Memory Management

COSC346

Life cycle of an object

- Create a reference pointer
- Allocate memory for the object
- Initialise internal data
- Do stuff
- Destroy the object
- Release memory

Constructors and destructors

- **Constructor** is a method that creates an object instance
  - Allocates memory for instance data
  - May initialise instance variables
  - Usually can be overloaded to accept parameters for initialisation
  - Associated with the **new** operator
- **Destructor** is a method that deletes an object instance
  - Releases any memory allocated in the constructors (or during lifetime of the object)
  - Associated with the **delete** operator

What is memory management

- Memory management is recycling for your program
  - Each object takes memory in the computer
  - When you finish with an object, you can remove it to re-use the space
  - If objects are never removed, memory fills up and the program can crash

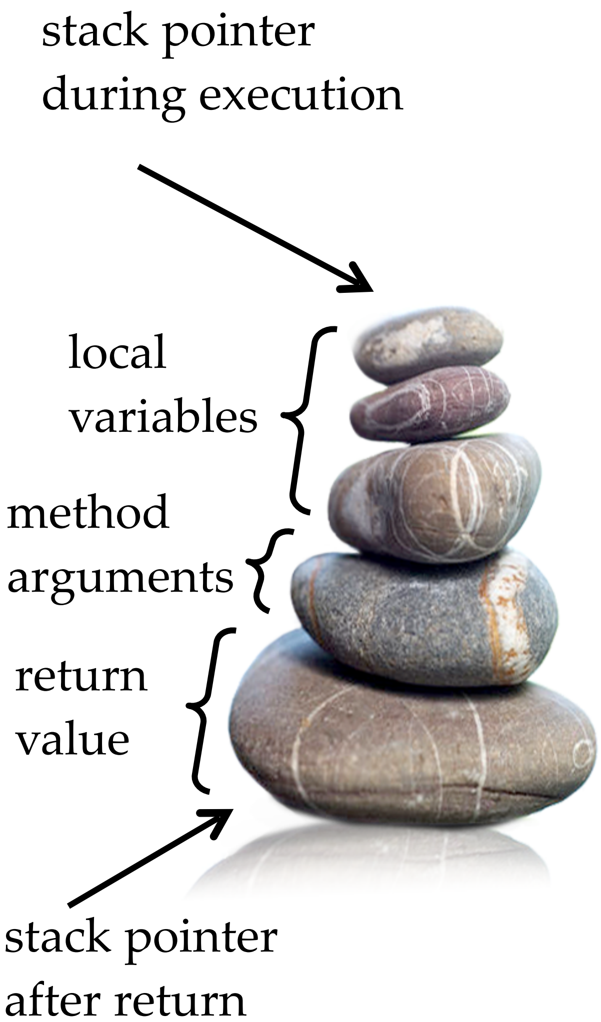
What is memory management

- These days memory management is done behind the scenes by the compiler or garbage collector
- Understanding memory management will allow you to write more efficient code, in terms of memory usage and execution time

The stack and the heap

- When your program starts, it gets two areas of memory that it can use:
  - A stack, typically used for arguments and local variables within functions and methods.
  - A heap, used for objects and large/persistent data structures.

The stack—local scope



- The stack is a part of memory that is managed Last-In-First-Out.
  - To allocate / deallocate memory you either "push" things onto the stack, or "pop" them off.
- The stack is most often used to contain local variables, arguments and return values for function/method calls.
  - All of this is done automatically for you.
- The stack can be used up, in which case you have "stack overflow," which generally crashes your program in unpredictable ways.

The heap—global scope

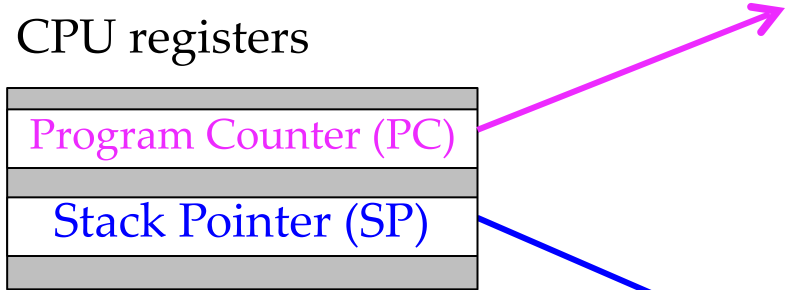
- The heap is a part of memory that is managed independent of program flow
  - You can add and remove things from the heap in any order from anywhere within your program

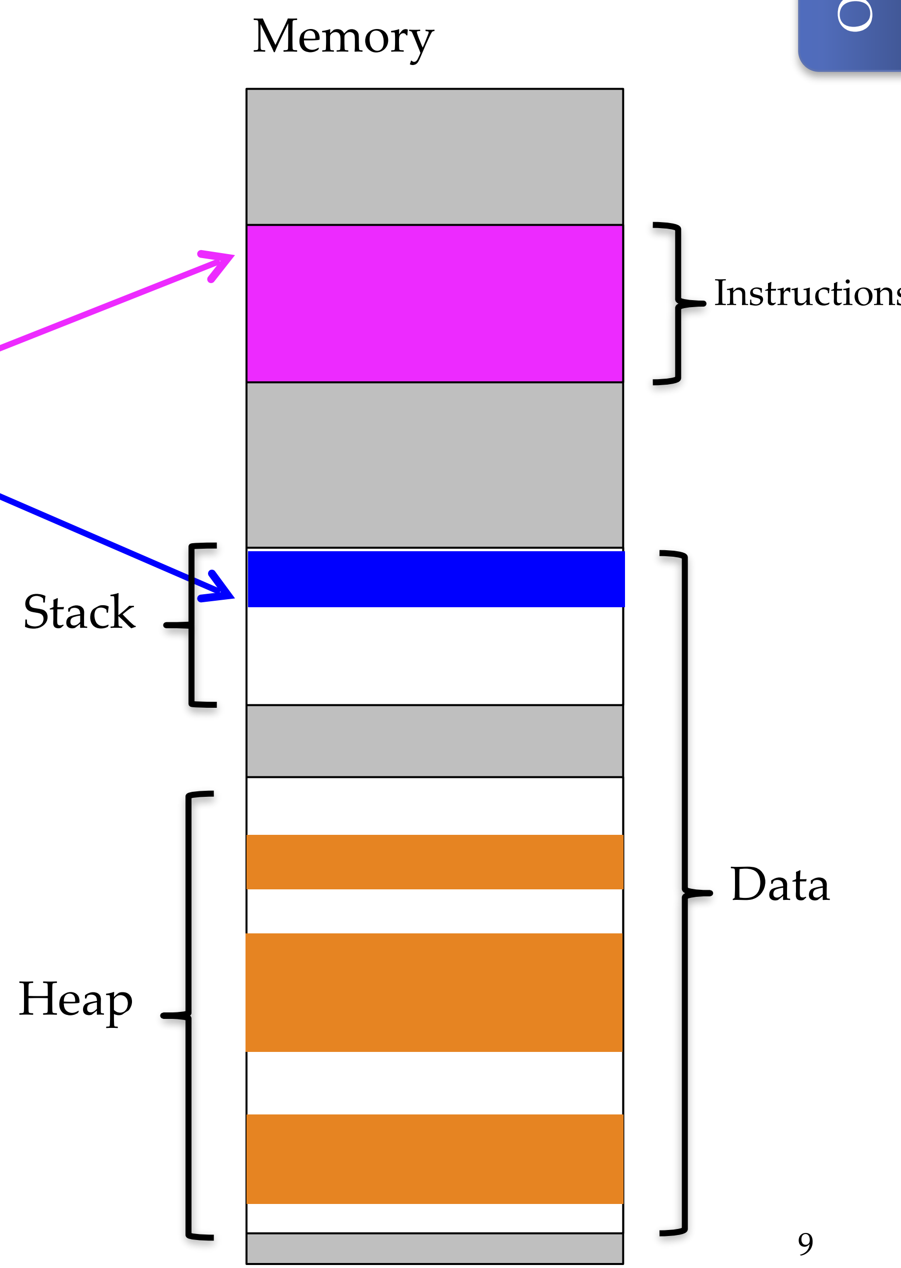
- Whenever you allocate a new object, it is added to the heap

```
var x: Fraction = Fraction()
```

adding a Fraction object

Memory





- Memory regions:
  - **Instructions**—read only, stores program code
  - **Data**—read/write, stores variable data
    - Stack—local scope data
    - Heap—global scope data
- CPU registers
  - **PC**—points to next instruction to execute
  - **SP**—points to the top of the stack

Short term vs. long term storage

Stack is short term:

- Fast allocation
- Fast access
- …but can be slow with large data structures (due to value copying)
- **Cleared automatically** on function exit

Heap is long term

- Slow allocation—have to go through memory management
- Slower access—data might not be contiguous (cache misses)
- …but no need to copy large chunks of data (only references get copied, data stays in place)
- **Not obvious when to clear**—is the data still needed?

Options for heap memory management

- **Manual**—e.g., C++
  - Programmer explicitly frees up (deallocates) any memory that has been allocated on the heap and is no longer needed
  - Prone to bugs and memory leaks
- **Automatic at run-time** (garbage collector)—e.g., Java
  - A special thread on the system scans through your program and removes objects that are no longer being used
  - Little chance of human error, but a bit of impact on execution —the collector takes a bit of CPU time
- **Automatic at compile-time**—e.g., Swift
  - Compiler figures out when objects are not referenced by any part of the program and places release calls appropriately
- All these options require some method of keeping track of the number of references made to an object

Reference counting

- In **reference counting**, each object keeps a **retain count**
  - The retain count tracks how many variables/objects hold a reference to that object
- If you want to keep a valid reference to an object, you send it a retain message
  - Retain count increments
- If you no longer need an object, you send it a release message
  - Retain count decrements
- When **retain count reaches 0**, the object is **deallocated**
  - Since no one wants to reference the object, there is no point keeping it in memory

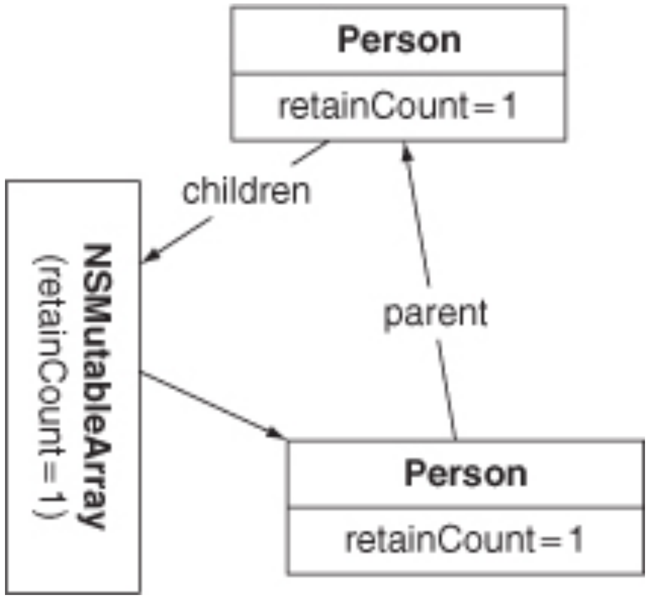
Automatic Reference Counting (ARC)

These days compilers are so smart that:

- They can figure out where to place retain and release calls
- It's all done automatically at compile time, so there is no need for the programmer to explicitly send retain and release messages
- It works extremely well, except in one scenario…

Retain cycles

- A **retain cycle** occurs when objects reference each other
    - Typically this occurs when a child references a parent



- In this situation the retain count will never go to zero on these objects—they will **never get deallocated**, even if they are not being used

Memory management

Swift

- All objects are allocated on the heap
  - Value variables such as integers, strings, *etc.*, are allocated on the stack
  - Variables defined as some class type are references
  - References are stored on the stack, but the object data which they refer to is allocated on the heap
- Compiler uses **Automatic Reference Counting** (ARC)
  - No need for the programmer to send retain and release messages, but …
  - it is the **programmer's responsibility** to ensure that there are **no retain cycles** in the program!

Initialisers & deinitialisers

Swift

- Initialisers are required for object instantiation and stored property initialisation
- Deinitialisers are optional, as memory is released by ARC
  - … but sometimes useful for manually allocated resources

```
class Shape {
    var pos: CGPoint;

    init(pos: CGPoint) {
        self.pos = pos;
    }

    deinit {
        //Shape specific de-initalisation
    }
}

var s = Shape(pos: CGPoint(x: 0, y: 1))
```

Initialiser & deinitialiser hierarchy

Swift

- In Swift, all properties of the child must be initialised before the call to parent's initialiser
  - Opposite of the convention in most languages, where the parent has to be initialised first
- No explicit calls from child to parent's deinit
  - Deinitialisers for child and parent will be invoked by the compiler

```
class Circle : Shape {
    var radius: Float

    init(pos: CGPoint, radius: Float) {
        self.radius = radius
        super.init(pos: pos)
    }

    deinit {
        //Circle specific de-initalisation
    }
}

var c = Circle(pos: CGPoint(x: 0, y: 1),
               radius: 3.0)
```

Weak and strong references

Swift

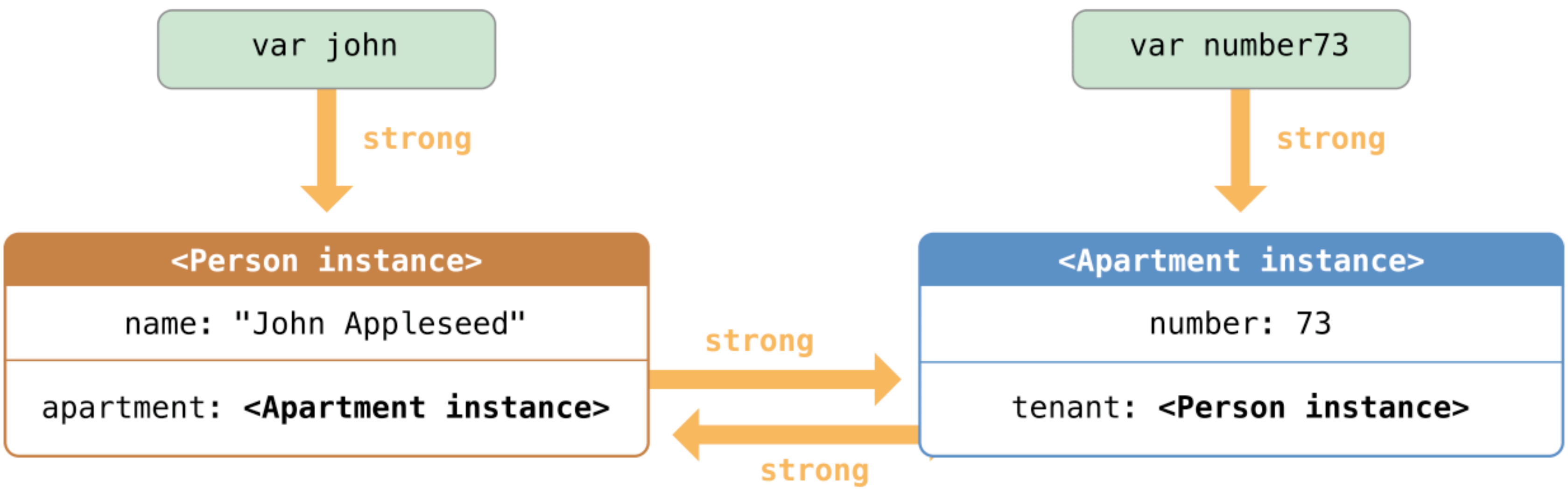
- Designed to combat retain cycle problems
- **Strong reference**—affects object's retain count
  - Retain count is incremented when strong reference is pointed to an object
  - Retain count is decremented when reference is destroyed or pointed elsewhere
  - **Swift references are strong by default**

```
class Person {
    let name: String
    var apartment: Apartment?

    init(name: String) { self.name = name }
}

class Apartment {
    let number: Int
    var tenant: Person?

    init(number: Int) { self.number = number }
}

var john = Person(name: "John Appleseed")
var number73 = Apartment(number: 73)

john.apartment = number73
number73.tenant = john
```

Weak and strong references

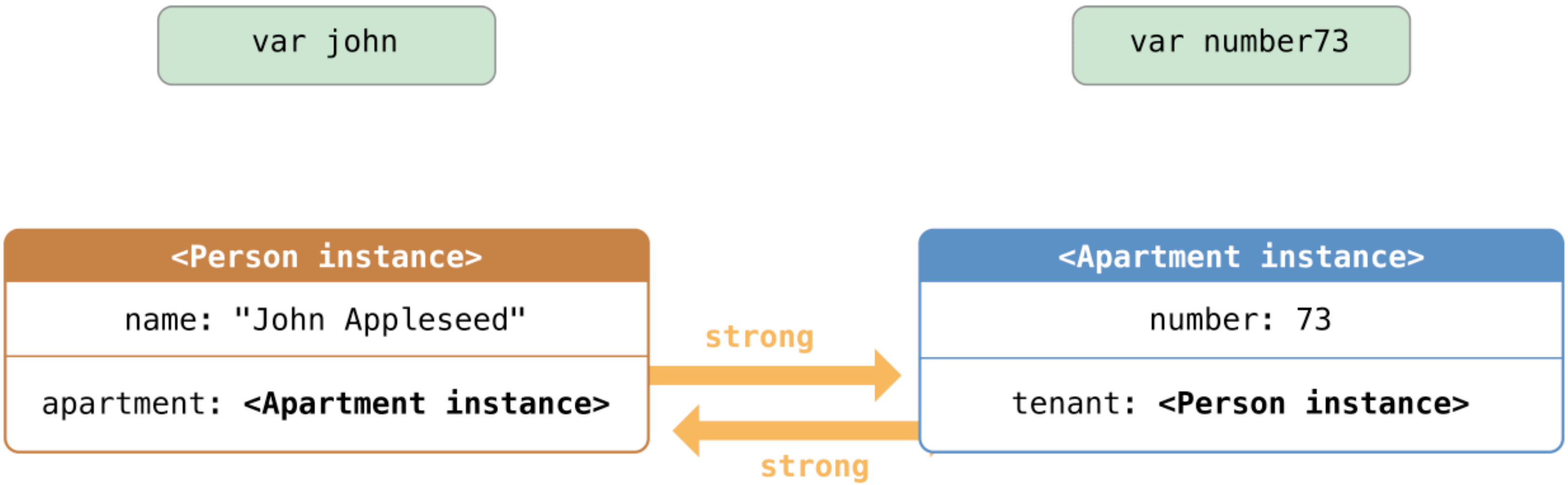
Swift

- Code on previous slide effects the graph of references illustrated below
- When the john and number73 references are changed we get …



Weak and strong references

- … a situation in which the interconnected Person instance and Apartment instance form a retain cycle
  - Neither will be deallocated—a memory leak



Weak and strong references

Swift

- **Weak reference**—does not affect object's retain count
  - The most common use of a weak reference is when a child references a parent

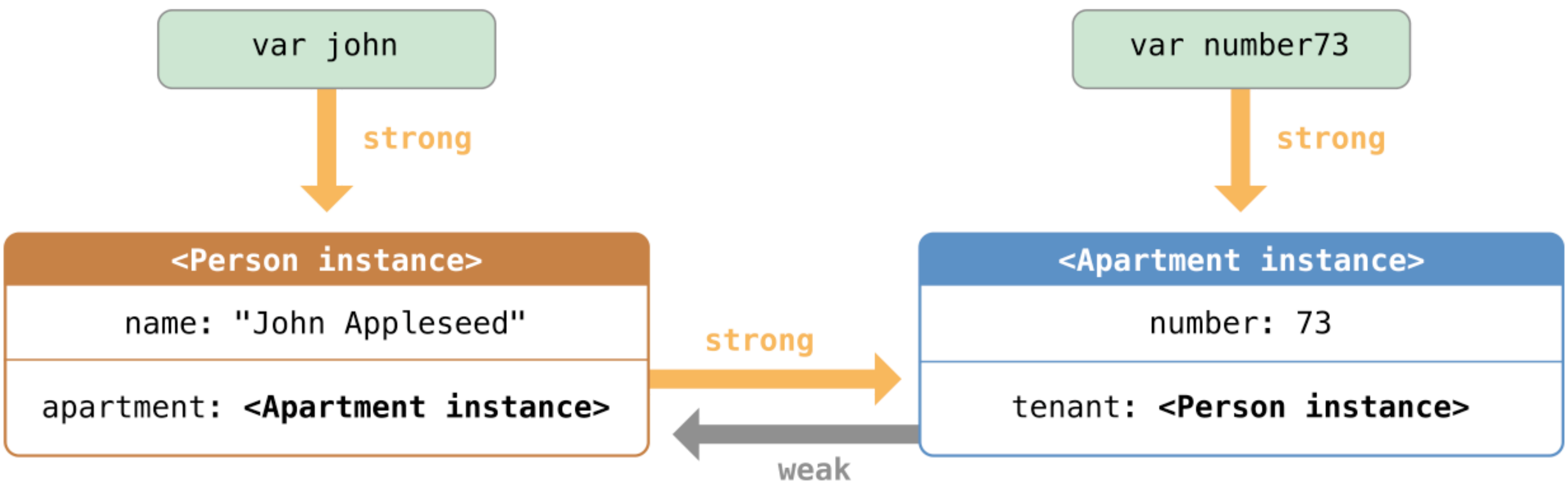
```
class Person {
    let name: String
    var apartment: Apartment?

    init(name: String) { self.name = name }
}

class Apartment {
    let number: Int
    weak var tenant: Person?

    init(number: Int) { self.number = number }
}

var john = Person(name: "John Appleseed")
var number73 = Apartment(number: 73)

john.apartment = number73
number73.tenant = john
```

- The graph of references for this code …

Weak and strong references

Swift

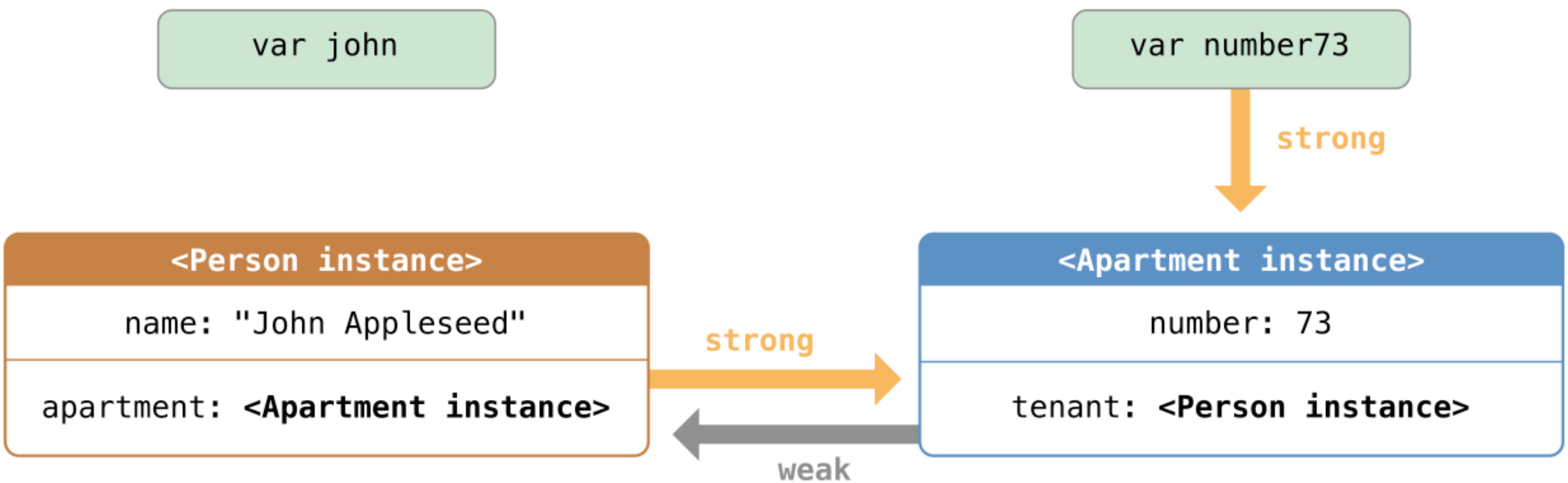
- … means that the Person instance only has a reference count of one
- Thus if we change the john reference …



Weak and strong references

Swift

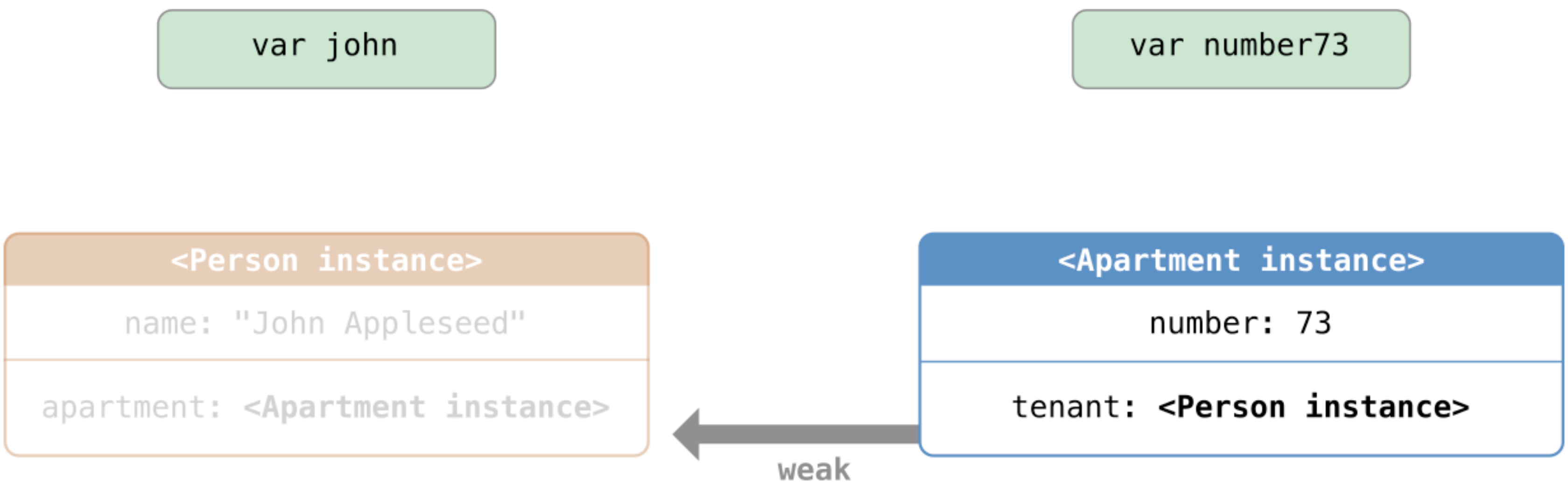
- The reference count of the Person instance will drop to zero and will be deallocated



Weak and strong references

Swift

- Once deallocated the weak reference in the Apartment instance will become nil
- … and when the number73 reference is changed, the Apartment will be deallocated



Design Pattern - Flyweight

- Structural pattern
- Minimise memory for similar objects
- Share data
  - Intrinsic - internal
  - Extrinsic - external, immutable, shared

- Tradeoff encapsulation and memory
- Savings are a function of:
  - reduction of the number of instances
  - amount of intrinsic state
  - whether extrinsic state is computed or stored

Flyweight - Example

Toolmaker

```
protocol Potion{
    func drink()
}
class HealingPotion: Potion{
    func drink(){
        print("You feel healed.")
    }
}
class HolyWaterPotion: Potion{
    func drink(){
        print("You feel blessed.")
    }
}
class InvisibilityPotion: Potion{
    func drink(){
        print("You become invisible.")
    }
}
enum PotionType{
    case HealingPotion,
        HolyWaterPotion, InvisibilityPotion
}
```

Builder

```
var inventory = [PotionType:Int]()
inventory[PotionType.HealingPotion] = 5
inventory[PotionType.HolyWaterPotion] = 1
inventory[PotionType.InvisibilityPotion] = 2
```

```
func usePotion(which: PotionType,
               inventory: inout [PotionType:Int]){

    if let count = inventory[which]{
        var potion:Potion
        switch(which){
        case PotionType.HealingPotion:
            potion = HealingPotion()
            break
        case PotionType.HolyWaterPotion:
            potion = HolyWaterPotion()
            break
        case PotionType.InvisibilityPotion:
            potion = InvisibilityPotion()
            break
        }
        potion.drink()
        inventory[which] = count - 1
    }
}
```

```
usePotion(which: PotionType.HealingPotion,
     inventory: &inventory)
usePotion(which: PotionType.HolyWaterPotion,
     inventory: &inventory)
usePotion(which: PotionType.InvisibilityPotion,
     inventory: &inventory)
```

Factory in the real world?

Summary?