

COMP 2511 8s2

Object Oriented Design & Programming

Introduction

Our Team

LiC: Aarthi Natarajan

a.natarajan@unsw.edu.au

Web: <http://webcms3.cse.unsw.edu.au/COMP2511/18s2/>

Course Admin:

Robert-Clifton Everest

About me...

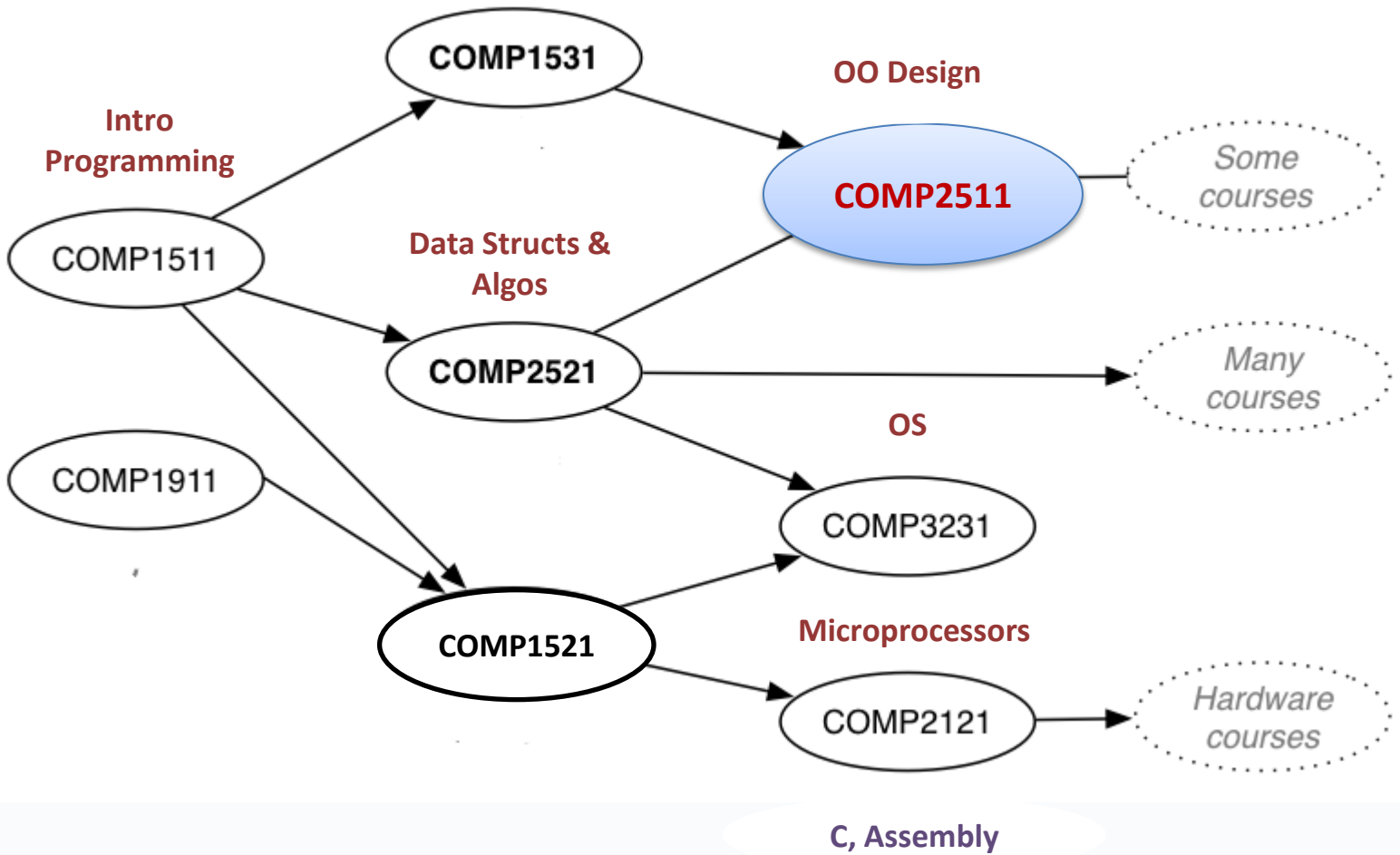
Aarthi Natarajan: (a.natarajan@unsw.edu.au)

Background:

- 16 years enterprise software development (IBM, Westpac, RTA...)
- 8 years instructor (Oracle Education)
- Email is the best way to contact me
- I believe in interactive lectures, so please ask questions!

Course Context

Software
Engineering



COMP 2511 thoughts

- What have you heard about COMP2511?
- Why take COMP2511?

Pre-requisites

- Completed **COMP 1511**
 - competent C programmers who can understand and use abstract data types
- Completed **COMP 1531**
 - understand and know how to use git and Github reasonably
 - be familiar with fundamental object-oriented design concepts
- **25%** cohort have not completed COMP 1531
 - Week 1 lab to familiarise with git/GitHub
 - Additional bridging material (particularly in weeks 1,2)

COMP 2511 Major Themes

❖ Object Oriented Design Process (SE)

- Understand the principles of object-oriented design
- Be able to follow a systematic object-oriented design process
- Be able to interpret and use tools for object-oriented design
- Learn how to apply design principles and design patterns effectively to design flexible, maintainable and reusable systems

COMP 2511 Major Themes

❖ Object Oriented Programming in Java

- Be able to write medium-scale object-oriented programs in Java

• Software Engineering process

- Problem solving – apply SE principles to solve a real-world problem
 - Be able to work within a small team in the context of a software development project
 - Be able to plan and execute a software project according a systematic software process

• User Interfaces

- Effective coding and testing techniques

Credit teaching material

- ❖ No text book, comprehensive lecture slides will be provided for COMP 2511 - some content and ideas are drawn from:
 - *Head First Design Patterns* , by Elisabeth Freeman and Kathy Sierra, The State University of New Jersey
 - *Refactoring: Improving the design of existing code* , by Martin Fowler
- ❖ Links to useful tutorials will be uploaded as necessary

How do we obtain our educational objectives?

❖ Lectures... (Recorded through Echo 360)

- 3 hour lectures
- Wed: 13:00, Fri: 10:00, weeks 1-12
- 4th hour (some weeks) will be devoted to practical demos and revision of COMP 1531

❖ Tutorials and Labs...

- 3 hours per week (1 hour tut followed by lab), weeks 1 – 12
- Tutorials re-enforce learning from lectures
- Labs comprise small design and practical programming exercises, individual or in pairs

❖ A Group Project...

- Project iteration demos scheduled in some lab sessions

Assessments

- Tutorial & Lab Sessions: 3 hours per week (12%)
 - Tutorials re-enforce learning from lectures and provide practical demonstrations
 - Marks will be awarded for tutorials based on participation
 - Lab sessions encompass both design and practical programming exercises
 - To obtain any lab marks for a Week X lab, you must:
 - *upload your completed solution to GitHub*
 - *demonstrate your work to your tutor in the lab class in the week it is due OR in the first hour of the following week (completed solution must be submitted to GitHub by Sunday of Week X)*
 - **Cannot** obtain marks by emailing solution to tutors
- **No tut/labs in week 10**

Assessments

- Online Quizzes (3%)
 - Review of content covered in lectures
 - Taken in your own time (via WebCMS3)
 - Due before Sunday 11:59 at end of week
- Assignment 1 (10%)
 - Specification will be released in week 2
- Final Exam (50%)
 - 3 hour final exam, more details through the term

Group Project

- Contributes to **25%** of the final course mark
- Carried out in teams of **4**
- Project specification will be released in Week 05
- Implemented using an *Agile Software Development Model*
 - Working software to be delivered in iterations
 - Marks will be awarded for each iteration demo, which will count towards your overall group project mark
 - Responsibilities to be assigned to each group member during each iteration and all team members **MUST** contribute equally (**Tutors will check GitHub**)
 - Final project demo held in week 13

Course Mark

- Course Work Mark =
Lab + Quiz + Assignment 1 + Group Project (out of 50)
- Exam Mark = Mark from the 3 hour final exam (out of 50)
 - ExamOK = Exam Mark $\geq 25/50$
- Final Course Mark (out of 100) =
Course Work Mark + Exam Mark
- Final Grade
 - UF, If !ExamOK (even if final course mark > 50)
 - FL, if Final Course Mark $< 50/100$
 - PS, if $50/100 \leq$ Final Course Mark $< 65/100$
 - CR, if $65/100 \leq$ Final Course Mark $< 75/100$
 - DN, if $75/100 \leq$ Final Course Mark $< 85/100$
 - HD, if Final Course Mark $\geq 85/100$

Supplementary Exam

- Students are eligible for a Supplementary Exam if and only if:
 - they cannot attend the final exam due to illness or misadventure
 - their Final Mark is in the range $47 \leq \text{Final Mark} < 50$ **AND** Exam Mark $\geq 24/50$ (in this case, final course mark is 50, if student passes supplementary exam)
 - a supplementary exam will not be awarded for any other reason.

System

- ❖ Most work done on Linux or Mac
 - Lab work and group project can be done on the **CSE machine labs** or your own device
 - Technology stack
 - Java 8 SE, Eclipse Oxygen
 - Use Eclipse Oxygen
- ❖ Collaboration and Versioning Tool - GitHub

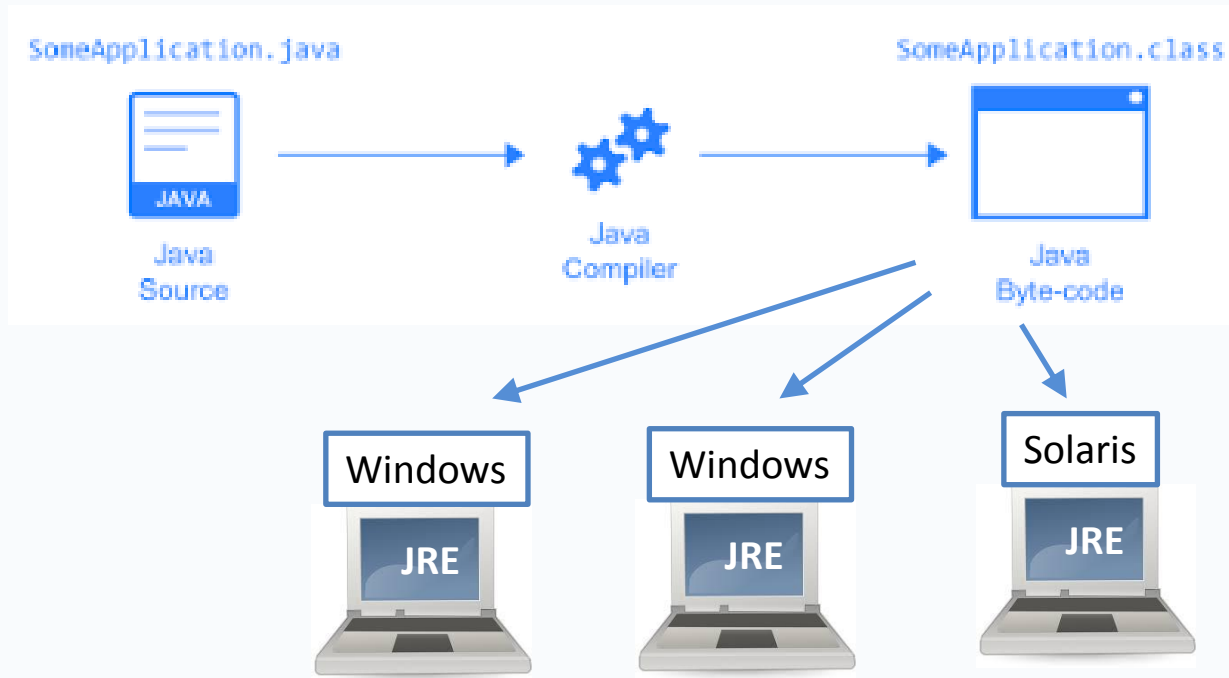
Plagiarism



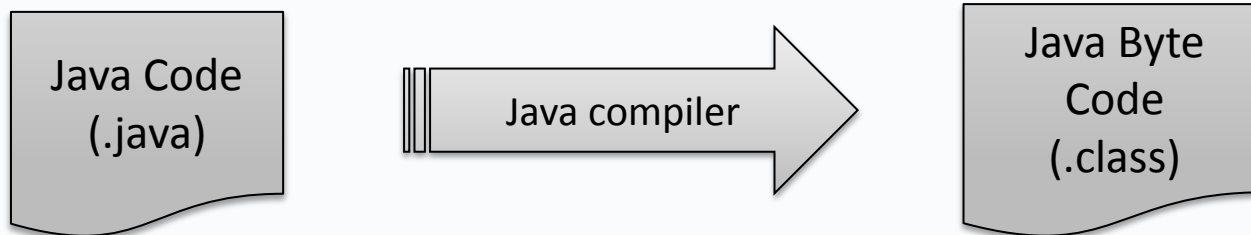
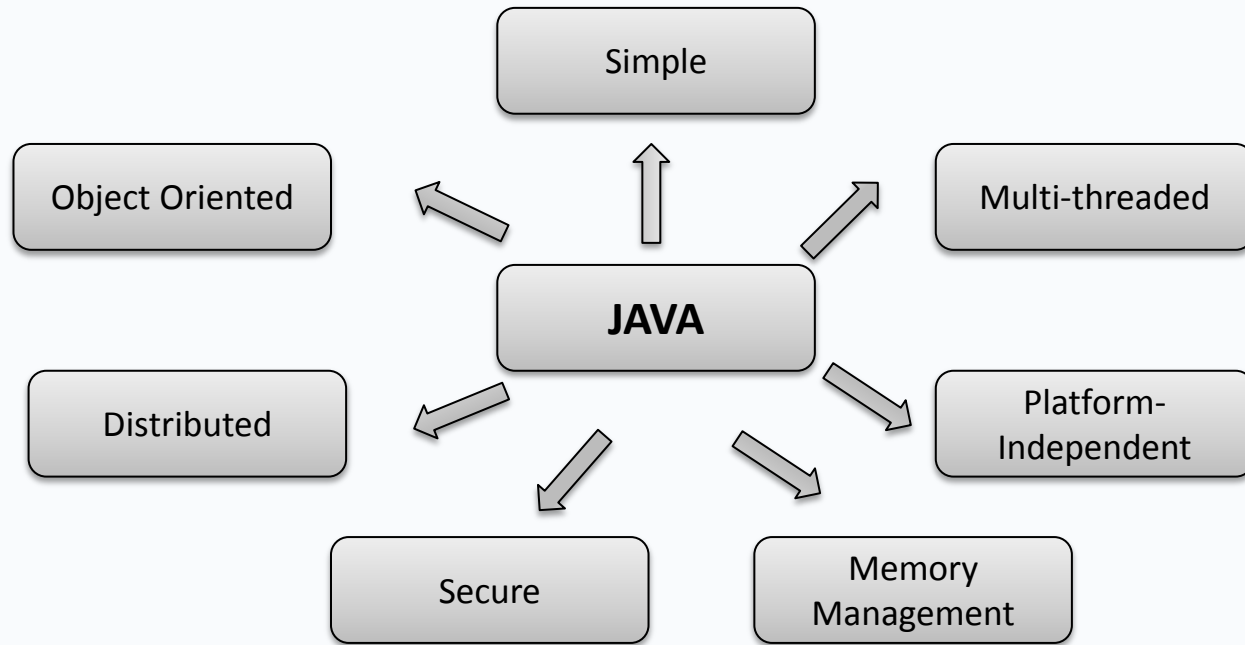
Just don't do it!

Introduction to Java Platform

Java is Platform Independent



The Java Platform



Resources

- Java SE Downloads:
<https://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Java Tutorials: <https://docs.oracle.com/javase/tutorial/>
- Java SE 8 API Documentation:
<https://docs.oracle.com/javase/8/docs/api/>
- Eclipse Oxygen Downloads:
<https://www.eclipse.org/downloads/packages/release/Oxygen/3A>
- Jenkov Java Tutorials: <http://tutorials.jenkov.com/java/index.html>

Java Language Basics

Lecture Demo

This week, we will look at:

- Java Language Basics
 - How to create a simple Java `class`
 - Structure of a Java `class`
 - Look at the `main` method
 - Creating primitive variables
 - Use control-loop structures `if-else`, `switch`
 - Iterate with loops
 - Create arrays
 - How to create comments
 - Working with Strings
- Use `package` and `import` statements
- Create Java classes, object instances, constructors, methods

Friday ...

Thinking Objects

- Classes and Objects
- Abstraction
- Encapsulation
- Inheritance

Thinking Objects

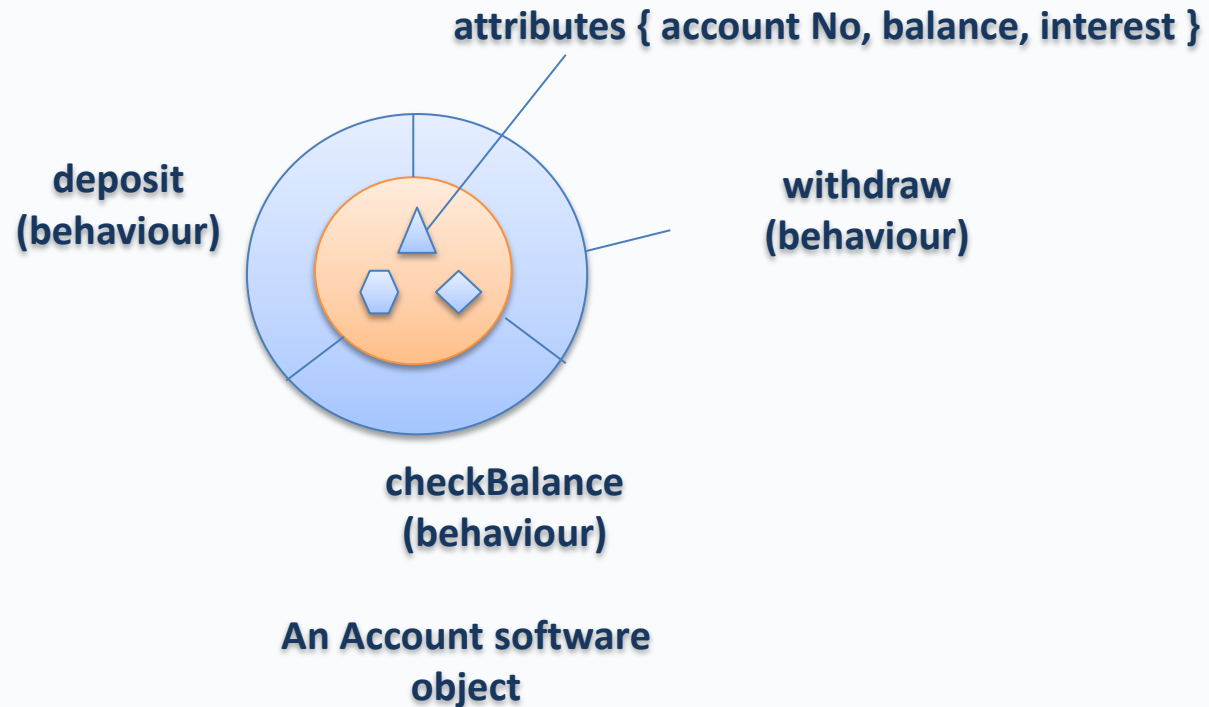
What are objects?



- Objects are real-world entities :
 - Something tangible and visible e.g., your car, phone, apple or pet or
 - Something intangible (you can't touch) e.g., account, time
- Objects have *state* (characteristics or *attributes*) and *behaviour* (*methods* – what the object can do) e.g.,
 - a car has *state* (colour, model, speed, fuel etc.) and *behaviour* (start, change gear, brake, refuel)
 - a dog has *state* (colour, breed, age, gender) and *behaviour* (bark, eat, run)
- Each object encapsulates some state (the currently assigned values for its attributes)

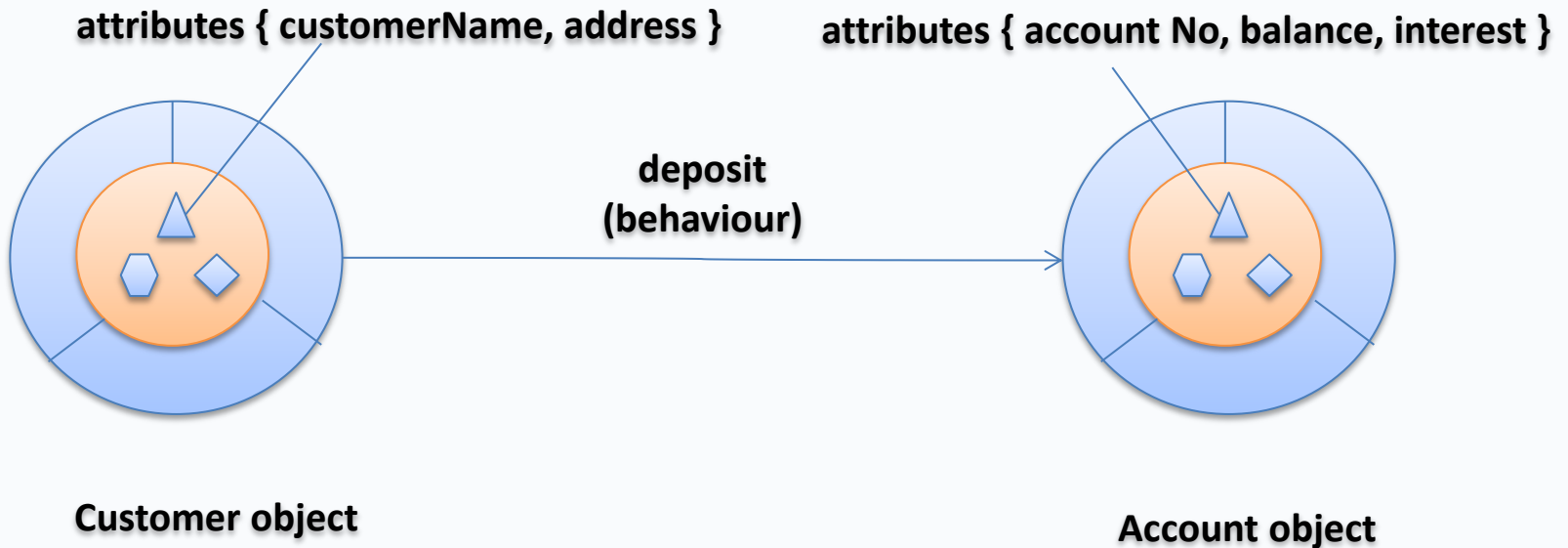
Object Oriented Design

- Identify your domain
- Identify objects



Object collaboration

- Objects interact and communicate by sending *messages* to each other
- If *object A* wants *object B* to perform one of its methods, it sends a message to *B* requesting that behaviour



Object Collaboration

This message is typically made of three parts:

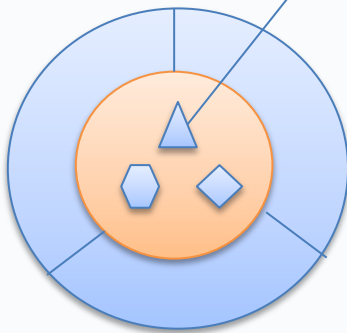
- The *object* to whom the message is addressed (e.g., John's "Account" object)
- The *method* you want to invoke on the object (e.g., deposit())
- Any additional information needed (e.g., amount to be deposited)



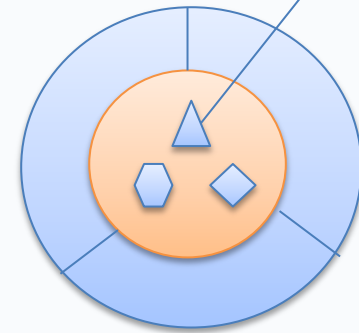
Objects and Classes

- Many objects are of the same “**kind**” but have different identity *e.g., there are many Account objects belonging to different customers, but they all share the same attributes and methods*
- “Logically group” objects that share some common properties and behavior into a **class** (a blue-print for this logical group of objects)

John's account
{ accountNo = 123, balance = 100, interest=5% }



Tom's account
{ accountNo = 567, balance = 600, interest=5.2% }



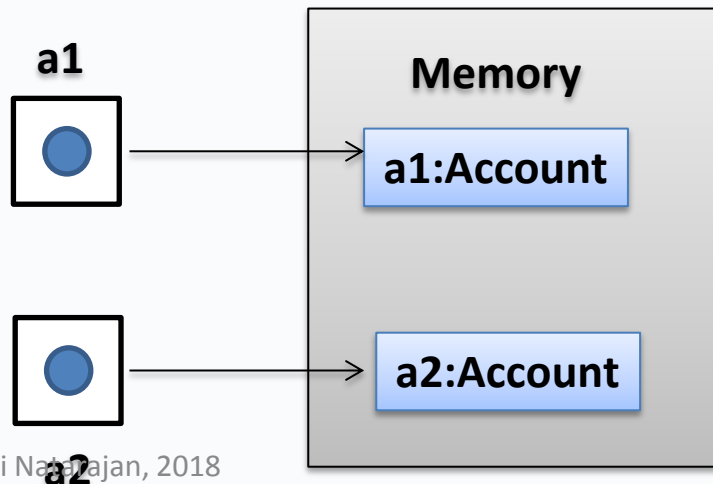
Objects and classes

- Defining a class, does not actually create an object
- An object is **instantiated** from a class and the object is said to be an **instance** of the class
 - An **object instance** is a specific realization of the class
- Two object instances from the same class share the same attributes and methods, but have their own **object identity** and are independent of each other
 - An object has state but a class doesn't
 - Two object instances from the same class share the same attributes and methods, but have their own **object identity** and are independent of each other

Creating object instances

- A class is sometimes referred to as an **object's type**
- An **object instance** is a specific realization of the class
- create an instance of the Account class as follows

```
public class Account {  
    int accountNo;  
    int bsb;  
    float balance;  
  
    public static void main(String[] args) {  
        Account a1 = new Account();  
        Account a2 = new Account();  
    }  
}
```



a1 == a2 -----> True or False?

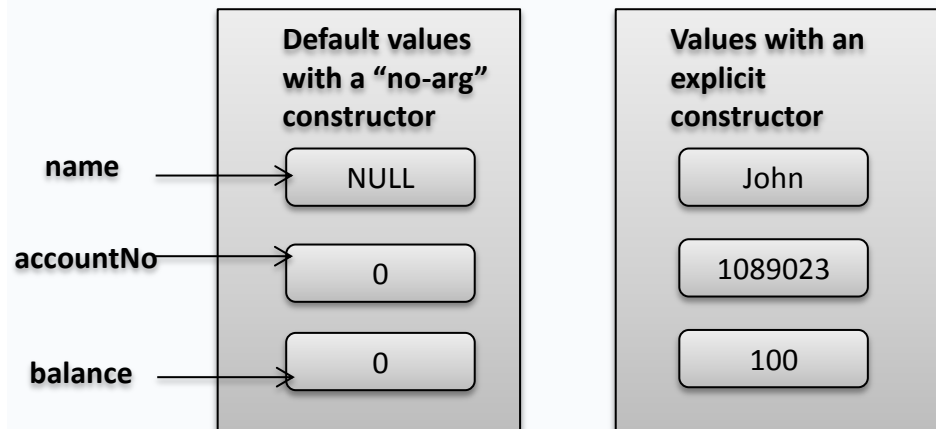
Lecture Demo

- Creating a java class
 - How to create a `class`
 - How to create an `object instance`
 - Object data types
 - Using `constructors` to initialise values of `instance variables`
 - Accessing `instance` methods and variables
 - Use of `static` variables and methods

Constructor & Instance Variables

- A special method that creates an object instance and assigns values (**initialisation**) to the attributes (**instance variables**)
- Constructors eliminate default values
- When you create a class without a constructor, Python automatically creates a default “no-arg” constructor for you

```
public class Account {  
  
    // instance variables  
    String name;  
    int accountNo;  
    float balance;  
  
    // constructor  
    public Account(String aName, int acctNo, float bal) {  
        this.name = aName;  
        this.accountNo = acctNo;  
        this.balance = bal;  
    }  
  
    public static void main(String[] args) {  
        Account a1 = new Account("John", 1089023, 250);  
    }  
}
```

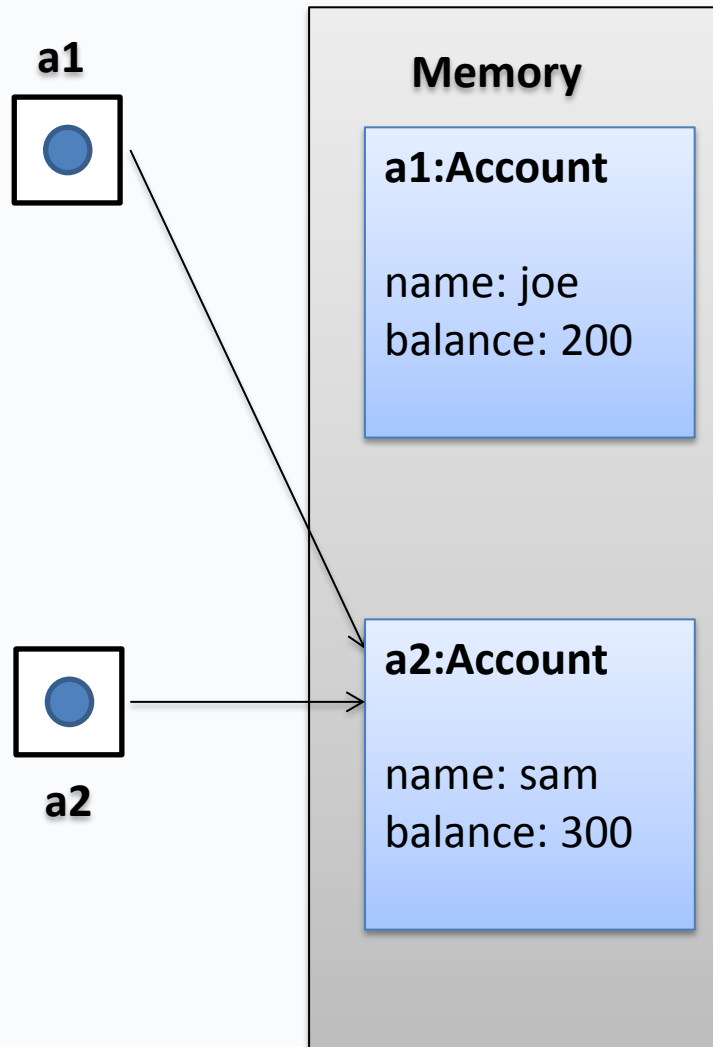


Instance Methods

- Similar to instance variables, methods defined inside a class are known as **instance methods**
- Methods define what an object can do (behaviour)

```
public class Account {  
  
    // instance variables  
    String name;  
    int accountNo;  
    float balance;  
  
    // instance method  
    public void deposit(float amt) {  
        this.balance += amt;  
    }  
  
    // constructor  
    public Account(String aName, int acctNo, float bal) {  
        this.name = aName;  
        this.accountNo = acctNo;  
        this.balance = bal;  
    }  
  
    public static void main(String[] args) {  
        Account a1 = new Account("John", 1089023, 250);  
    }  
}
```

Object References



a1 == a2 -----> True or False?

Consider,

a1 = a2

a1 == a2 -----> True or False?

What is UML?

UML stands for **Unified Modelling Language** (<http://www.uml.org/>)

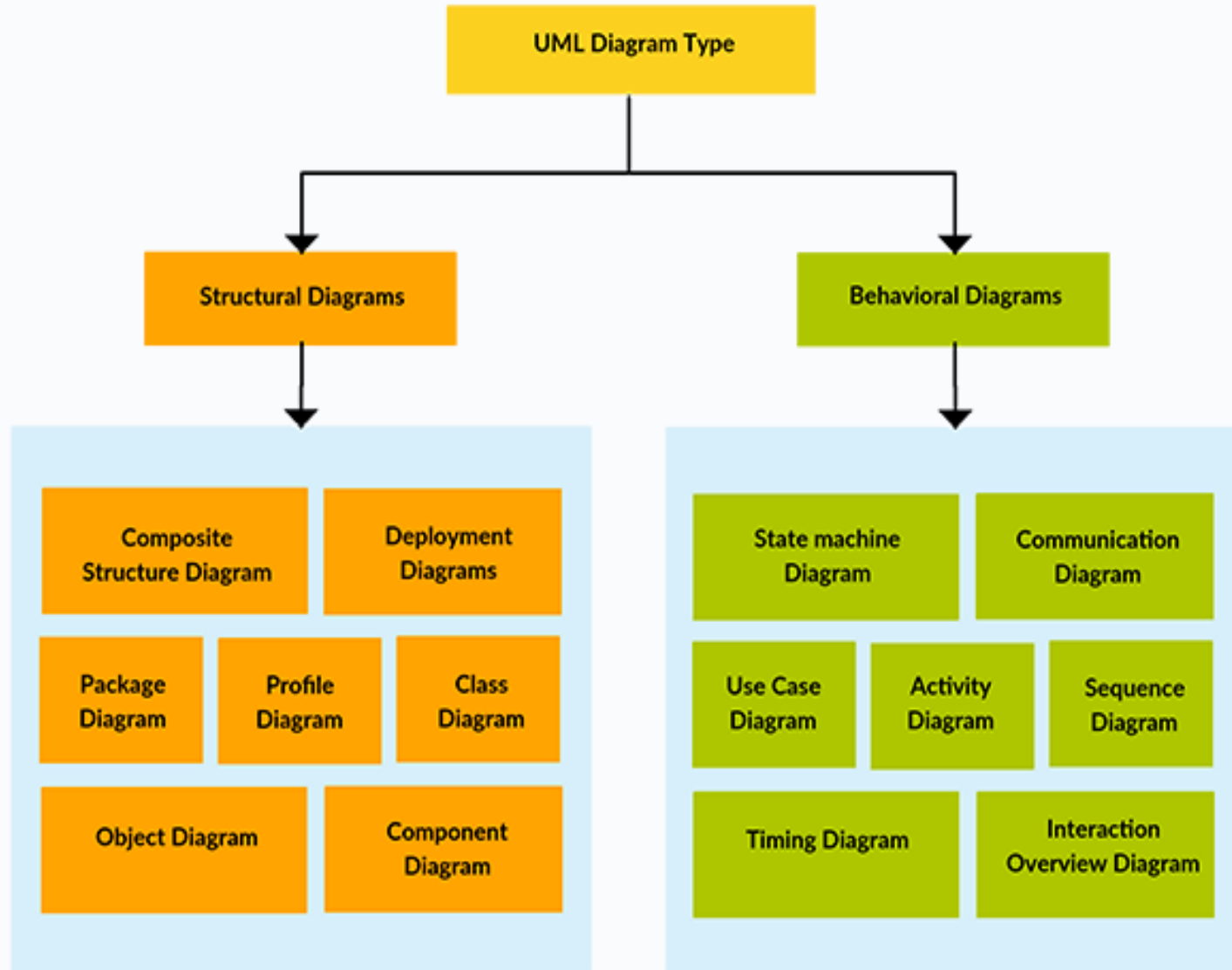
Programming languages not abstract enough for OO design

An open source, graphical language to model software solutions, application structures, system behaviour and business processes

Several uses:

- As a design that communicates aspects of your system
- As a software blue print
- Sometimes, used for auto code-generation

UML Diagram Types



Representing classes in UML

class (class diagram)

Account

-name: String
-balance: float

+getBalance(): float
+getName() : String
+withdraw(float)
+deposit(float)

object instances (object diagram)

a1:Account

name = "John Smith"
balance = 40000

a2:Account

name = "Joe Bloggs"
balance = 50000

Lecture Demo

- Creating a java class
 - Using `access modifiers` in Java
 - Defining `encapsulation`
 - Defining `inheritance`
 - Invoking `super constructors`

Key principles of OO

- Abstraction
- Encapsulation
- Inheritance

Abstraction

- Helps you to focus on the common properties and behaviours of objects
- Good abstraction help us to accurately represent the knowledge we gather about the problem domain (discard anything unimportant or irrelevant)
- What comes to your mind when we think of a “car” ?
Do you create a class for each brand (BMW, Audi, Chevrolet...)
?
 - write *one* class called Car
and *abstract*;
 - focus on the common essential qualities of the object
 - focus on the application domain
- What if a specific brand had a special property or behaviour?
Later on....*inheritance*

Encapsulation

- An OO design concept that emphasises hiding the implementation
- When you drive a car, do you ever worry how a steering-wheel makes a right-turn or a left-turn?
- You are only concerned with the function of the steering wheel
- Encapsulation leads to abstraction

Encapsulating Object State

- Encapsulation of object state implies *hiding* the object's attributes
- An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted
 - **Methods** provide explicit access to the object

e.g. use of *getter* and *setter* methods to access or modify the fields

**Private
Attributes**

Class Name

- attribute1: int
- attribute2: boolean

**Public
Methods**

+ operation_1(): void
+ operation_2(): int
+ operation_3(): boolean

Account

-name: String
-accountNo: int
-balance: float

+getBalance(): float
+getAccountNo():float
+setBalance(float)
+setAccountNo(float)
+deposit(float)

Why is encapsulation important?

1. Encapsulation ensures that an object's state is in a **consistent state**
2. Encapsulation increases **usability**
 - Keeping the data private and exposing the object only through its interface (public methods) provides a clear view of the role of the object and increases usability
 - Clear contract between the invoker and the provider, where the client agrees to invoke an object's method adhering to the method signature and provider guarantees consistent behaviour of the method invoked (if the client invoked the method correctly)
3. Encapsulation **abstracts** the implementation, **reduces the dependencies** so that a change to a class does not cause a rippling effect on the system

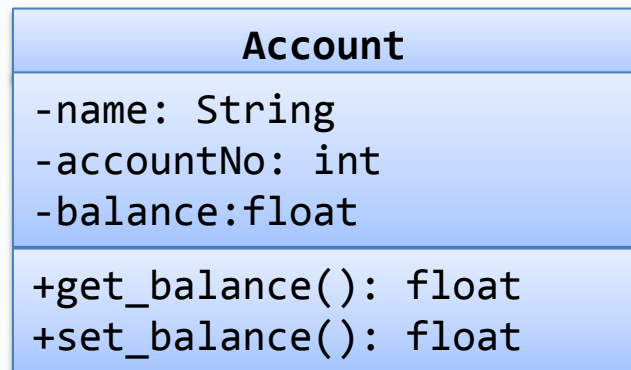
- So far,
 - we have defined classes and object instances
 - objects have attributes and responsibilities
- let us now look at **relationships between objects** e.g.,
 - a dog **is-a** mammal
 - an instructor **teaches** a student
 - a university **enrols** students
- Relationships between objects can be broadly classified as:
 - Inheritance
 - Association

Relationships (1) – Inheritance

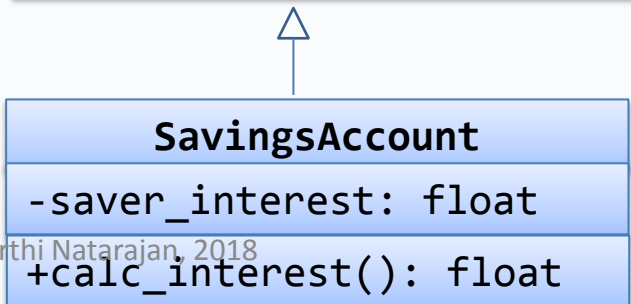
- So far, we have logically grouped objects with common characteristics into a class, but what if these objects had some special features?
 - e.g., if we wanted to store that sports car has spoilers
- Answer is inheritance - models a relationship between classes in which one class represents a more general concept (**parent or base class**) and another a more specialised class (**sub-class**)
- Inheritance models a “is-a” type of relationship e.g.,
 - a savings account is a type of bank account
 - a dog **is-a** type of pet
 - a manager **is-a** type of employee
 - a rectangle **is-a** type of 2D shape

Inheritance

- To implement inheritance, we
 - create a new class (**sub-class**) , that inherits common properties and behaviour from a **base class** (parent-class or super-class)
 - We say the child class *inherits/ is-derived from* the parent class
 - sub-class can **extend** the parent class by defining additional properties and behaviour specific to the inherited group of objects
 - sub-class can **override** methods in the parent class with their own specialised behaviour

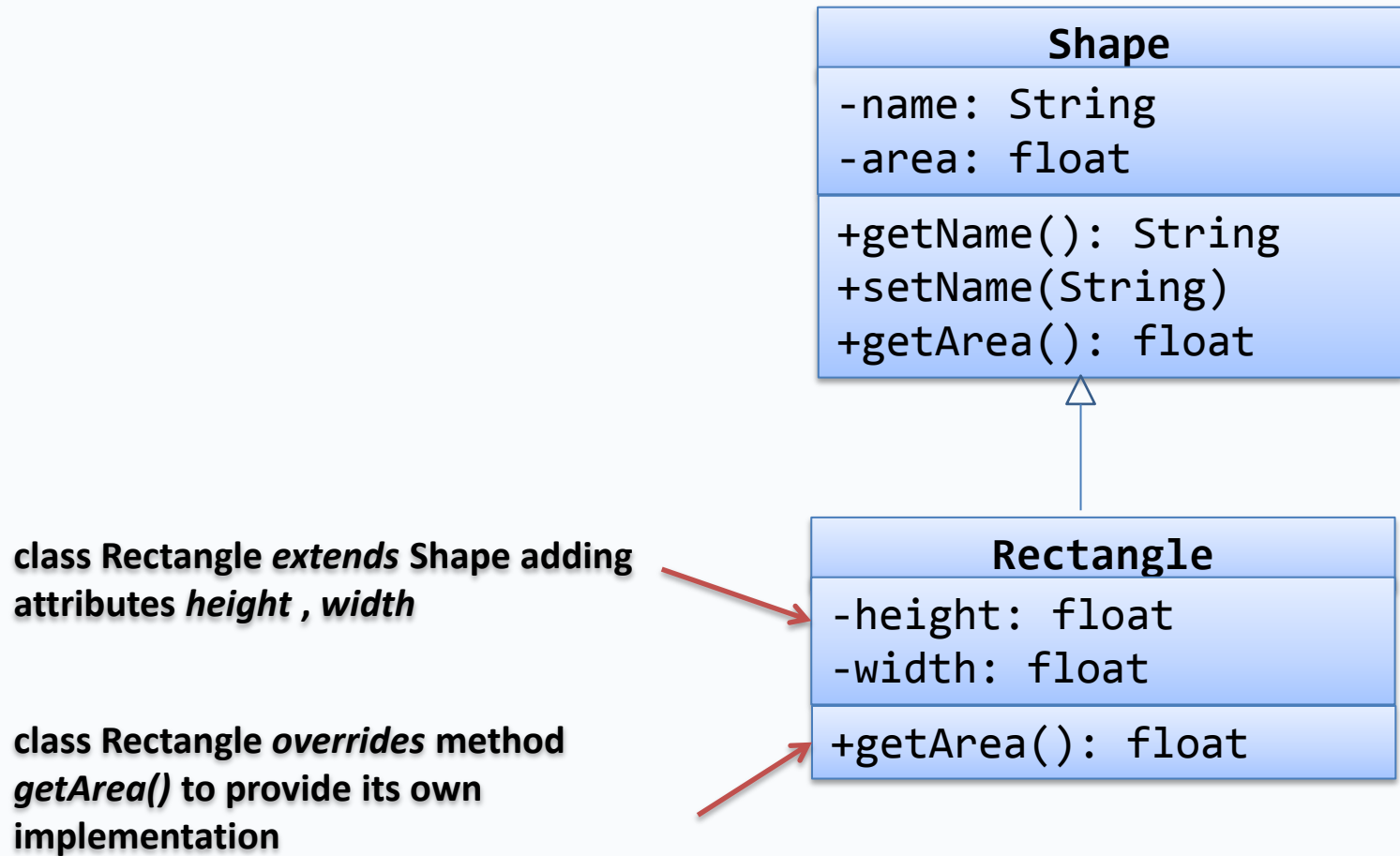


Parent class - Account
class Account defines *name*,
accountNo, *balance*



Child class - SavingsAccount
extends Account class adding its own
attributes and methods e.g.,
saver_interest & *calc_interest()*

Inheritance – another example



Next week...

- Relationships between classes (association , composition, aggregation)
- Creating a domain model applying object-oriented design principles...