Objects, UML, and Kotlin

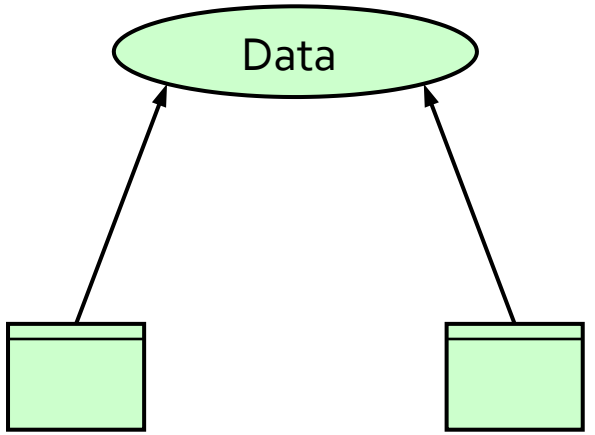
Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong

Modeling Principles

Language Evolution

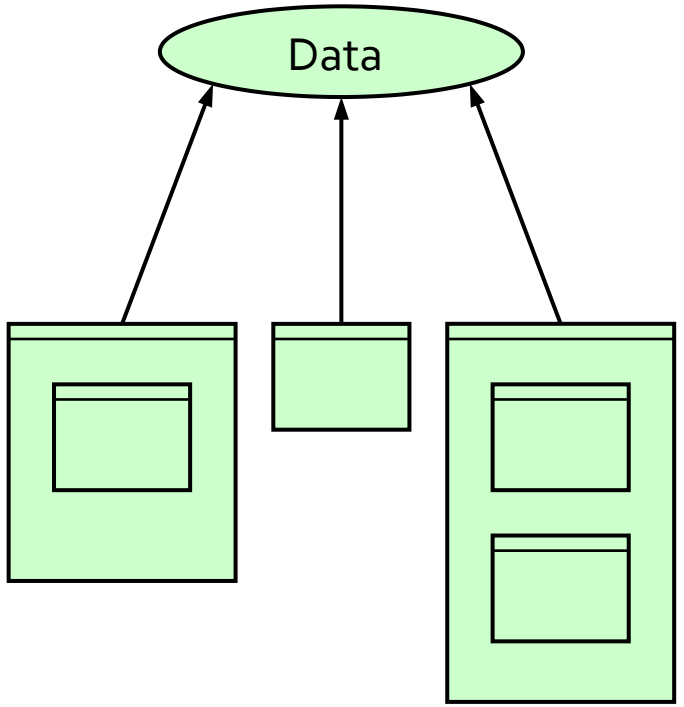
- COBOL, Fortran:
  - Subprograms (subroutines) access global data
  - Break up system into subroutines



Subprograms

Language Evolution

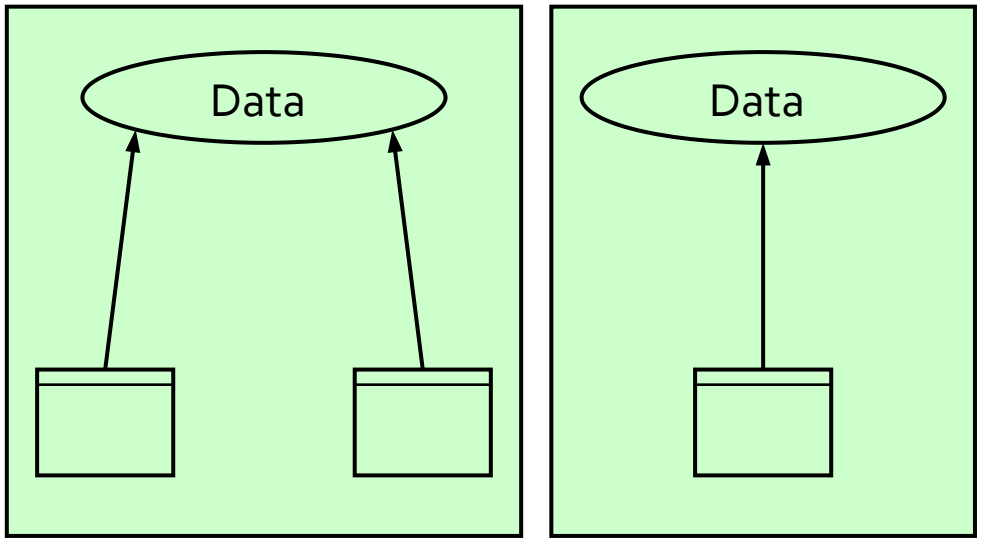
- Algol, Pascal:
  - (Nested) procedures with block structured scope
  - Break up system into nested procedures



Nested procedures

Language Evolution

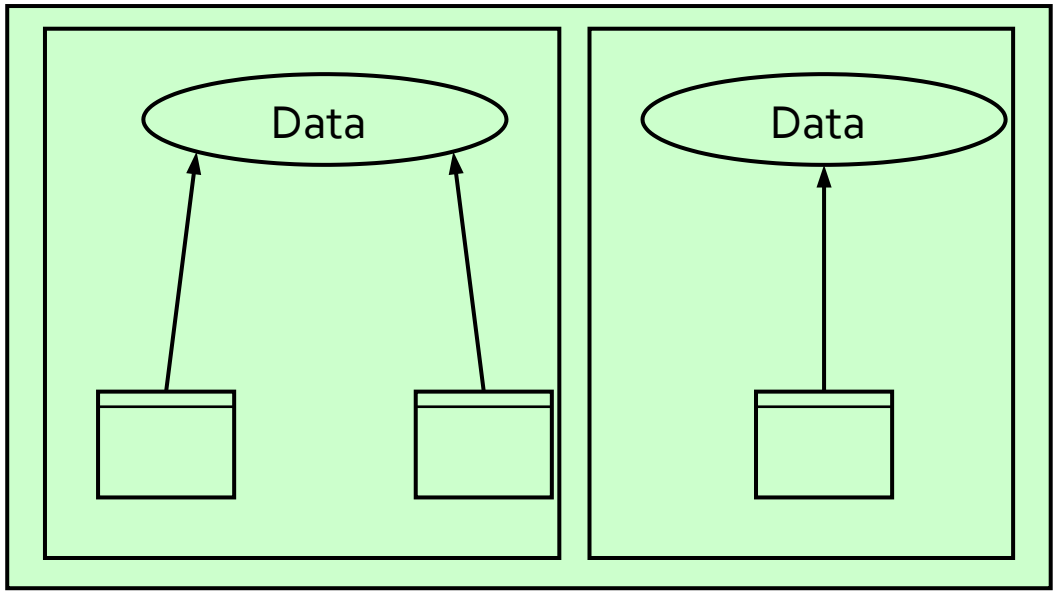
- Modula-2, C:
  - Modules (files) of related data and functions
  - Break up system into modules (e.g., abstract data types)



Modules

Language Evolution

- Smalltalk, C++, Java:
  - Classes with data and methods
  - Classes as "factories" for objects
  - Break up system into classes



Classes and packages

Discussion

- Question:
    - What software engineering design principles drove programming language evolution?

Abstraction

- Simplifying to its essentials the description of a real-world entity or concept
    - Coping with complexity
    - "Selective ignorance"
- Modeling the problem space
    - E.g., a "Person" abstraction

Encapsulation

- Bundling data with access functions
  - Distinguishing "what" from "how"
  - "Need to know" restricted access
  - Maintaining integrity
- Information hiding criterion
  - Hide changeable internal details from the outside world, but reveal assumptions through interface
  - E.g., a "Person" abstract data type

Decomposition

- Dividing whole things into parts
  - Or composing whole things out of parts
  - “Separation of concerns”
- Data parts
  - Fixed or dynamic number
  - Sharing of parts
  - Lifetime of parts

Generalization

- From specific cases, looking for commonalities that can be factored out
  - Reusing common designs
  - Reducing redundant code
- Making systems flexible and extensible

Object-Oriented Models

Object-Oriented Models

- Implementing OO models:
  - OO programming languages
    - E.g., Kotlin, Java, C++
- Expressing OO models:
  - OO design notations
    - E.g., UML

Java

- Principal designer:
    - James Gosling, Sun Microsystems
- Language goals:
    - Simple, object-oriented
    - Robust, secure
    - Network and thread support
    - "Compile once, run anywhere"

Java

- Language design inspired by:

Language	Feature
Lisp	Garbage collection, reflection
Simula-67, C++	Classes
Algol-68	Overloading
Pascal, Modula-2	Strong type checking
C	Syntax
Ada	Exceptions
Objective C, Eiffel	Interfaces
Modula-3	Threads

Kotlin

- Principal designer:
    - Andrey Breslav, JetBrains
- Language goals:
    - Concise, readable, pragmatic
    - Object-oriented with functional features
    - Robust and safer through null safety
    - Multiplatform support: JVM, Android, JavaScript, Native

Kotlin

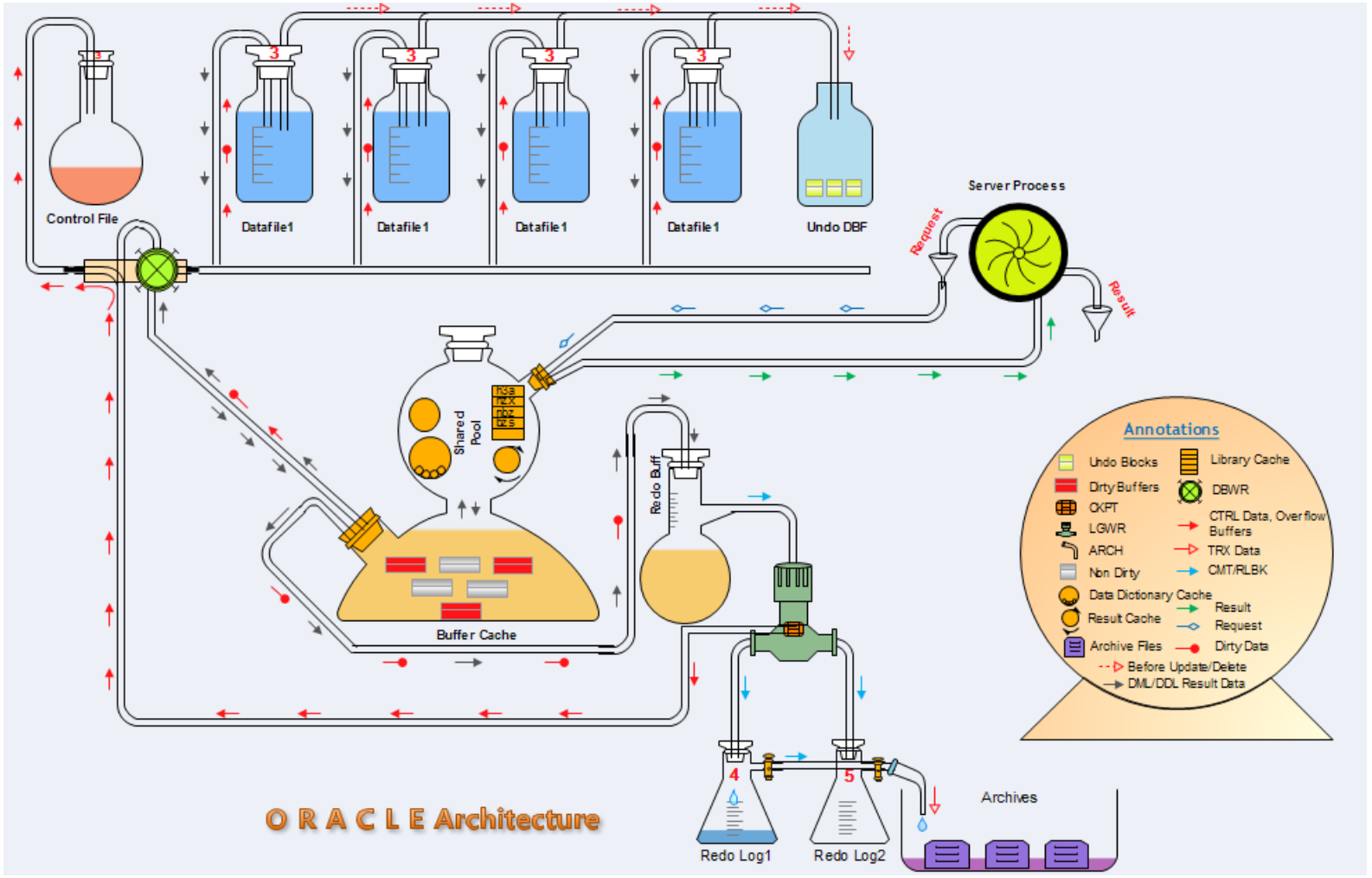
- Language design inspired by:

Language	Feature
Java	JVM compatibility, tooling, classes, interfaces, exceptions
Scala	Object-oriented + functional programming features
C#	Properties, null-safety style, modern concise syntax
Groovy	Concise JVM language, scripting-like convenience
JavaScript	Multiplatform/web support, familiar modern syntax

Unified Modeling Language (UML)

- Principal designers:
    - Grady Booch, Ivar Jacobson, James Rumbaugh
- Language goals:
    - Express object-oriented designs visually
    - Programming language independent
    - Communicate, evaluate, and reuse designs
    - Make design intent more explicit
- Allows thought about design before coding

Why UML?



Oracle©

Abstraction

- Object:
    - An entity with specific attribute values (state), behavior, and identity
    - Typically instantiated from a class
- Class:
    - Associated type of an object
    - Defines attributes and methods

Kotlin and UML Class

Classes are represented as a tall rectangle divided into several sections with the top section having the name of the class

- ```
  class Frame { // version 0
        // represents a 'window'
        /* body of class definition goes here */
    }
  ```

UML class notation
  ```

Encapsulation

- Class:
  - Access control for attributes and methods
    - E.g., public or private
  - Access is not the same as visibility
  - "Design by contract"
    - Public interface represents a contract between the developer who implements the class and the developer who uses the class

Basic Kotlin Class

```
class Frame { // version 1
      // private implementation

      private var variablename: DataType

      fun methodname( argument: ArgumentType ): ReturnType {
          // implementation of method
      }
  }
```

Kotlin Class with Constructors

```
class Frame(
        // class constructors
        private val name: String,
        private var x: Int,
        private var y: Int,
        private var width: Int,
        private var height: Int
    ){ // version 2
        …
        fun resize(
            newHeight: Int, newWidth: Int ) { … }

        fun moveTo(
            newX: Int, newY: Int ) { … }

        fun drawText(
            text: String, x: Int, y: Int ) { … }
    }
// create a frame by passing values to the constructor
val f = Frame( "Untitled", 0, 0, 480, 640 )
```

Required to create a Frame

'private val' / 'private var' become UML properties

UML Class

Classes are represented as a tall rectangle divided into several sections with the top section having the name of the class

The second section of the class lists fields
The third section of the class lists methods

Frame
-x : Integer -y : Integer -name: String -width:Int -height:Int
+Frame(name: String, x: Integer, y: Integer, height: Integer, width: Integer) +resize(height: Integer, width: Integer): Unit +moveTo(x: Integer, y: Integer): Unit +drawText(text: String, x: Integer, y: Integer): Unit

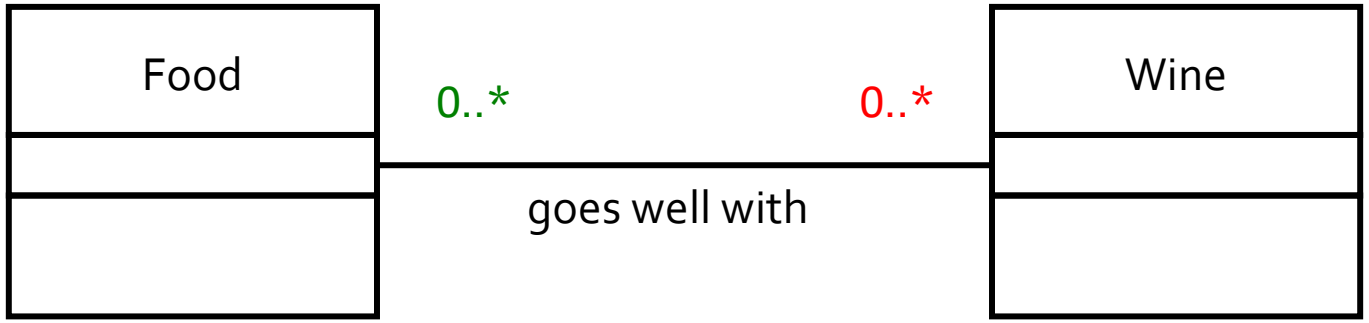
- – Private
- \+ Public
- # Protected

Decomposition

- Association relationship:
    - "Some" relationship between classes
        - E.g., between Book and Patron

UML Association

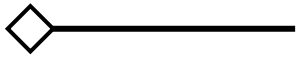
Association is a plain solid line with no arrows
It can be labelled to describe the association in the middle
Multiplicities can be added at either end



- Read class diagram using "objects"
  - A Food *object* goes well with a Wine *object*
  - A Food *object* is associated with 0 or more Wine *objects*
  - A Wine *object* is associated with 0 or more Food *objects*

Aggregation is a plain solid line with
a white (unfilled) diamond on the side with the collection
Multiplicities can be added at either end

Decomposition

- Aggregation relationship: 
  - Weak "has-a" relationship
  - Whole "has-a" part
  - A part may belong to (be shared with) other wholes
    - E.g., a Section and a Student

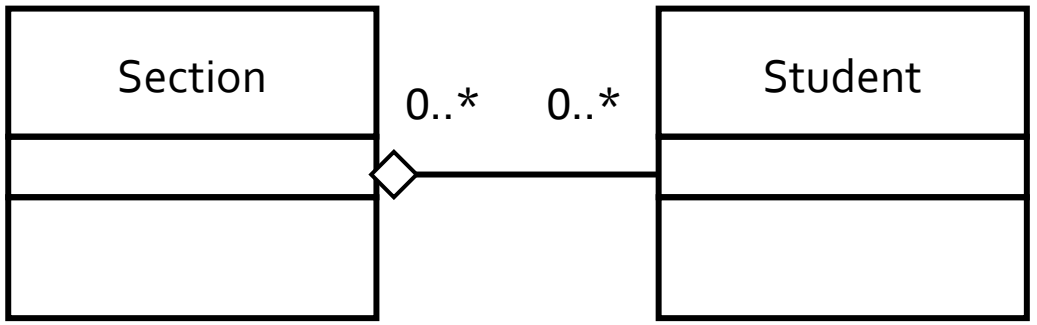
Aggregation is a plain solid line with
a white (unfilled) diamond on the side with the collection
Multiplicities can be added at either end

Kotlin and UML Aggregation

- Dynamic number of aggregated objects:

```
class Section {
    private val roster: MutableList<Student> = ArrayList()
    …

    fun add( s: Student ) {
        roster.add(s)
    }
}
```

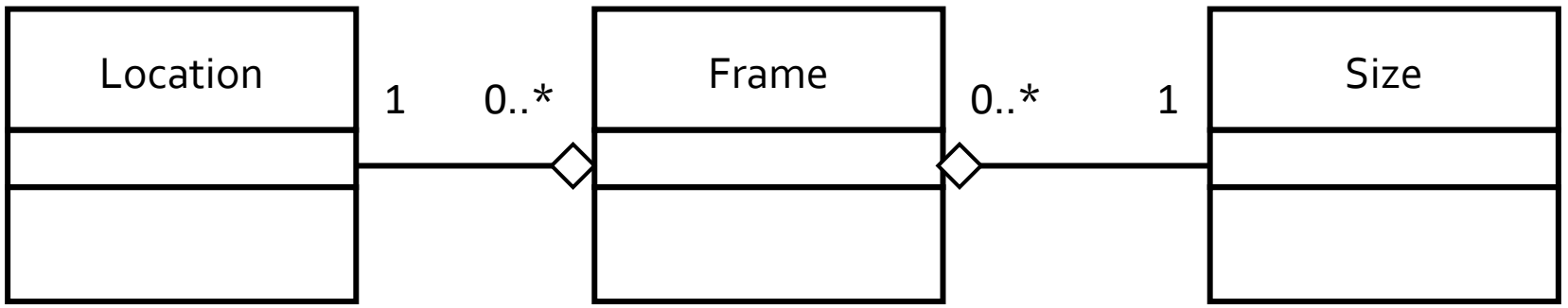


Notice that the diamond is on the Section side

Kotlin and UML Aggregation

- Fixed number of aggregated objects:

```
class Frame (
    private val defaultLocation: Location, // shared
    private val defaultSize: Size          // shared
) {
    …
}
```



Composition is a plain solid line with
a black (filled) diamond on the side with the collection
Multiplicities can be added at either end

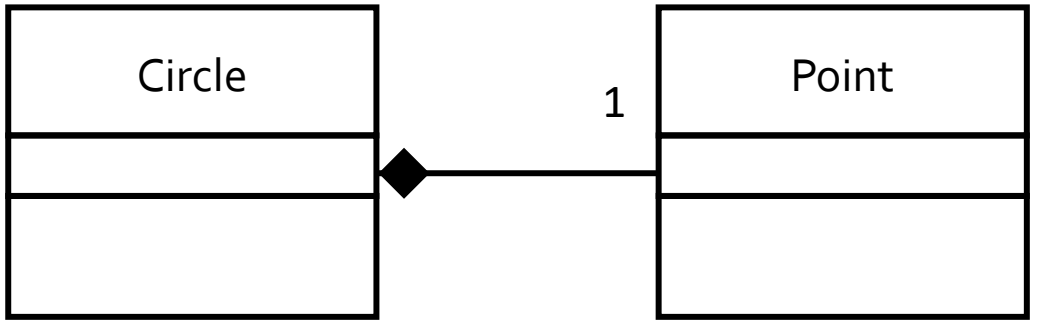
Decomposition

- Composition relationship: 
  - Strong "has-a" relationship
  - Exclusive containment of parts
  - Related object lifetimes
    - The whole cannot exist without having the parts; if the whole is destroyed, the parts should also be destroyed
  - Often access the parts through the whole

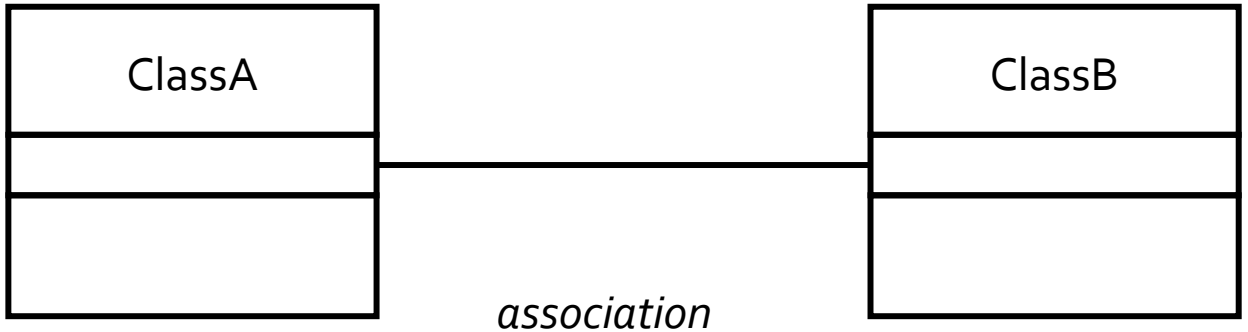
Notice that the diamond is on the side of the Circle

UML Composition

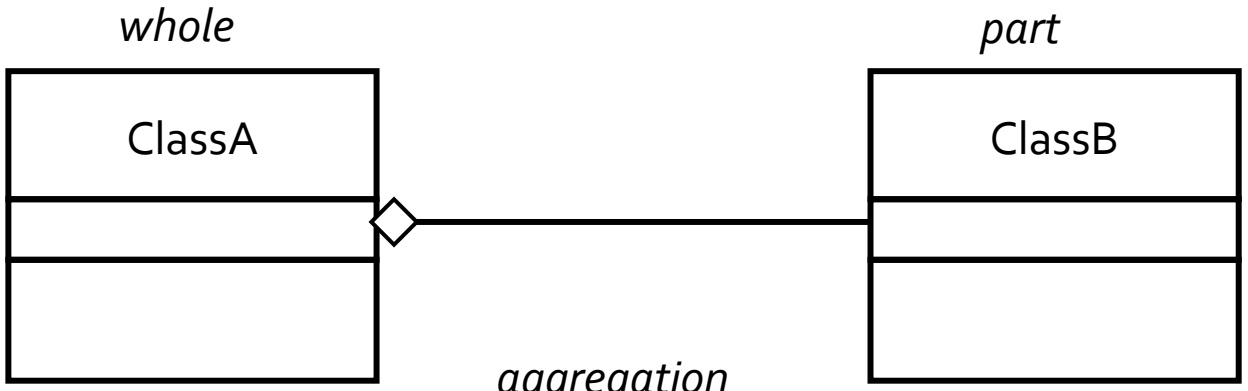
- Contained *objects* are exclusive to the container



- A Circle object has a Point object that is exclusive to it (however, other objects may contain Point objects, just not this one)

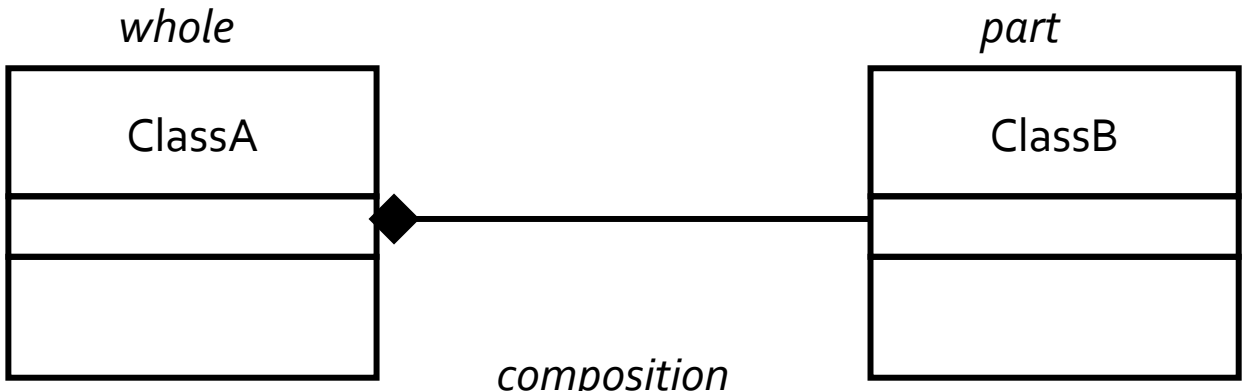


association



aggregation

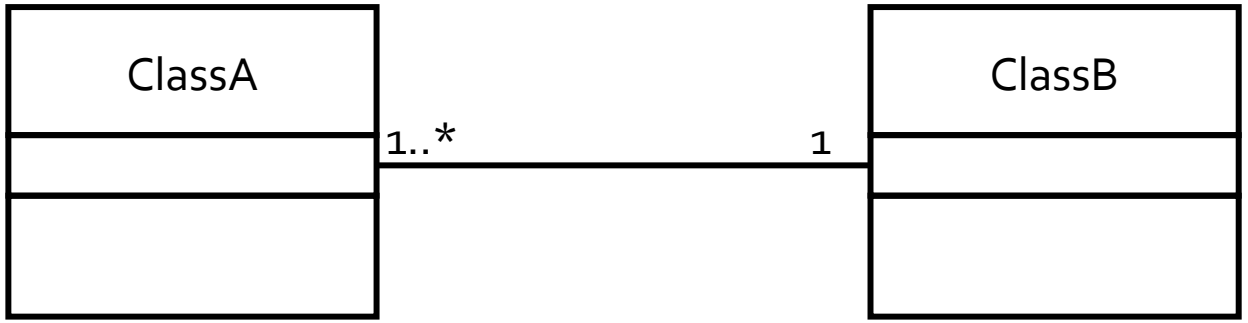
ClassA and ClassB can created/destroyed independently



composition

If ClassA instance is deleted, all of its ClassB instances get deleted too

Navigability

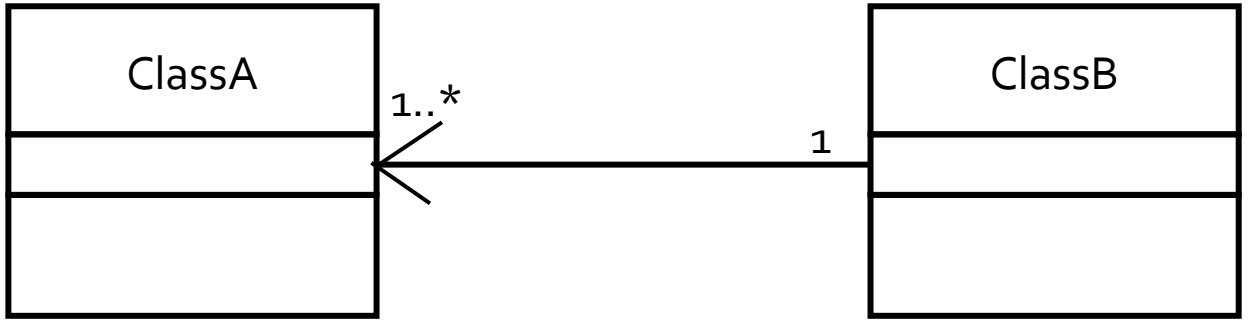


Missing navigability

Usually implies:

A instances have references to one B

B instances have references to one or more A



Navigability

← Arrow

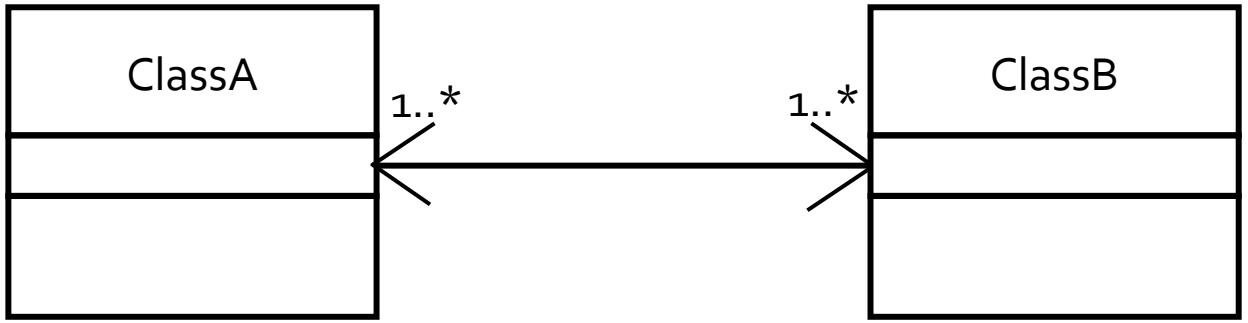
B instances have references to one or more A

A instances DO NOT have reference(s) to B

Navigability is a plain solid line with
a skeleton arrowhead showing
the direction we can navigate
Multiplicities can be added at either end

No arrowheads (same as association) usually means we can navigate Both ways, but it could mean that the author just didn't bother to specify

We can put both skeleton arrowheads on to explicitly specify that we can navigate either direction



Explicit two-way navigability

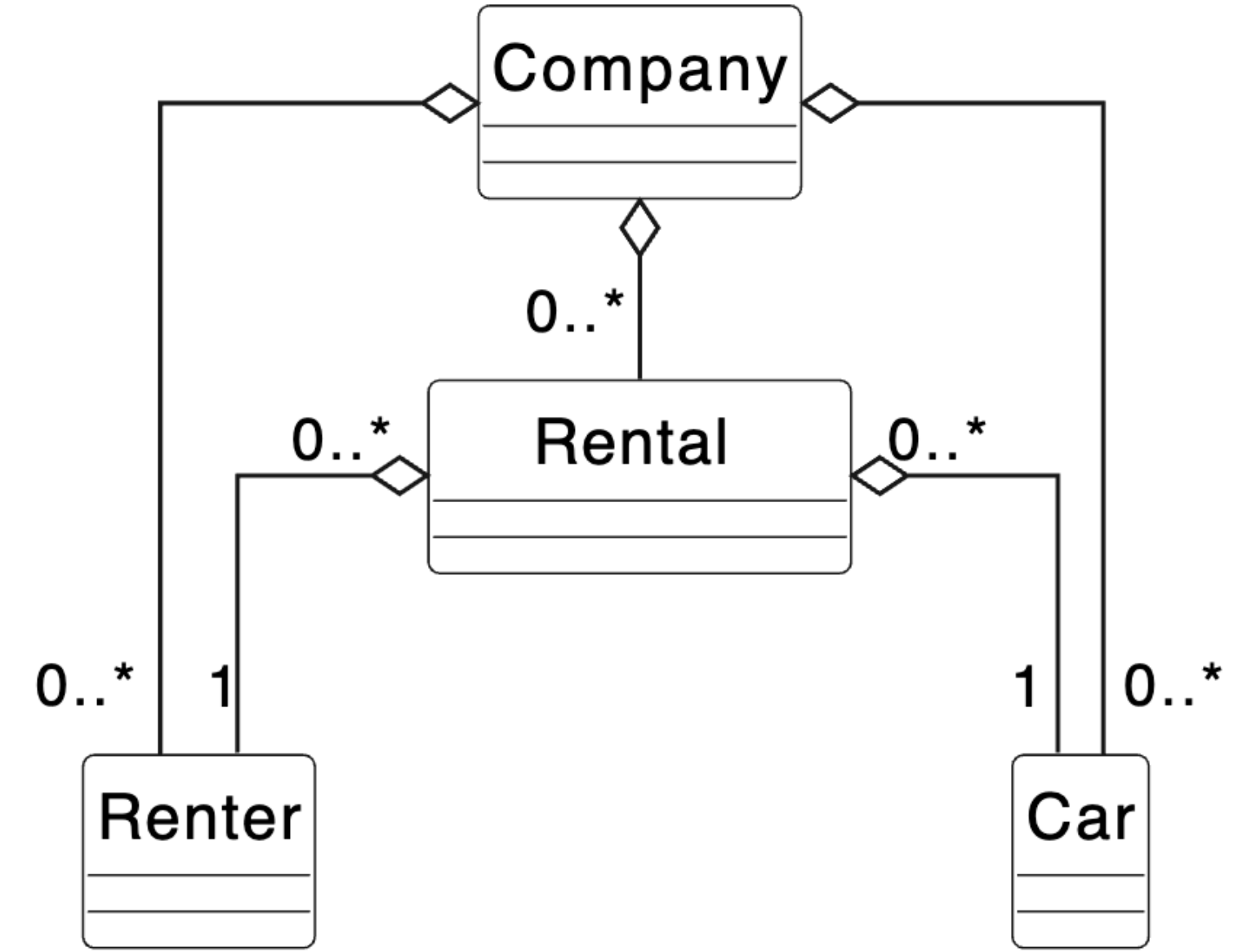
(rare)

A instances have references to one or more B

B instances have references to one or more A

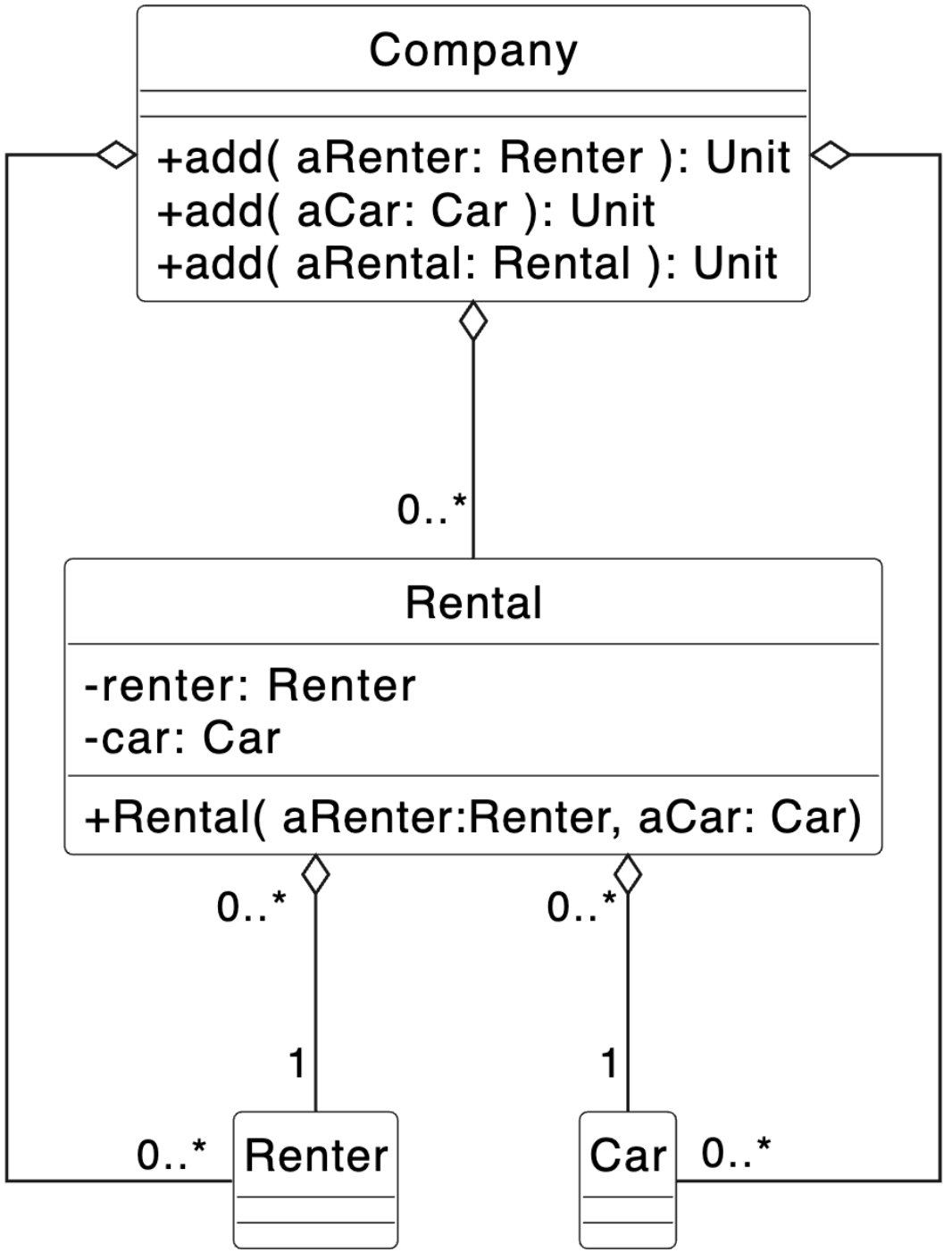
Exercise

- Analyze a UML class model for a car rental company that keeps track of cars, renters, and renters renting cars.


Company
0..*
Rental
0..*
0..*
0..*
1
1
0..*
Renter
Car

Exercise

- Implement the corresponding Kotlin code.



Generalization

Generalization

- Look for commonalities:
  - Common attributes
    - E.g., all vehicles have…
  - Common methods (behaviours)
    - E.g., all vehicles can…
- Generalize:
  - Find what is common and factor it out into a more general "base" abstraction

Generalization

- Implementation inheritance:
    - Generalize about method signatures, method implementations, and/or attributes
        - i.e., classes having these in common

Implementation Inheritance

- General part:
  - A base class (or "superclass") defines the attributes and methods to be shared
- Specific part:
  - A derived class (or "subclass") is endowed with the attributes and methods of its base class
  - A subclass may "extend" a superclass by adding attributes and methods, or overriding an existing method

Kotlin Implementation Inheritance

```
open class Shape { // superclass
    var myLocation: Location = Location()
        protected set
}


class Circle : Shape() { // subclass
    var diameter: Int = 0
        private set
}

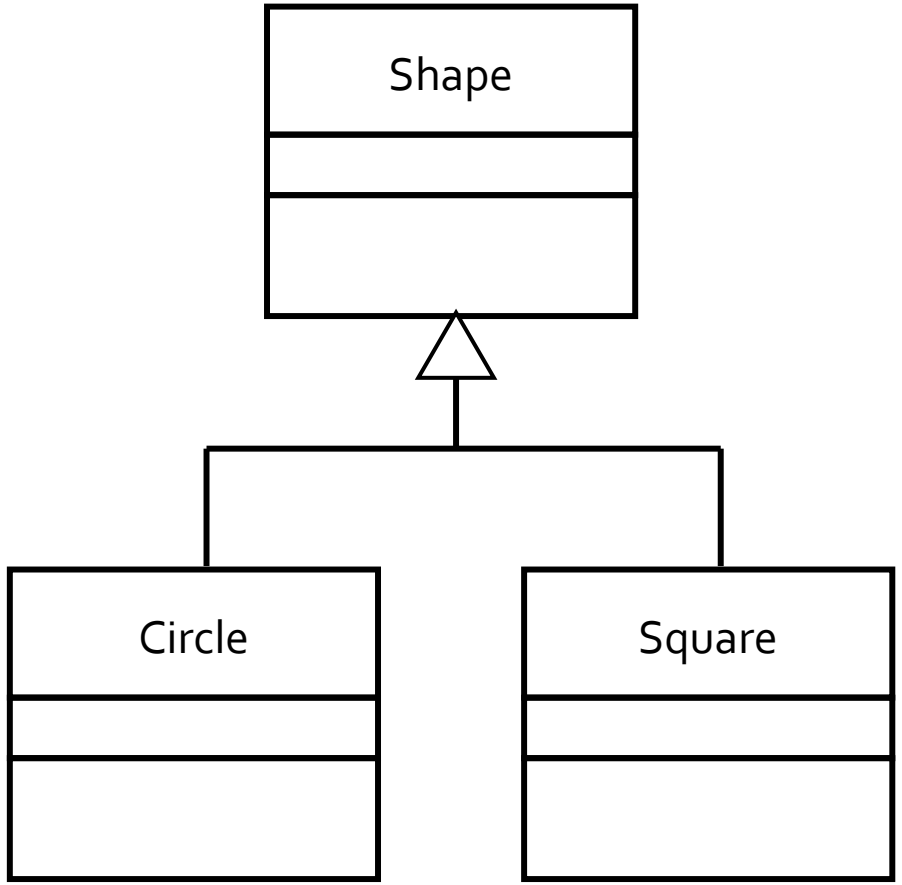
class Square : Shape() { // subclass
    var side: Int = 0
        private set
}
```

UML Inheritance

- Implementation inheritance relationship:
  - "Is-a" relationship between classes
  - i.e., subclass "is-a" kind of superclass
  - i.e., subclass "inherits from" superclass
- E.g., Circle "is-a" kind of Shape

UML Inheritance is a line going up to an open (white) triangle arrow head

Subclasses should be below superclass



Generalization Principles

- Inappropriate inheritance:
  - Subclass inherits from superclass but "is-a" (is a kind of) relationship *does not* exist
  - If "is-a" test fails:
    - Likely not appropriate
  - If "is-a" test succeeds:
    - *May* or *may not* be appropriate

Generalization Principles

- Liskov substitution principle:
  - An instance of the subclass should be substitutable anywhere a reference to a superclass object is used

```
var s: Shape
s = Circle() // instance of subclass
…
s.draw()     // superclass method
```

Inheritance Example

- Suppose:
  - open class Dog
    - Provides bark(), fetch()
  - class Cat : Dog()
    - Overrides bark(), overrides fetch(), and adds purr()
- Question:
  - Cat "is a" Dog?

Inheritance Example

- Suppose:
  - open class Window
    - Provides show(), move(), resize()
  - class FixedSizeWindow : Window()
    - “Hides” resize()
- Question:
  - FixedSizeWindow “is a” Window?

Inheritance Example

- Suppose:
  - open class ArrayList
    - Provides add(), get(), remove(), …
  - class ProjectTeam : ArrayList()
- Question:
  - ProjectTeam “is a” ArrayList?

Inheritance Issue

- Problem:
    - Superclass method is inherited, but it is not appropriate
    - What to do?

Inheritance Issue

```
open class Rectangle( s: Size ) {
    …
    fun setLocation( p: Location ) { … }
    open fun setSize( s: Size ) { … }
    fun draw() { … }
    fun clear() { … }
    fun rotate() { … }
    …
}

class Square(s: Size ) : Rectangle( s ) {
    // inherits setSize(), but want to "override" it
}
// Square 'is a' Rectangle?
// Square specializes Rectangle?
```

Override the Method Approach

```
class Square(s: Size) : Rectangle( s ) {
    override fun setSize( s: Size ) {
        // should not implement
    }

}
```

Aggregation Approach

```
class Square ( side: Int ) {
    private var rect: Rectangle = Rectangle( Size( side, side ) )
    // Square 'has a' Rectangle,
    // not 'is a' Rectangle

    fun setSide( newSide: Int ) {
        rect.setSize(
            Size( newSide, newSide )

    )
    }

    fun draw() {
        rect.draw()
    }
    …

}
```

Restructuring Approach

```
open class Quadrilateral {
    // Kotlin classes/methods are final by default unless declared open
    …
    open fun setLocation( p: Location ) { … }
    open fun draw() { … }
    open fun clear() { … }
    fun rotate() { … }
}

class Rectangle ( s: Size ) : Quadrilateral() {
    …
    fun setSize( s: Size ) { … }
}

class Square ( side: Int ) : Quadrilateral() {
    …
    fun setSide( side: Int ) { … }
}
```

Inheritance

- Kotlin abstract class:
  - May declare one or more abstract methods
  - Cannot be instantiated; must be subclassed
  - Abstract methods are implicitly `open` and must be implemented with `override`

Inheritance

```
abstract class Shape {
    abstract fun area(): Double
    abstract fun perimeter(): Double
    // there may be other instance data and methods

}
class Circle : Shape() {
    override fun area(): Double { … }
    override fun perimeter(): Double { … }
}
```

Interface Inheritance

- Kotlin interface:
    - Open by default
    - Declares function signatures
    - Classes implement interface using override for each function

```
interface Bordered {
    fun area(): Double
    fun perimeter(): Double
}

class Circle : Bordered {
    override fun area(): Double { … }
    override fun perimeter(): Double { … }
}
```

Interface Inheritance

- Kotlin interface:
  - A "contract", specifying *capabilities* that implementing classes must provide
  - Gives function signatures, but typically no implementation
  - Cannot be instantiated
  - May inherit from other (sub)interfaces

```
interface Transformable : Scalable, Translatable, Rotatable {
    …
}
```

Kotlin Interface

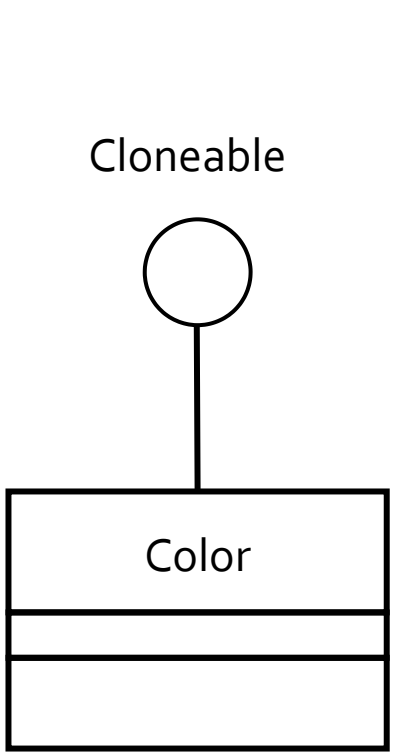
```
interface Copyable {
    fun clone(): Copyable
}

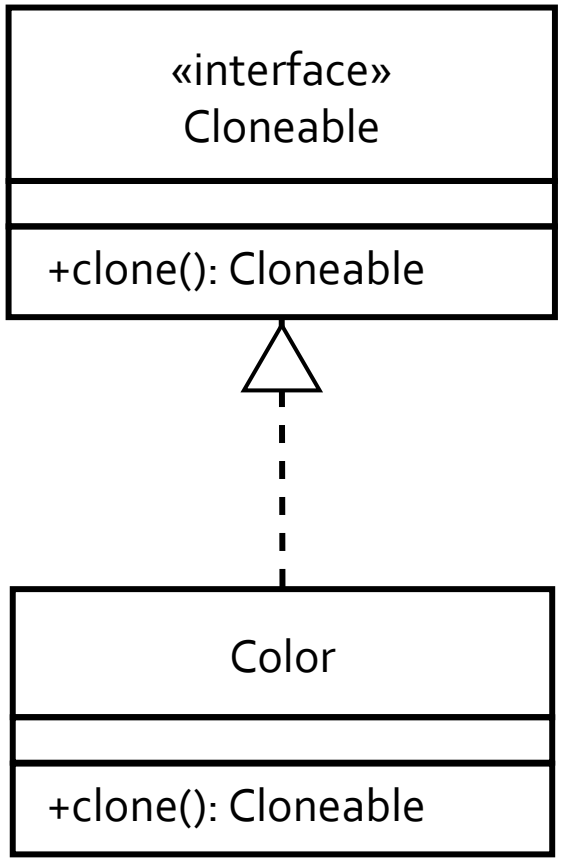

class Color(
    private var red: Int,
    private var green: Int,
    private var blue: Int
) : Copyable {
    override fun clone(): Copyable {
        return Color( red, green, blue )
    }
}



val red = Color( 255, 0, 0 )
val redClone: Copyable = red.clone()
val redClone2: Copyable = redClone.clone()
```

UML Interface



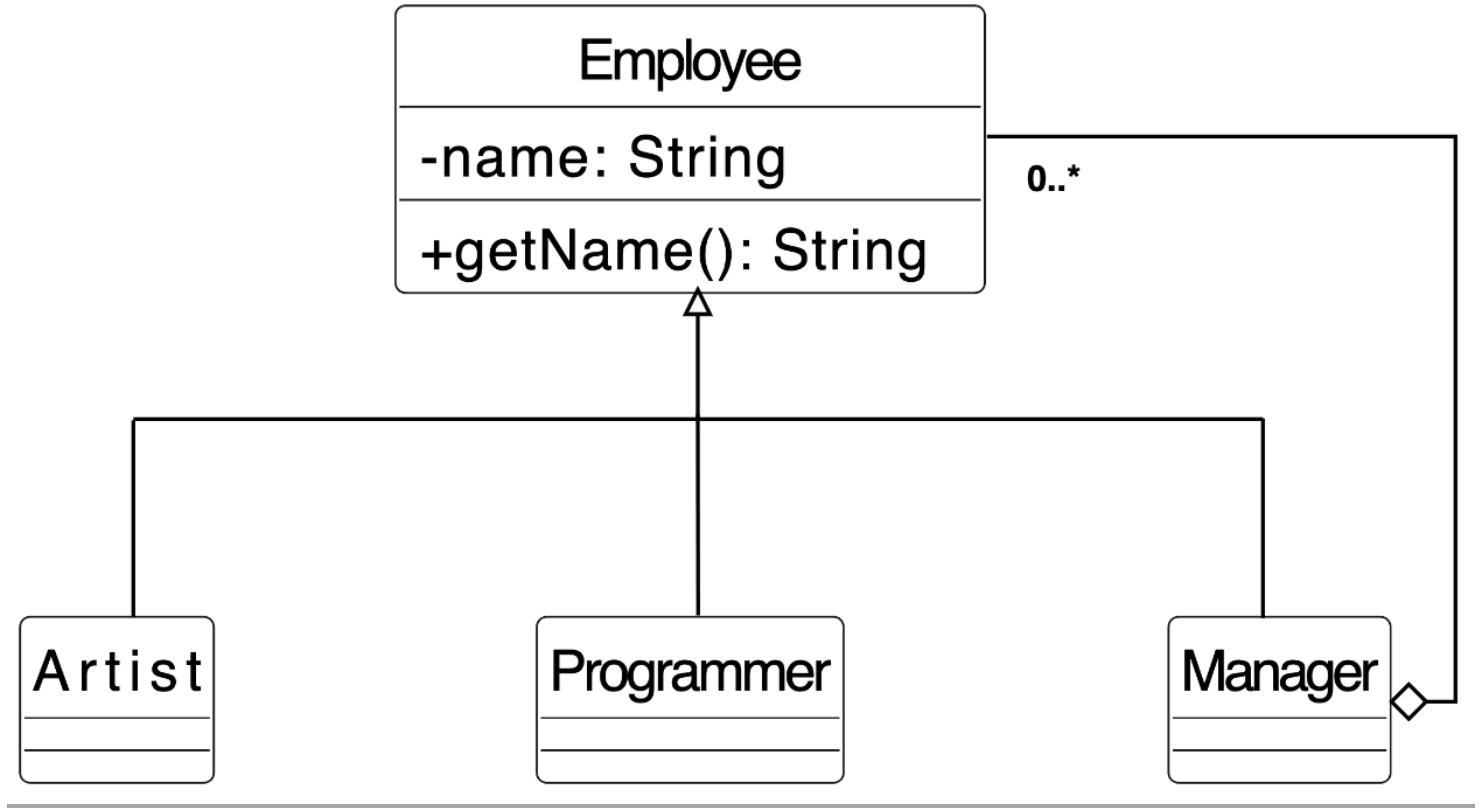


«Guillemets» denote a stereotype

Abstract Class versus Interface

- Differences:
  - An abstract class may provide a partial implementation
  - A class may implement any number of interfaces, but inherit from only one superclass
  - Adding a function to an interface will "break" any class that previously implemented it, unless a default implementation is provided
  - A function with a default body does not break existing implementing classes

Exercise



- Implement the corresponding Kotlin code.

Kotlin Subtleties

Kotlin Call-by-Value

```
class Sender {
    …
    fun send() {
        val r = Receiver()
        val argRef = Info()

        r.receive( argRef )
        argRef.doSomeMore()  // Still refers to same object
    }
}

class Receiver {
    …
    fun receive( infoRef: Info ) {
        infoRef.doSomething()
        // infoRef = Info()
        // error: function parameter cannot be reassigned
        // This code will not compile because you cannot
reassign
    }
}
```

Kotlin Constructors

```
open class Base {
    protected var value: Int = -1
}


class Derived : Base()


val d = Derived()
```

Kotlin Constructors

```
open class Base (
    protected var value: Int = -1
    // super() called implicitly, no explicit constructor needed
)


class Derived : Base() {
    // Base() is called before Derived is initialized
}


val d = Derived();
```

Kotlin Constructors

```
open class Base (
    protected var value: Int
)


class Derived(
    initValue: Int
) : Base( initValue ) // Base( initValue ) calls the superclass constructor


class DerivedAlt: Base {
    constructor( initValue: Int ) : super( initValue )
} // Secondary constructor calls superclass constructor


val d = Derived( -1 )
val dAlt = DerivedAlt( -1 )
```

Kotlin Dynamic Binding

```
open class Base {
    open var i: Int = 1
    open fun f(): Int { return i }
}

class Derived : Base() {
    override var i: Int = 2                  // overriding
    override fun f(): Int { return -i }    // overriding
}

fun test() {
    val d = Derived()
        // d.i is 2
        // d.f() returns -2
    val b: Base = d
        // b.i is 2 – overridden property
        // b.f() returns -2, 'dynamic binding'
    }
}
```

Kotlin Dynamic Binding

```
open class Base {
    // default implementation
    open fun op() { … }
}

class Derived1 : Base() {
    // does not override op()
}

class Derived2 : Base() {
    // override ...
    override fun op() { … }
}
```

Selection of function to be run is made at run time, depending on type of receiving object

```
var base: Base
base = Derived1()     // implicit upcast
base.op()             // calls op() in Base
base = Derived2()     // implicit upcast
base.op()             // calls op() in Derived2
```

Receiving object does the "right thing", even if the calling code does not show its actual type

Kotlin Dynamic Binding

- **Upcast:**
  - **"Widening" cast is safe due to the principle of substitutability**

```
val base: Base = Derived2() // implicit upcast
base.op() // calls op() in Derived2
```

- **Downcast:**
  - **"Narrowing" cast must be explicit**

```
val base: Base = Derived2() // implicit upcast
val derived: Derived2() = base as? Derived2
// explicit downcast
derived.op() // calls op() in Derived2
```

Object-Oriented Analysis and Design

UML and OOA&D

- Analysis:
  - Requirements specification activity
    - Create UML use cases and class diagrams
- Design:
  - Architectural design activity
    - Refine UML class diagrams
  - Detailed design activity
    - Refine UML class diagrams
    - Create UML sequence and state diagrams

Object-Oriented Analysis

- Steps:
    - Discover objects from problem domain
        - Nouns may lead to classes and attributes
        - Verbs may lead to relationships and methods
    - Use CRC cards to note the analysis
    - Evaluate

Problem Description

- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Nouns

- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Verbs

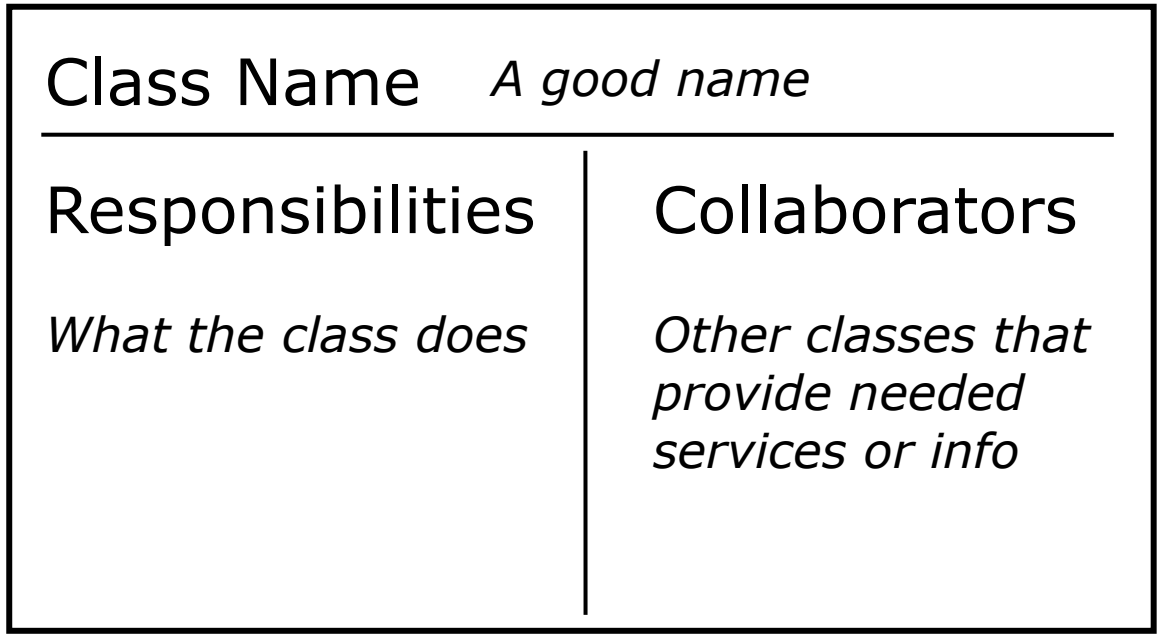
- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Discover Objects

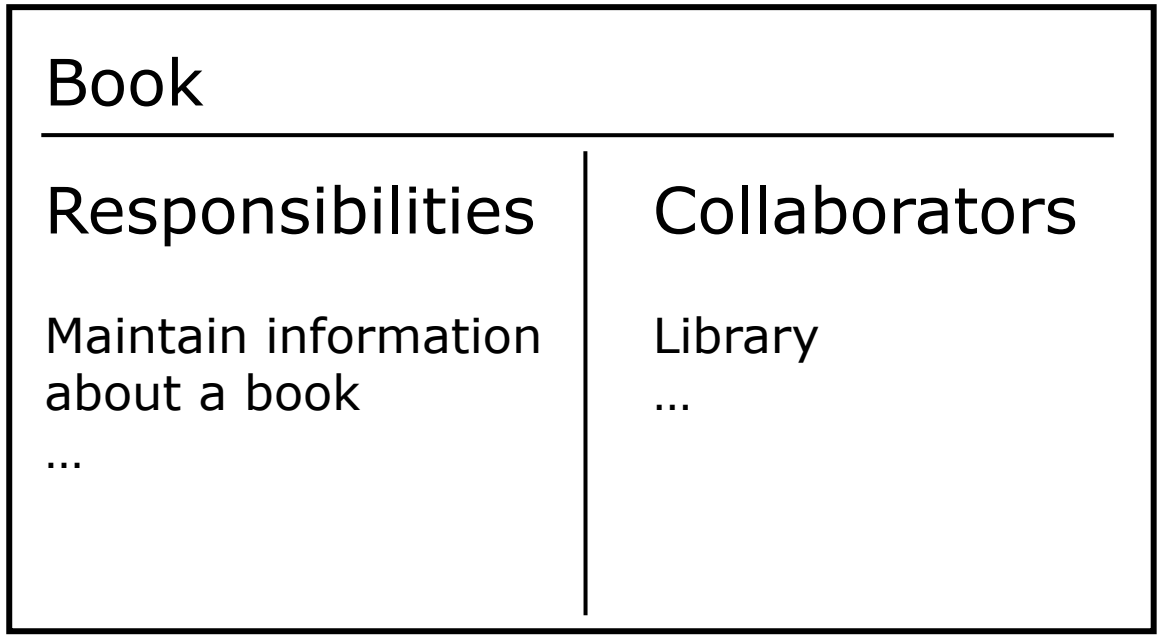
- Entity objects:
  - Things that model the problem domain
- Control objects:
  - Things that respond to events and coordinate services
- Boundary objects:
  - Things that interact with the system
    - E.g., other applications, devices, sensors, actors, roles, windows, forms

Use CRC Cards

- Class-Responsibility-Collaborator
  - Explore classes, their responsibilities, and their interactions
  - Organize index cards on a table



Use CRC Cards



Use CRC Cards

- Role playing:
  - Refine the cards by acting out a particular scenario with the candidate objects
  - "Become" the object
    - What do I do?
    - What do I need to remember?
    - With whom do I need to interact?
    - How do I respond?

Evaluate

- Principles:
  - During analysis, objects should initially be technology independent
  - If an object has only one attribute, perhaps it should not be a separate object at all
  - If an object has several highly related attributes, perhaps these attributes should form a separate object

Exercise

- Chess is a two-player, turn-based game where players move pieces on an 8x8 board. Each player begins a game with 16 initial pieces: 1 king, 1 queen, 2 rooks, 2 bishops, 2 knights, and 8 pawns. One player has the white pieces, and the other player has the black pieces. Pieces of different types move in specific ways or situations (e.g., navigating directionally, capturing an opponent's piece, castling, promoting, etc.).
- Consider a chess tutorial app, where a learner can tap on a piece and see its possible moves from which to choose. A learner can make a move upon the board. After making successive move(s), the learner can take back move(s).
- Using CRC cards, identify the key entity classes (and their generalizations as appropriate) in a well-designed, object-oriented model for this app. For each class, give it a good name, outline its main responsibilities, and list its collaborators.

Guidelines

- Get the big picture:
  - Understand the problem
    - Talk to the customer, end users, domain experts
  - Understand the target environment
    - Know the implementation constraints
  - Avoid reinventing the wheel
    - Reuse designs

Guidelines

- Modularity:
  - Increase cohesion
    - Class has a clear specific responsibility
  - Reduce coupling
    - Class is not connected to or knows too many others
  - Separate the layers
    - Identify entity, control, and boundary objects
    - Allow replacing layers

Guidelines

- Classes:
  - Use good names
    - Should be meaningful and explanatory
  - Avoid huge "blob" classes
    - A single class should not do everything
  - Use information hiding
    - Hide changeable details, reveal assumptions

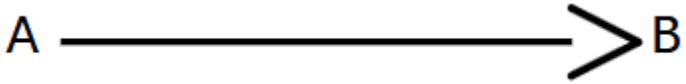
Guidelines

- Generalization:
  - Find superclasses
    - Look for and factor commonalities among classes
  - Apply Liskov principle for proper inheritance
    - Or use "is-a" test
  - "Is-a" test is not always enough
    - Class names can mislead, look at specific behaviour

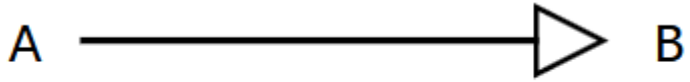
Guidelines

- Adaptation:
  - Hard to get it right the first time
    - Recognize problems and fix them
  - Your software won't go away
    - Make it easy to adapt to change
  - Simplicity (as simple as possible)
    - Does not always mean using the first thing that comes to mind
    - Elegant designs may need effort

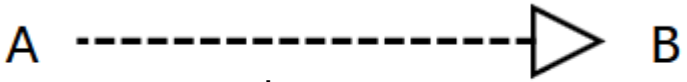
Association
Solid line, no arrows
Usually two-way navigability,
but sometimes just not specified



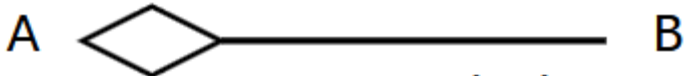
One-way Navigability
Solid line, skeleton arrowhead pointing at B
We can only follow references from A to B



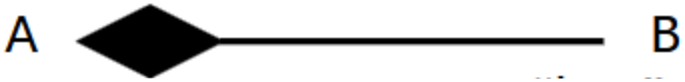
Inheritance
Solid line, white triangle arrowhead pointing at B
A extends B



Implementation
Dashed line, white triangle arrowhead pointing at B
A extends B



Aggregation: weak "has"
Solid black line with a white diamond the A side
A has some Bs
The same Bs that A has can
be shared
Not exclusive!



Composition: strong "has"
Solid line with a black diamond on the A side
B is a part of A:
When instance of A is deleted,
all of its B are also deleted

More Information

- Books:
  - Kotlin in Action, Second Edition
    - S. Aigner, R. Elizarov, S. Isakova, D. Jemerov
    - Manning, 2024
  - Head First Kotlin
    - D. Griffiths, D. Griffiths
    - O'Reilly, 2019

More Information

- Books:
  - UML Distilled
    - M. Fowler
    - Addison-Wesley, 2003
  - The Elements of UML 2.0 Style
    - S. W. Ambler
    - Cambridge, 2005

More Information

- Links:
  - UML Quick Reference
    - http://www.holub.com/uml/
  - Kotlin
    - Kotlin Overview by Android Developers
    - Kotlin: The Good, the Bad, the Ugly by Martin Häusler