら…

Messaging deallocated objects

```
Person *person = [[Person alloc] init];      person 生成
// ...
[person release]; // Object is deallocated   person 廃棄
```

廃棄されたオブジェクトにメッセージを送ったら…

Messaging deallocated objects

```
Person *person = [[Person alloc] init];      person 生成
// ...
[person release]; // Object is deallocated   person 廃棄


[person doSomething]; // Crash!
```

廃棄された person にメッセージを送ると・・・クラッシュ！

廃棄されたオブジェクトにメッセージを送ったら…

Messaging deallocated objects

```
Person *person = [[Person alloc] init];      person 生成
// ...
[person release]; // Object is deallocated   person 廃棄
```

廃棄されたオブジェクトにメッセージを送ったら…

Messaging deallocated objects

```
Person *person = [[Person alloc] init];      person 生成
// ...
[person release]; // Object is deallocated   person 廃棄
person = nil;     廃棄したら nil を入れておく
```

廃棄されたオブジェクトにメッセージを送ったら…

Messaging deallocated objects

```
Person *person = [[Person alloc] init];
// ...
[person release]; // Object is deallocated
person = nil;
[person doSomething]; // No effect
```

person 生成

person 廃棄

廃棄したら nil を入れておく

何も起こらない
（nil 送ったメッセージはすべて無視される）

-dealloc（廃棄）メソッドの実装

Implementing a -dealloc method

```
#import "Person.h"

@implementation Person

- (void)dealloc {
    // Do any cleanup that's necessary
    // ...

    // when we're done, call super to clean us up
    [super dealloc];
}

@end
```

Person クラス独自の「お掃除」

親クラスの -dealloc を呼び出す
（最終的に NSObject の -dealloc が
メモリを解放してくれる）

recapitulation（要約・反復）

Object Lifecycle Recap

オブジェクトの一生のまとめ

- Objects begin with a retain count of 1
- Increase and decrease with -retain and -release
- When retain count reaches 0, object deallocated automatically
- You **never** call dealloc explicitly in your code
  - Exception is calling -[super dealloc]
  - You only deal with alloc, copy, retain, release

生成時は１

-retain で＋１

-release でー１

０になったら
自動的に -dealloc

-dealloc を直接呼び出すことは絶対ない

- 唯一の例外は [super dealloc]
- 直接扱うのは +alloc, -copy, -retain, -release のみ

Object Ownership オブジェクトの所有権

```
#import <Foundation/Foundation.h>

@interface Person  : NSObject
{
    // instance variables
    NSString *name;  // Person class “owns” the name
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;

- (BOOL)canLegallyVote;
- (void)castBallot;

@end
```

Person クラスは name オブジェクトを所有

name のゲッタ

name のセッタ

Object Ownership オブジェクトの所有権

```
#import "Person.h"

@implementation Person









@end
```

Object Ownership オブジェクトの所有権

```
#import "Person.h"

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {



}

@end
```

ゲッタの実装（これは簡単）

セッタの実装

Object Ownership

オブジェクトの所有権インスタンス変数の解放

```
#import "Person.h"

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {
    if (name != newName) {
        [name release];
        name = [newName retain];
        // name's retain count has been bumped up by 1
    }
}

@end
```

ゲッタの実装（これは簡単）

セッタの実装

name の retain count が１だけ増える

（実際の retain count は２以上かもしれない＝中古品）

Object Ownership オブジェクトの所有権

```
#import "Person.h"

@implementation Person

- (NSString *)name {  ゲッタの実装（これは簡単）
    return name;
}
- (void)setName:(NSString *)newName {  セッタの実装




}

@end
```

Object Ownership オブジェクトの所有権

```
#import "Person.h"

@implementation Person

- (NSString *)name {  ゲッタの実装（これは簡単）
    return name;
}
- (void)setName:(NSString *)newName {  セッタの実装
    if (name != newName) {
        [name release];
        name = [newName copy];   新しくコピーオブジェクトを生成
        // name has retain count of 1, we own it
    }        コピーは新品なので retain count は１
}

@end
```

Releasing Instance Variables インスタンス変数の解放

```
#import "Person.h"

@implementation Person

- (void)dealloc {
    // Do any cleanup that's necessary
    [name release];

    // when we're done, call super to clean us up
    [super dealloc];
}

@end
```

Person クラス独自の「廃棄」

name オブジェクトの解放（age は解放不要）

親クラス（NSObject）に
自分を廃棄してもらう

Autorelease オートリリース（自動解放）

新しく生成したオブジェクトを返すと…

Returning a newly created object

```
- (NSString *)fullName {
    NSString *result;

                                生成        初期化
    result = [[NSString alloc] initWithFormat:@"%@ %@",
                                        firstName, lastName];


    return result;
}
```

ここで生成したオブジェクト（文字列 result）を
呼び出し元に返すと…

Wrong: result is **leaked**!
よくない（エラーを招きやすい）

呼び出し元は，-fullName メソッドで得た文字列を，
後に自分で -release しなければならないが，
そのことを知る由もない．（ヘッダに書いてない）

新しく生成したオブジェクトを返すと…

Returning a newly created object

```
- (NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@"%@ %@",
                                        firstName, lastName];

    [result release];
    return result;
}
```

生成　初期化

ならば，ここで解放してから
呼び出し元に返すようにすると…

Wrong: result is **released too early**!
Method returns bogus value

間違い（エラー）：result の解放が早すぎる

-release により result は廃棄されてから，
意味のない値が呼び出し元に返されてしまう

新しく生成したオブジェクトを返すと…

Returning a newly created object

```
- (NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@"%@ %@",
                                        firstName, lastName];

    [result autorelease];

    return result;
}
```

生成　　初期化

result に「自動解放」を設定してから
result を返す

Just right: result is released, but not right away
Caller gets valid object and could retain if needed

これが正解：result は（今すぐでなく）やがて解放される．
呼び出し元は，生きているオブジェクトをゲットでき，
必要に応じて -retain することができる．

Autoreleasing Objects オブジェクトの自動解放

- Calling -autorelease flags an object to be sent release at some point in the future
- Let's you fulfill your retain/release obligations while allowing an object some additional time to live
- Makes it much more **convenient** to manage memory
- Very useful in methods which **return a newly created object**

- **-autorelease により，後に -release が送られるように，印がつけられる．**
- **retain/release のバランスを取るながら，それとは別に，**
  オブジェクトに少しだけ余命を与えてくれる．
- **メモリ管理を楽（便利）にしてくれる．**
- **新しいオブジェクトを生成して返すようなメソッドに使える．**

メソッド名とオートリリース

Method Names & Autorelease

alloc/copy/new を名前に含んだメソッドは，autorelease ではない．

- Methods whose names includes **alloc, copy, or new** return a retained object that the **caller needs to release**

```
NSMutableString *string = [[NSMutableString alloc] init];
// We are responsible for calling -release or -autorelease
[string autorelease];
```

明示的に autorelease を設定するか
あるいは不要になった時点で -release する．

- All other methods return autoreleased objects

それ以外のメソッドはすべて autorelease オブジェクトを返す．

```
NSMutableString *string = [NSMutableString string];
// The method name doesn't indicate that we need to release it
// So don't- we're cool!
```

したがって -release はしなくてよい．

- This is a convention- **follow it in methods you define**!

あなたがつくるカスタムクラス（サブクラス）も
この名前のつけかたを踏襲すべき．

-autorelease はどのように動くのか？

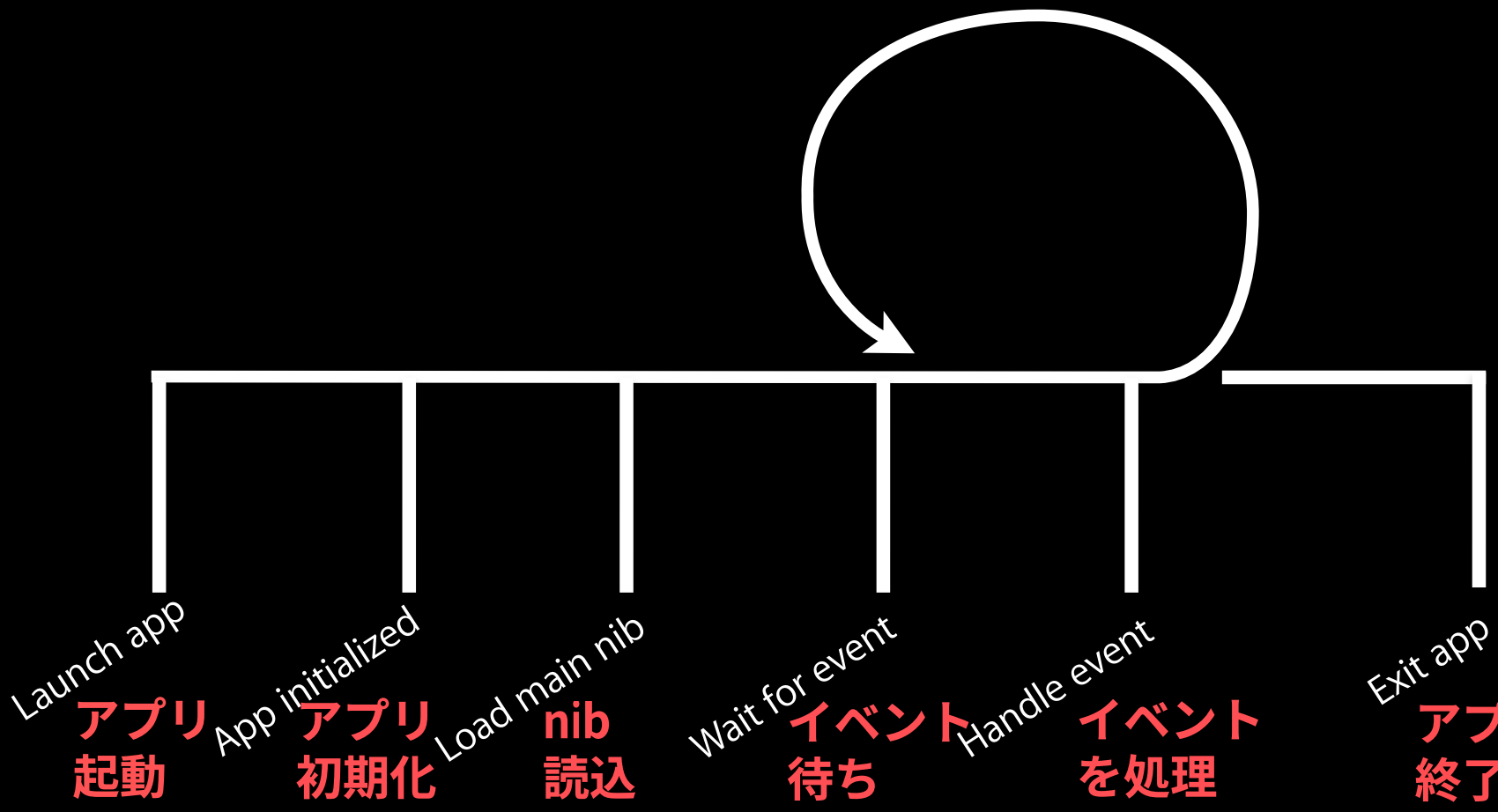
How does -autorelease work?

- Object is added to **current autorelease pool**
- Autorelease pools track objects scheduled to be released
  - When the pool itself is released, it sends -release to all its objects
- UIKit automatically wraps a pool around every event dispatch

- autorelease オブジェクトは現在の自動排水プールに加えられる．
- 自動解放プールは，解放が予定されたオブジェクトを追跡する．
  プールが排水されるとき，
  登録された全てのオブジェクトに -release が送られる．
- UIKit は，イベント処理ディスパッチごとに，プールを生成/排水する．
  （数ミリ秒ごとのイベント処理ループの１回）

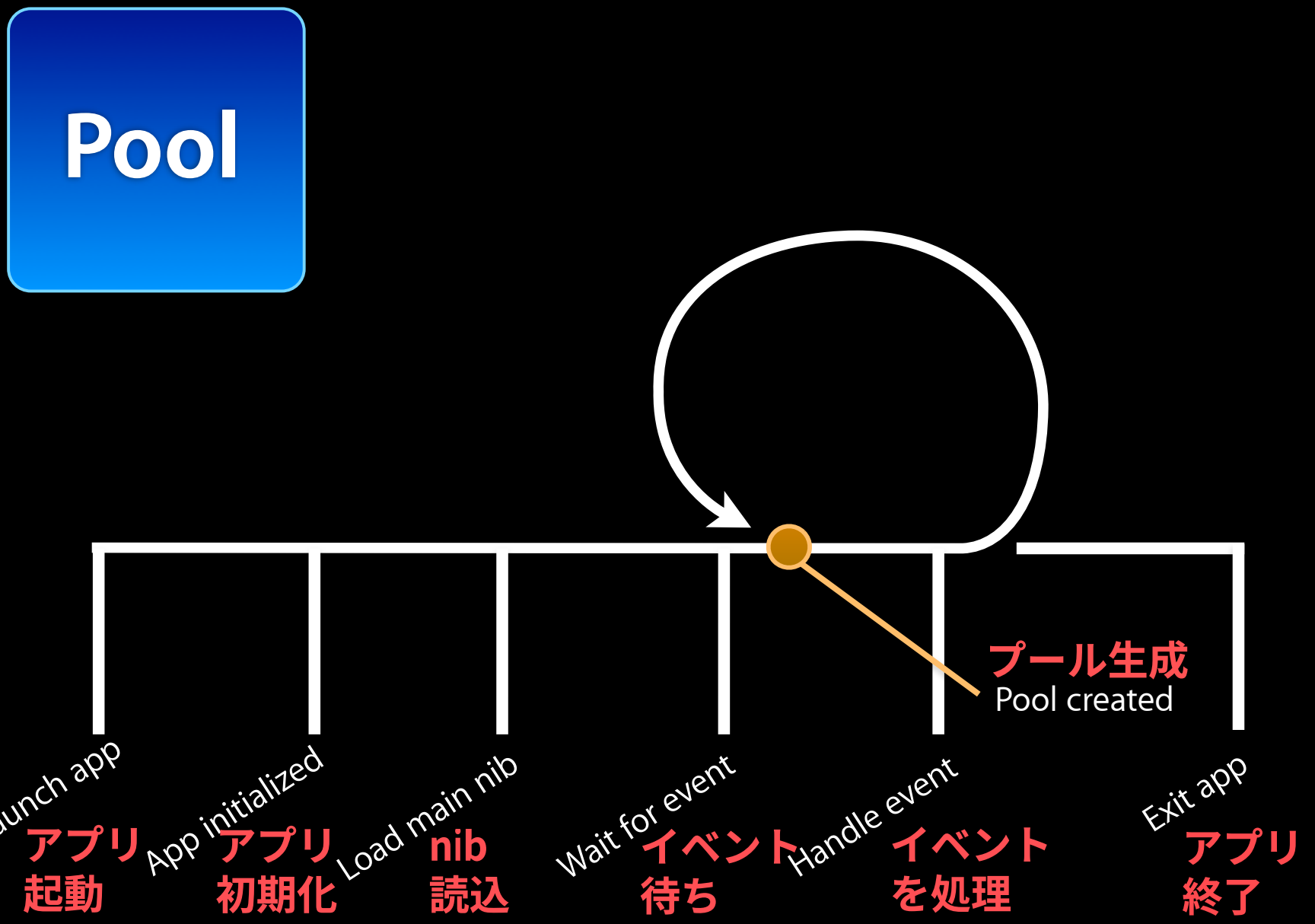
自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)



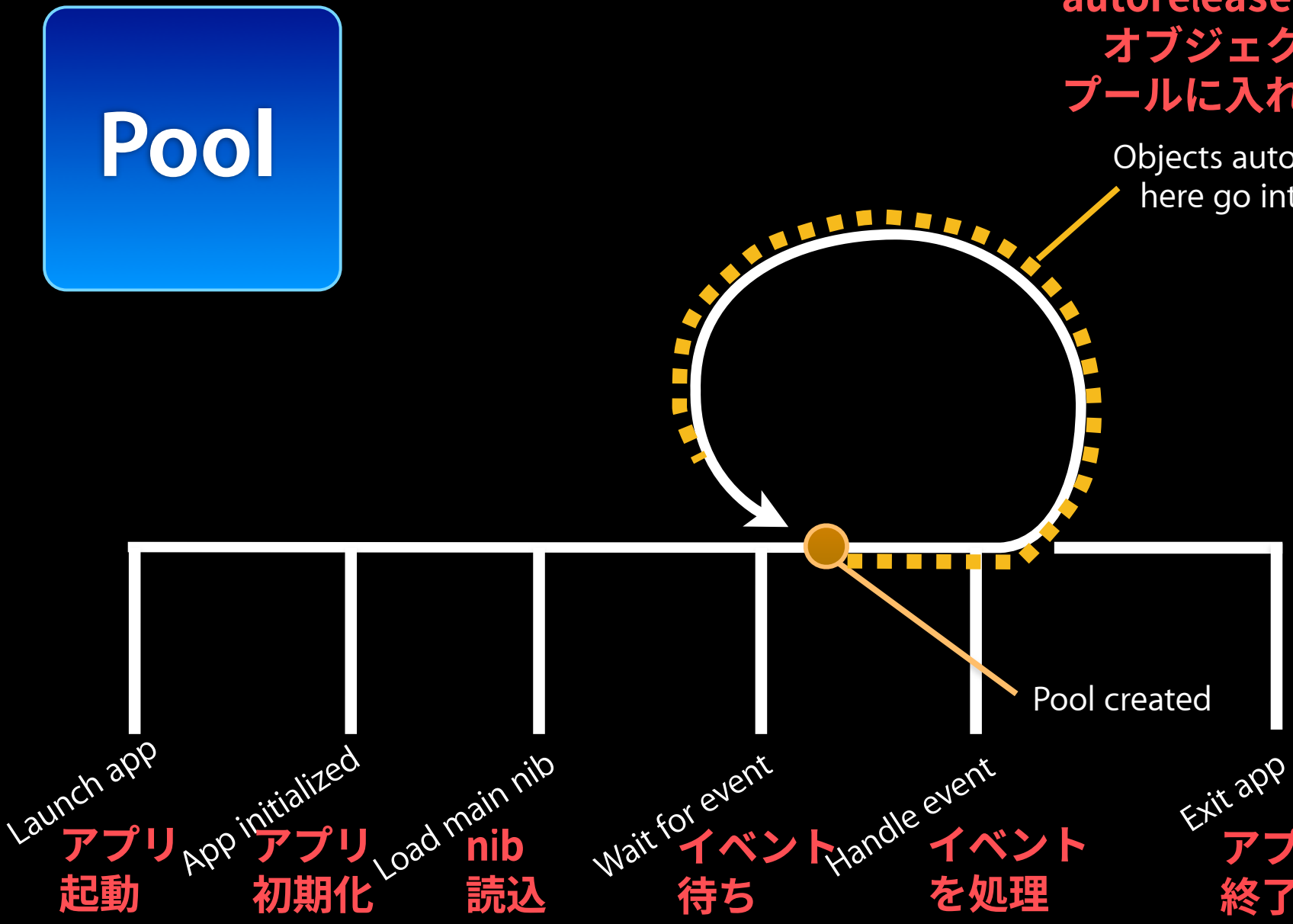
自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)



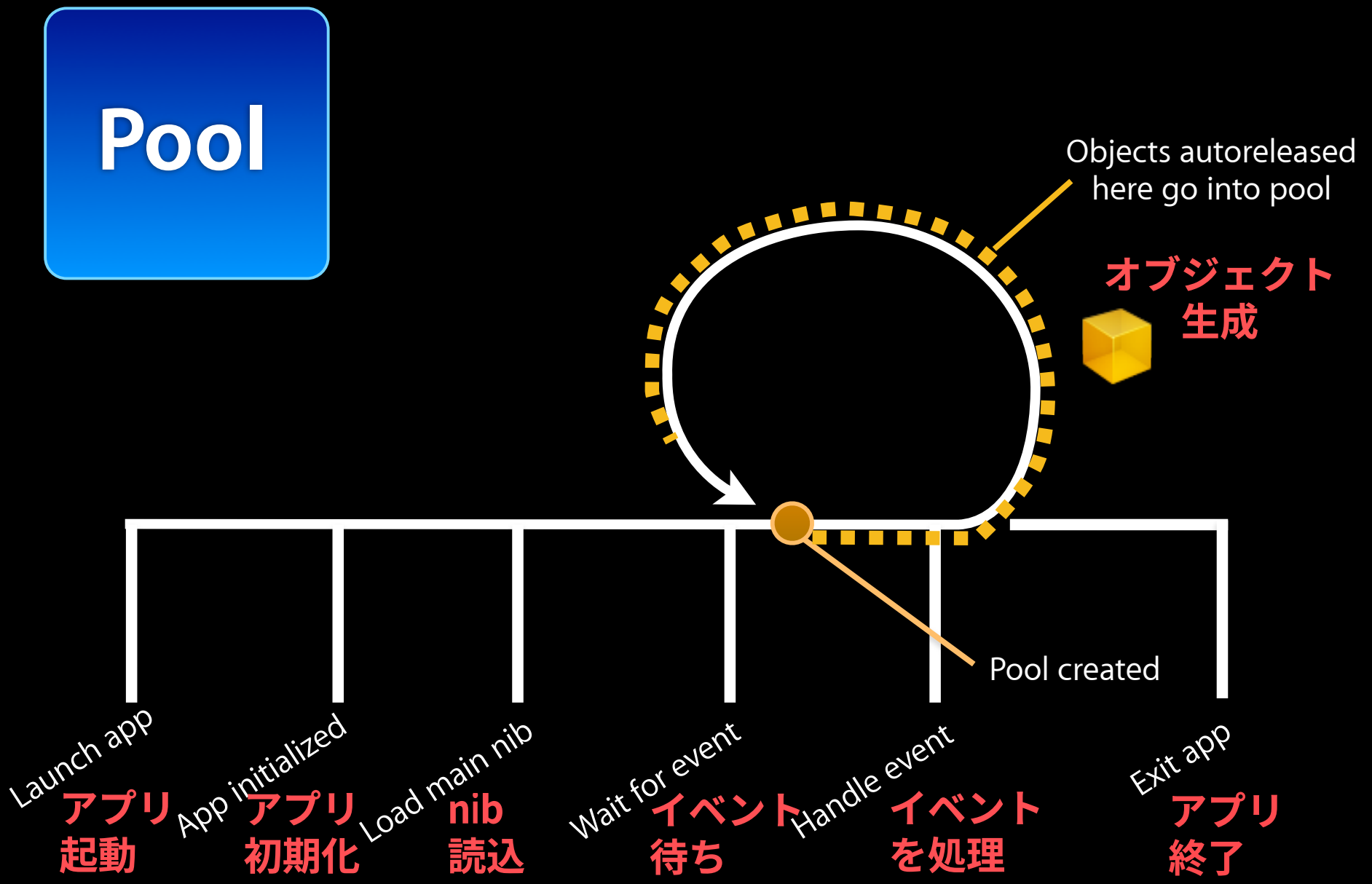
自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)



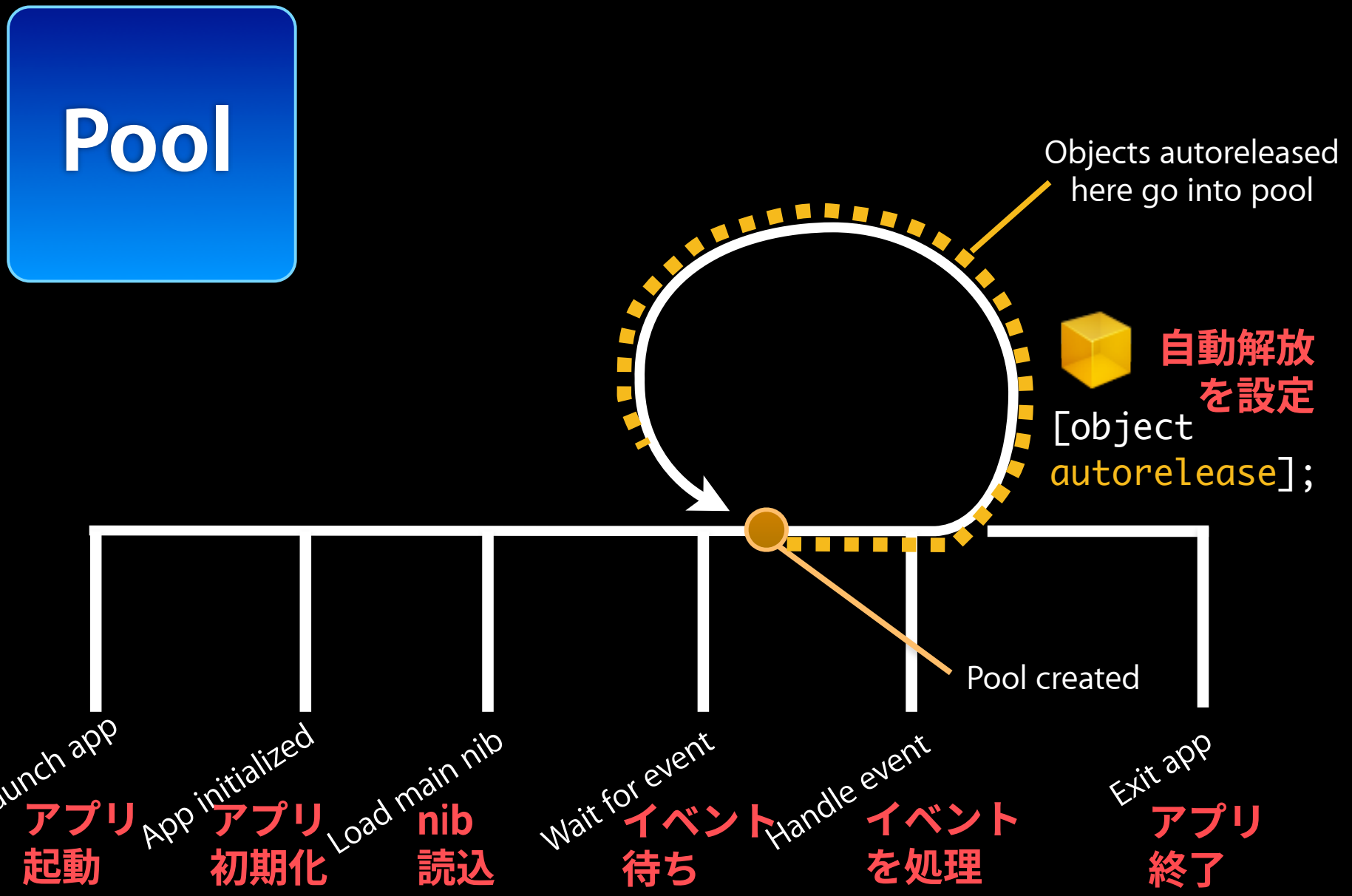
自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)



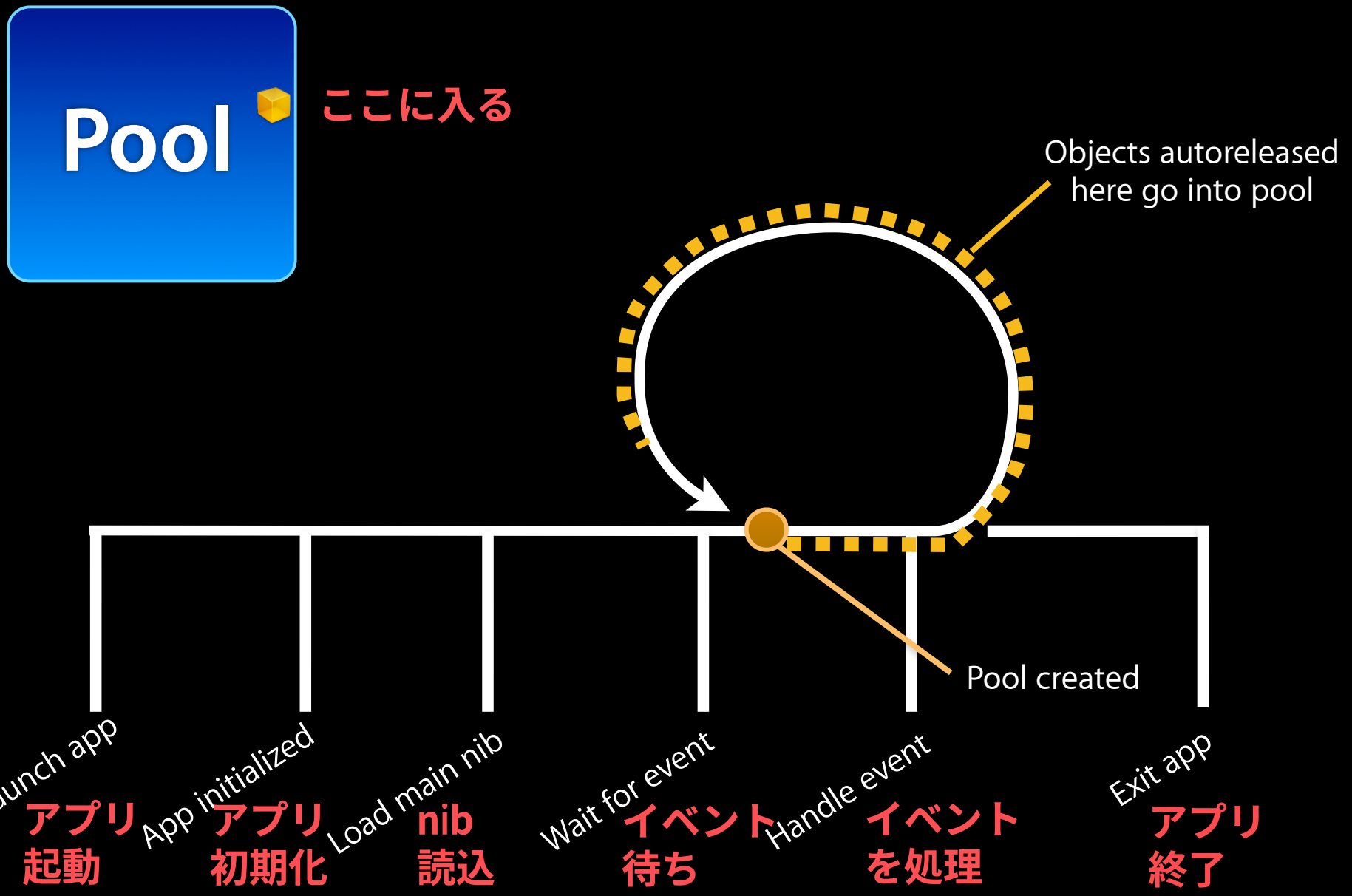
自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)



自動排水プール（図で説明しよう）

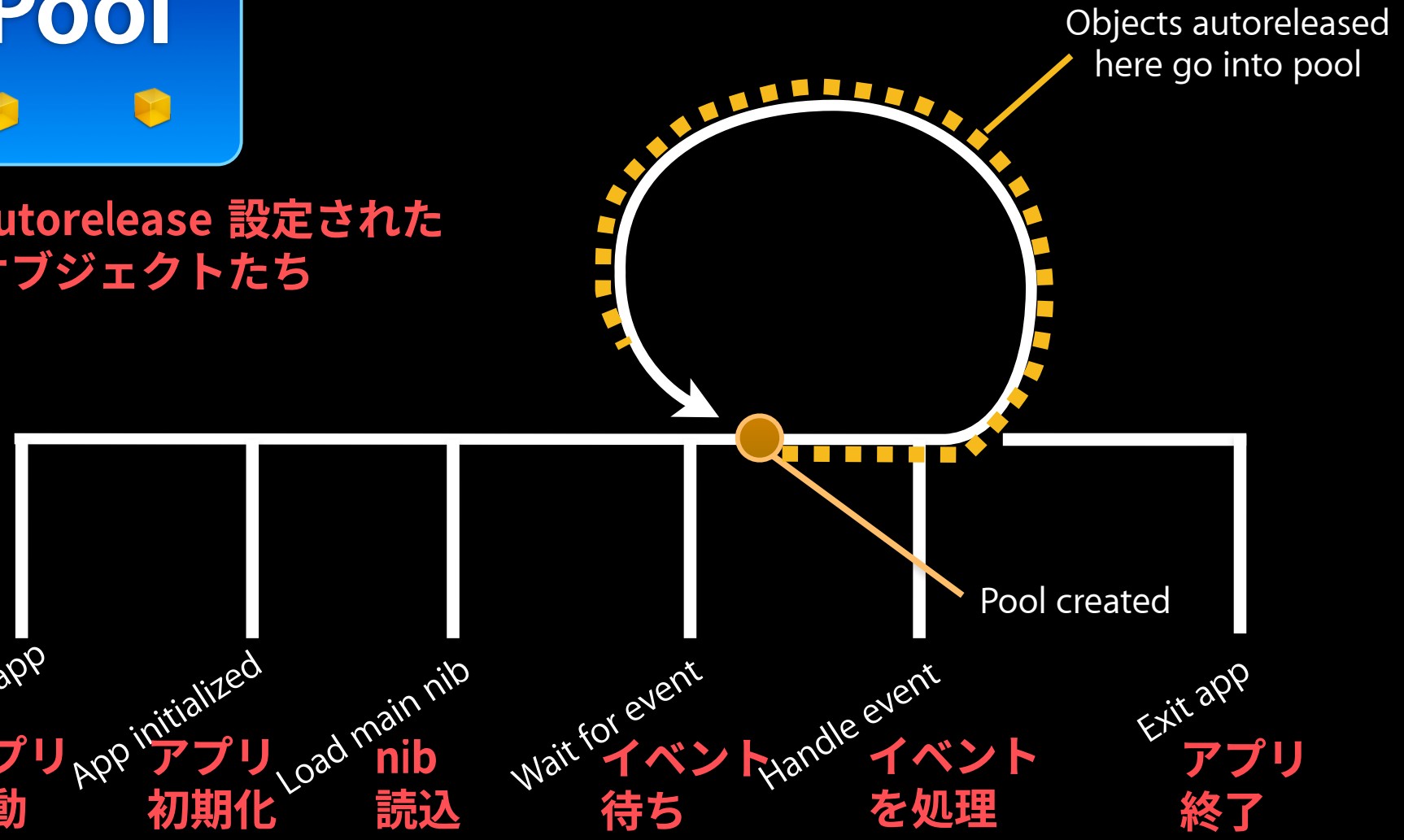
Autorelease Pools (in pictures)



自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)

autorelease 設定された
オブジェクトたち

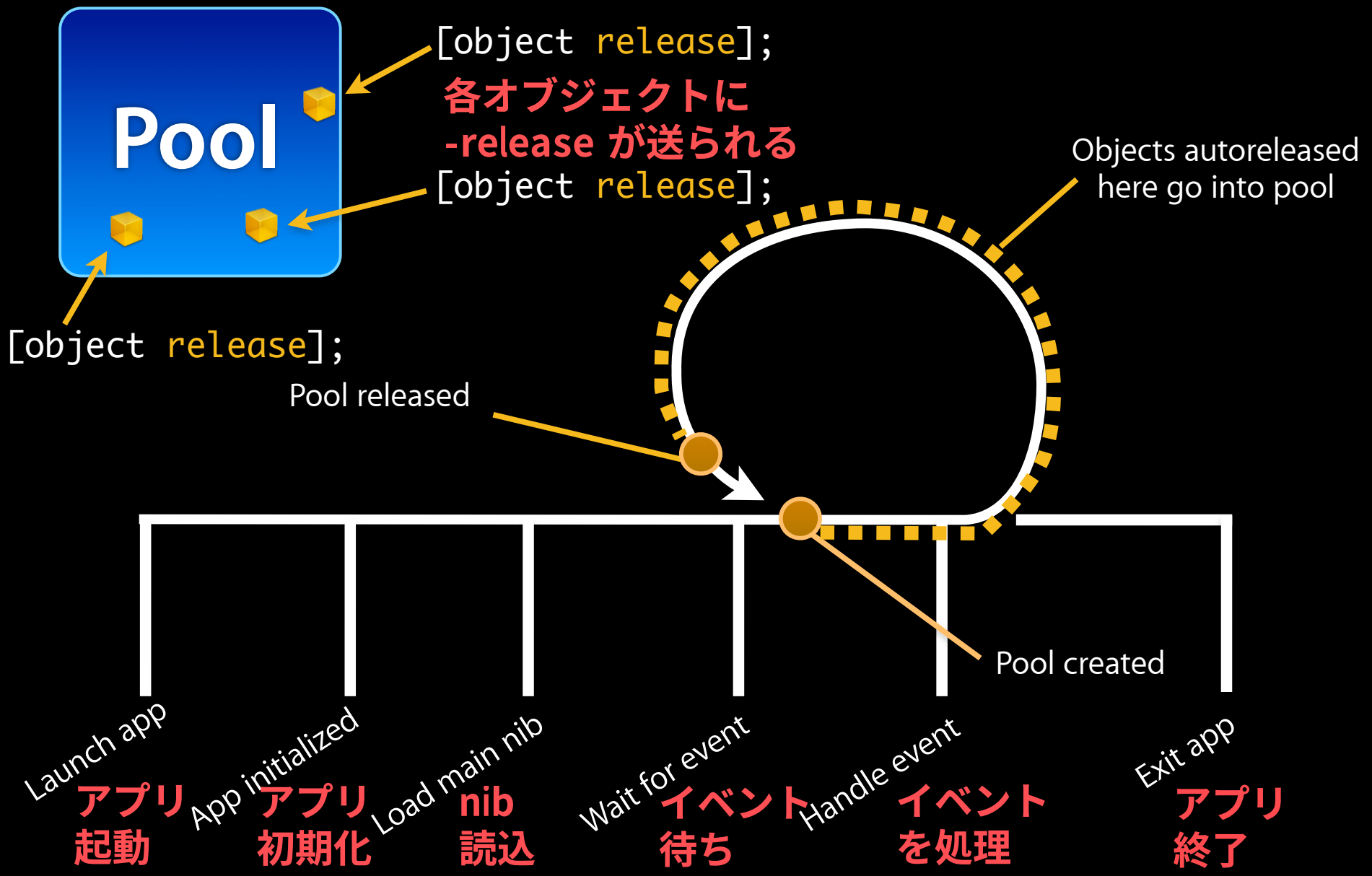


自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)

自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)

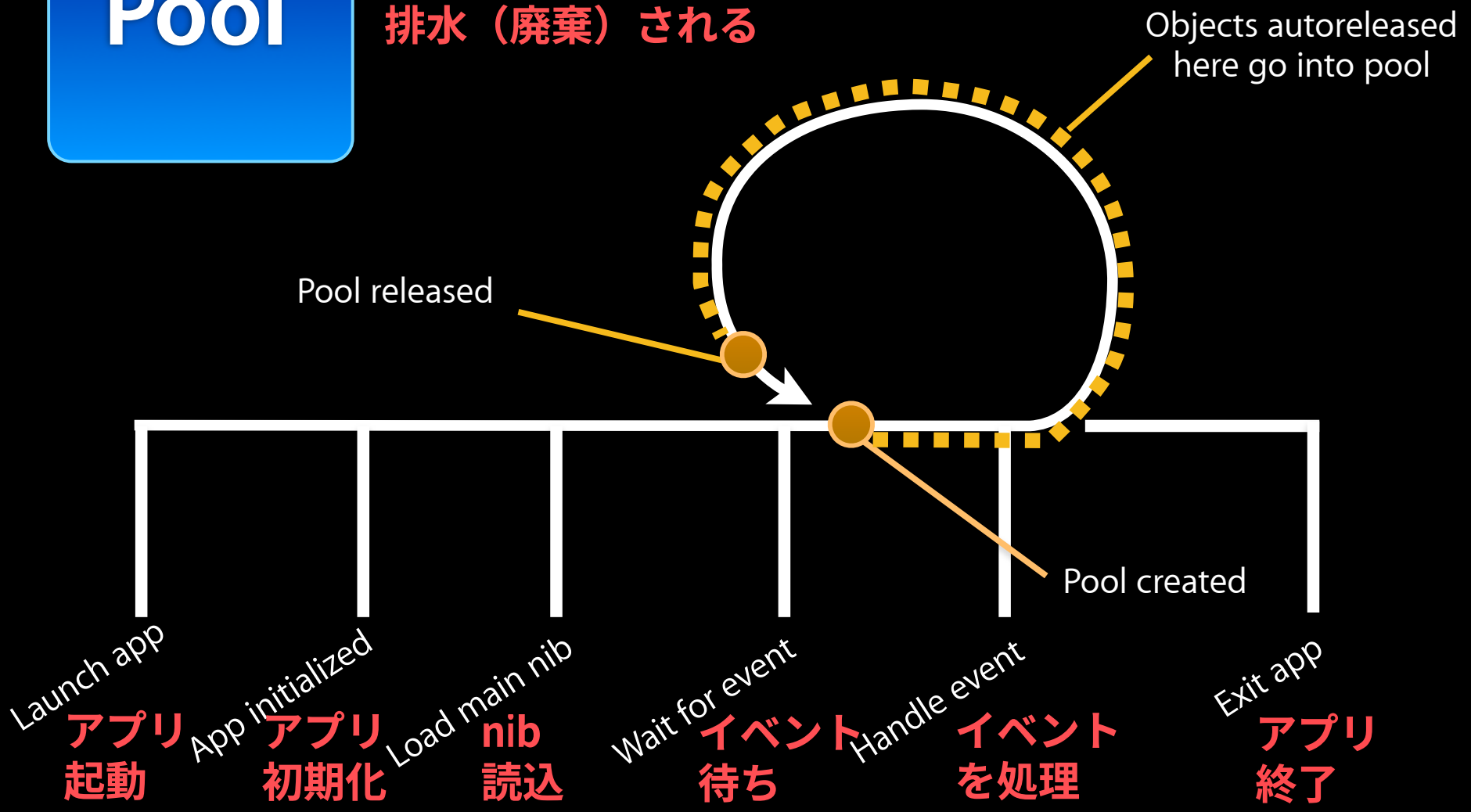


自動排水プール（図で説明しよう）

Autorelease Pools (in pictures)

Pool

明示的に retain された
オブジェクト以外は
排水（廃棄）される



自動解放オブジェクトをキープしておくには…

Hanging Onto an Autoreleased Object

多くのメソッドが自動解放オブジェクトを返す

- Many methods return autoreleased objects
  - Remember the naming conventions...
  - They're hanging out in the pool and will get released later

命名法

しばらくプールにいるが，
やがて解放（廃棄）される

- If you need to hold onto those objects you need to retain them
  - Bumps up the retain count *before* the release happens

自動解放オブジェクトをキープしたければ
-release される前に
-retain してカウンタ値を増す

```
name = [NSMutableString string];

// We want to name to remain valid!
[name retain];

// ...
// Eventually, we'll release it (maybe in our -dealloc?)
[name release];
```

-retain することで name は流されない

最後に（不要になったら） -release する
（たぶん，このクラスの -dealloc で？）

補足：ガベージコレクション（GC）

Side Note: Garbage Collection

- **Autorelease is not garbage collection**
- Objective-C on iPhone OS does not have garbage collection

・オートリリースは GC ではない
・iOS の ObjC は GC をサポートしていない．
（Mac OS X の ObjC-2.0 は GC をサポート）

Objective-C Properties

Objective-C の「プロパティ」という概念

Properties プロパティ

- Provide access to object attributes
- Shortcut to implementing getter/setter methods
- Also allow you to specify:
  - read-only versus read-write access
  - memory management policy

- オブジェクトの属性（変数）へのアクセス手段を提供する
- ゲッタ/セッタ メソッドを実装するための近道
- また，以下の設定が可能：
  - 読み出しのみ / 読み書きOK
  - メモリ管理のポリシー（方策）

Defining Properties プロパティを定義する

まずはヘッダファイルから

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;


- (void)castBallot;
@end
```

name のゲッタ
name のセッタ
age のゲッタ
age のゲッタ
canLegallyVote

Defining Properties プロパティを定義する

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}
```

ここの部分に魔法をかけると…

```
// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
```

name のゲッタ
name のセッタ
age のゲッタ
age のゲッタ
canLegallyVote

```
- (void)castBallot;
@end
```

Defining Properties プロパティを定義する

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}
```

するとインスタンス変数名だけが残り

```
// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;

- (void)castBallot;
@end
```

name の…

age の…

canLegallyVote の…

Defining Properties プロパティを定義する

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}
```

@property の１行に化ける

```
// property declarations
@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;
```

age
name (copy)
can… (readonly)

```
- (void)castBallot;
@end
```

Defining Properties プロパティを定義する

こんなヘッダファイルになる

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}


// property declarations
@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;



- (void)castBallot;
@end
```

Synthesizing Properties プロパティの生成

```
@implementation Person

- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
        [name release];
        name = [value copy];
    }
}

- (void)canLegallyVote { ...
```

BOOL

従来の方法で書いた
Person の実装ファイル

age ゲッタ

age セッタ

name ゲッタ

name セッタ

Synthesizing Properties プロパティの生成

```
@implementation Person
```

```
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
        [name release];
        name = [value copy];
    }
}
```

ここに魔法をかけると…

```
- (void)canLegallyVote { ...
```

BOOL

Synthesizing Properties プロパティの生成

```
@implementation Person
```

```
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
        [name release];
        name = [value copy];
    }
}
```

```
- (void)canLegallyVote { ...
```

BOOL

ここに魔法をかけると…

インスタンス変数名
だけが残り…

Synthesizing Properties プロパティの生成

```
@implementation Person

@synthesize age;
@synthesize name;
- (BOOL)canLegallyVote {
    return (age > 17);
}

@end
```

ここに魔法をかけると…

インスタンス変数名
だけが残り…
これだけの記述に
圧縮される

@synthesize により
read-write 設定のインスタンス変数には
変数名をもつゲッタ
set変数名 をもつセッタ が自動生成される.
read-only 設定のインスタンス変数には
変数名をもつゲッタ が自動生成される.

-canLegallyVote のように @synthesize を使わずにベタ書きしてもよい

Property Attributes プロパティの属性

- Read-only vs. versus read-write

read-write がデフォルト

```
@property int age;  // read-write by default
@property (readonly) BOOL canLegallyVote;
```

readonly なら明示する

- Memory management policies (only for object properties)

```
@property (assign) NSString *name; // pointer assignment
@property (retain) NSString *name; // retain called
@property (copy) NSString *name;   // copy called
```

NSString のようなオブジェクトをとるインスタンス変数には，
メモリ管理のポリシーを明示する．
assign : 引数オブジェクト（ポインタ）をそのまま代入
retain : 引数オブジェクトを -retain して代入
copy : 引数オブジェクトを -copy して代入
（「if (value != name) ...」の処理も勝手やってくれる）

プロパティ名 vs. インスタンス変数

Property Names vs. Instance Variables

- Property name can be different than instance variable

```
@interface Person : NSObject {
    int numberOfYearsOld;
}

@property int age;

@end

@implementation Person

@synthesize age = numberOfYearsOld;

@end
```

インスタンス変数

プロパティ名（メソッド名）

このように書けば
変数名とは異なる
プロパティ（メソッド）名を
使うことができる.

Properties プロパティ

自動生成された　　実装された（ベタ書きの）

- Mix and match synthesized and implemented properties

```
@implementation Person

@synthesize age;    -age と -setAge が自動生成
@synthesize name;

- (void)setAge:(int)value {  ここで -setAge を上書き定義すれば
    age = value;             こちらが有効になる（-age は不変）

    // now do something with the new age value...
}

@end
```

- Setter method explicitly implemented　セッタは明示的に実装
- Getter method still synthesized　　ゲッタは自動生成

Properties In Practice プロパティの実際

- Newer APIs use @property 新しい API は @property を使っている
- Older APIs use getter/setter methods 古いのはセッタ/ゲッタを実装
- Properties used heavily throughout UIKit APIs
  - Not so much with Foundation APIs
- You can use either approach
  - Properties mean writing less code, but "magic" can sometimes be non-obvious

UIKit では多用されるが
Foundation ではそうでもない

どちらを使ってもよい：
プロパティはコードが短くなるが，
このような「マジック」は時として「分かりづらい」

Dot Syntax and self ドット記法と self について

- When used in custom methods, be careful with dot syntax for properties defined in your class
- References to properties and ivars behave very differently

プロパティへの参照と
ivar への参照は「異なる」

```
@interface Person : NSObject
{
    NSString *name;
}
@property (copy) NSString *name;
@end

@implementation Person
- (void)doSomething {
    name = @“Fred”;
    self.name = @“Fred”;
}
```

-name / -setName の宣言

上の行は直接 ivar にアクセス

```
// accesses ivar directly!
// calls accessor method
```

下の行は [self name:@"Fred"]

ドット記法に関する「よくある落とし穴」

Common Pitfall with Dot Syntax

What will happen when this code executes?

これを実行するとどうなるか？

```
@implementation Person
- (void)setAge:(int)newAge {
    self.age = newAge;
}
@end
```

This is equivalent to:

```
@implementation Person
- (void)setAge:(int)newAge {
    [self setAge:newAge]; // Infinite loop!
}                         // 無限ループ
@end
```

ここは素直に
age = newAge;
と書けばよい

Further Reading さらに勉強したい人はこれらを読んでね

- Objective-C 2.0 Programming Language
  - “Defining a Class”
  - “Declared Properties”
- Memory Management Programming Guide for Cocoa

いずれも developer.apple.com からダウンロードできると思います

Questions?