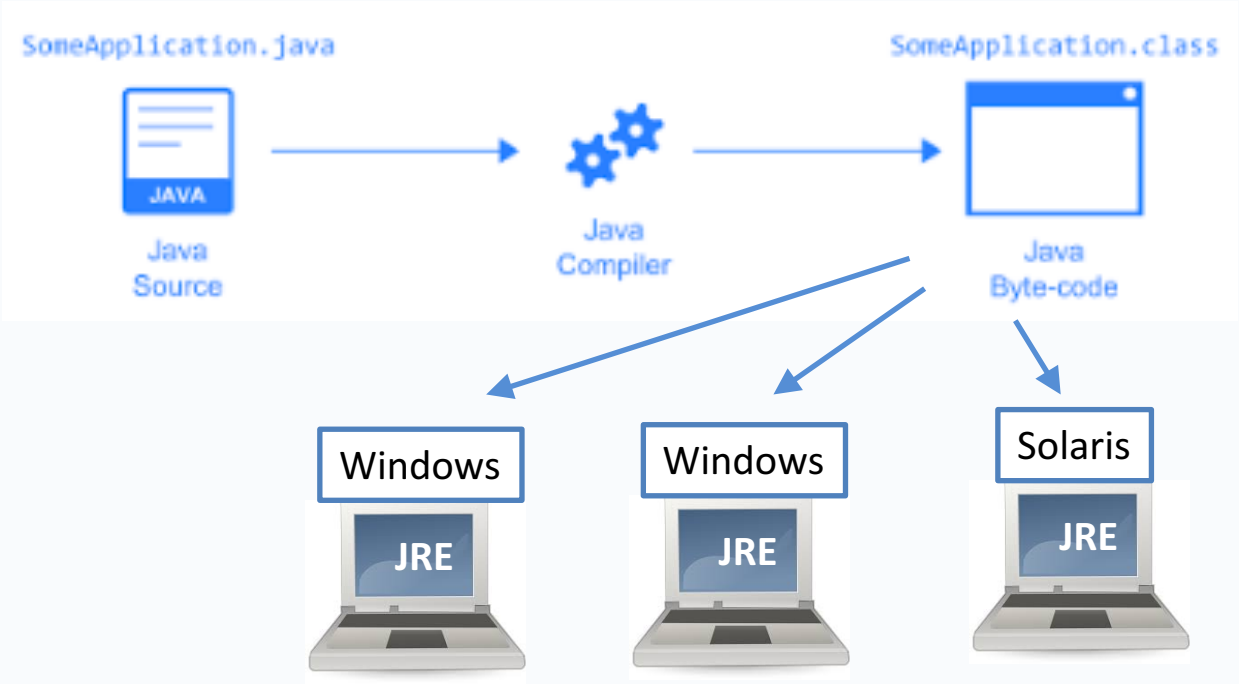
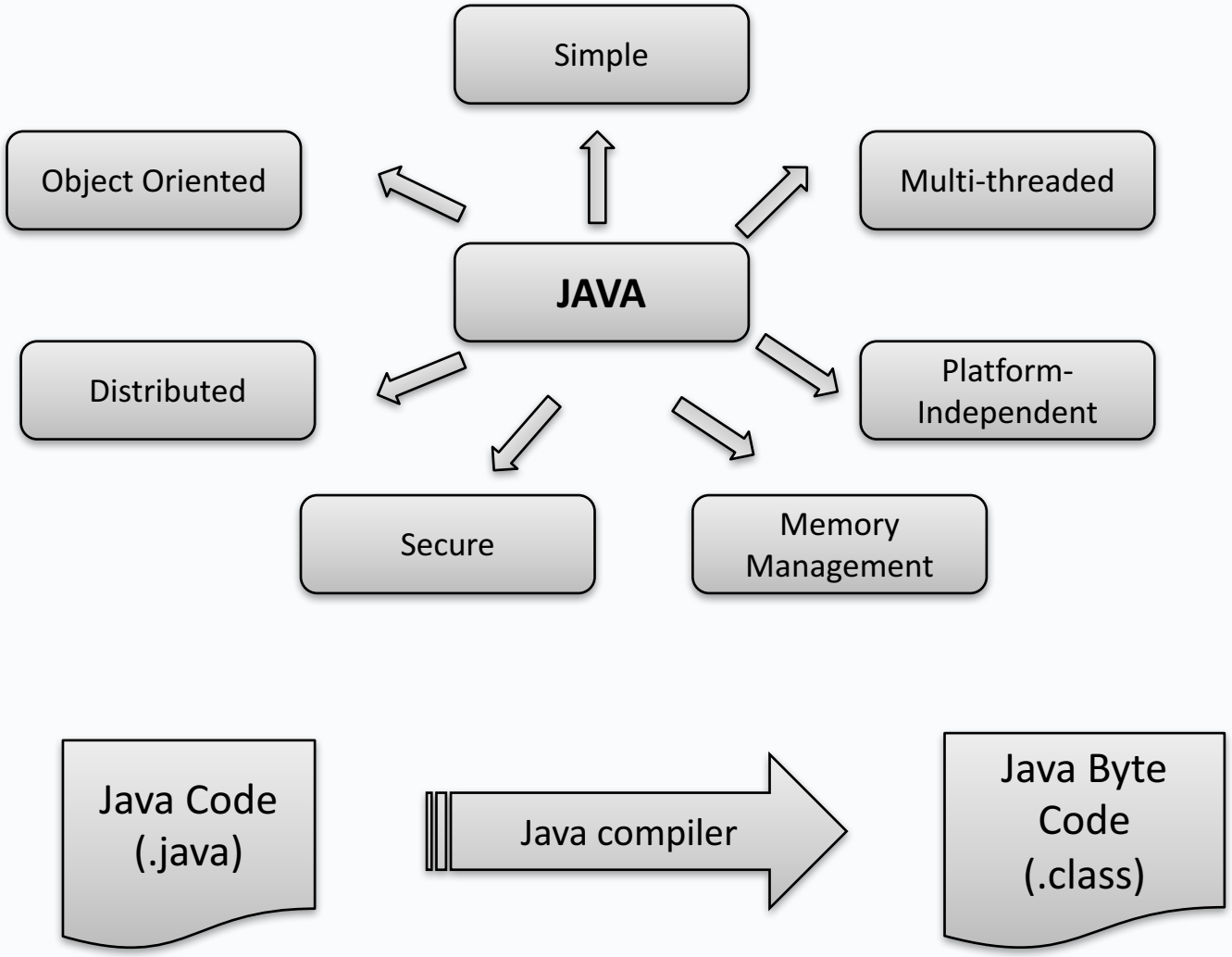
COMP 2511
Object Oriented Design & Programming

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 (Eclipse IDE 2019-03) Downloads:
  https://www.eclipse.org/downloads/
- 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

Coming up …

Thinking Objects

- Classes and Objects
- Abstraction
- Encapsulation
- Inheritance

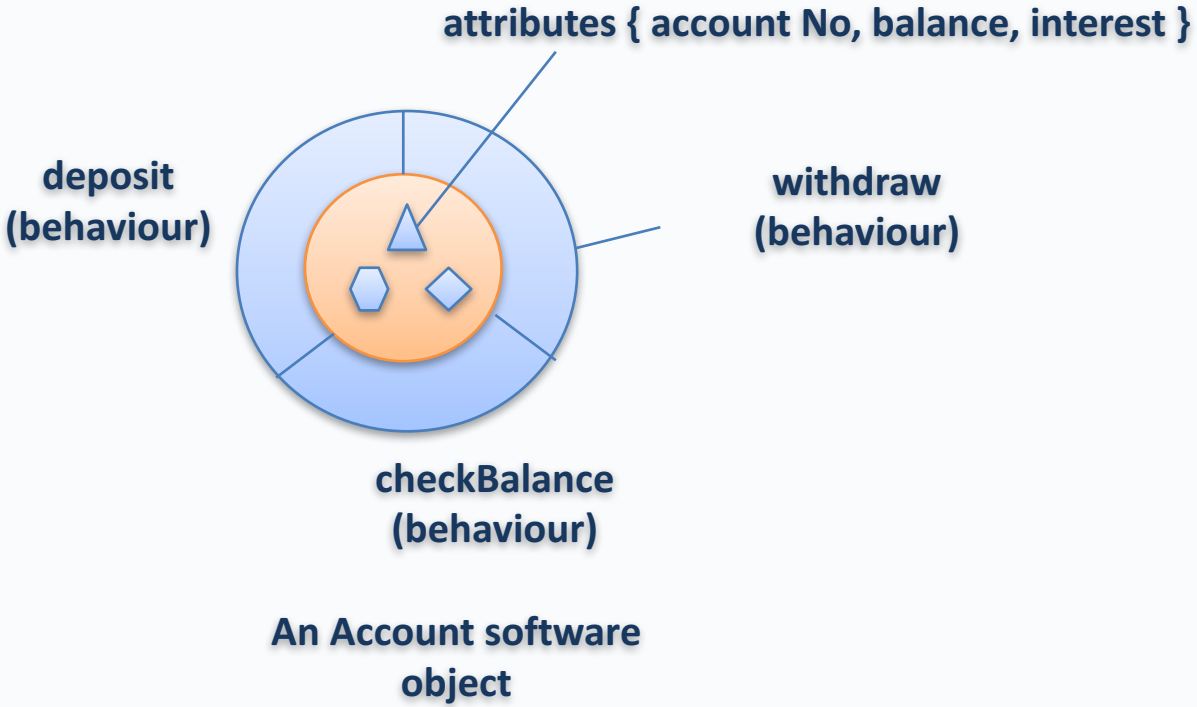
Thinking Objects

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)

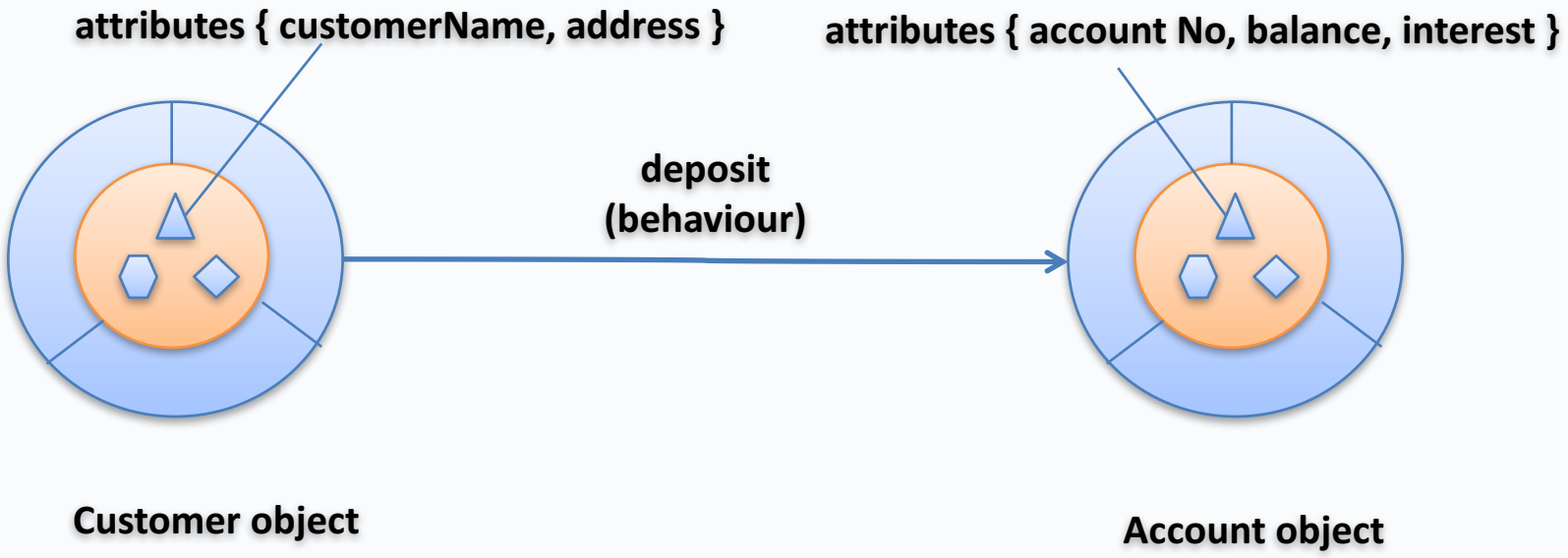
Object Oriented Design

- Identify your domain
- Identify objects



Object collaboration

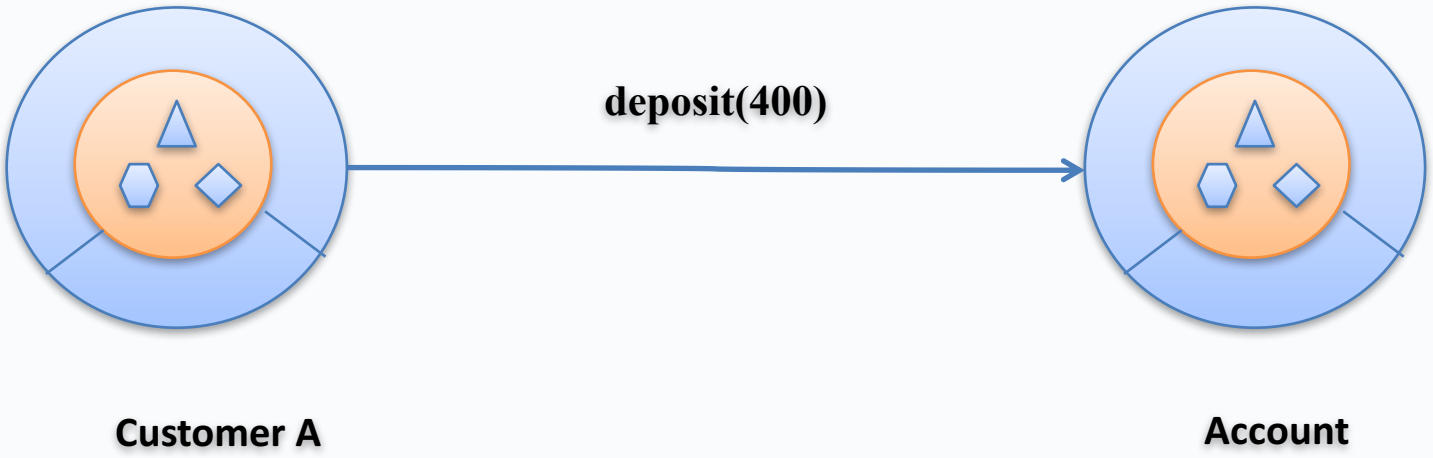
- Objects interact and communicate by sending *messages* to each other
- If *object A* wants *object B* to perform one of its methods, it sends a message to *B* requesting that behaviour



Object Collaboration

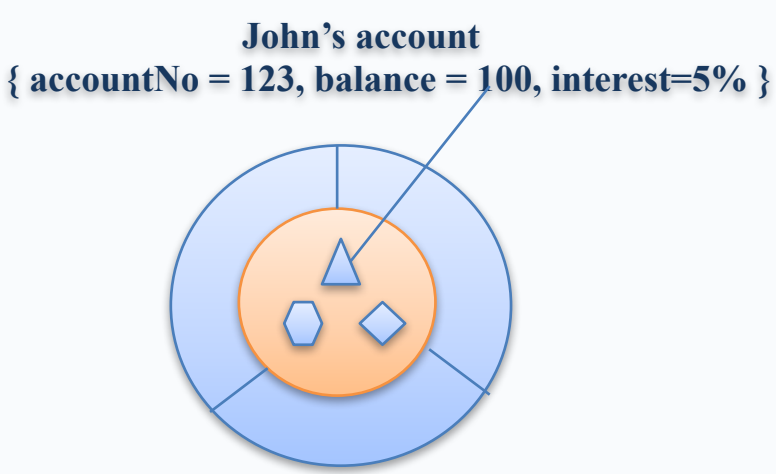
This message is typically made of three parts:

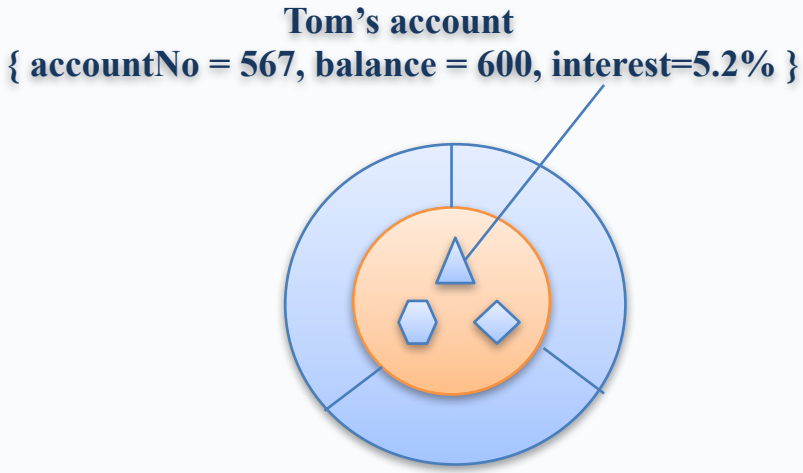
- The *object* to whom the message is addressed (e.g., John's "Account" object)
- The *method* you want to invoke on the object(e.g., deposit())
- Any additional information needed (e.g., amount to be deposited)



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)





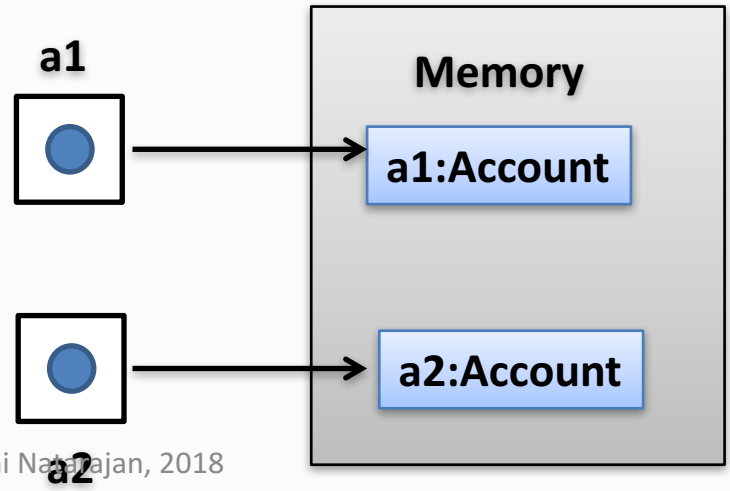
Objects and classes

- 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?

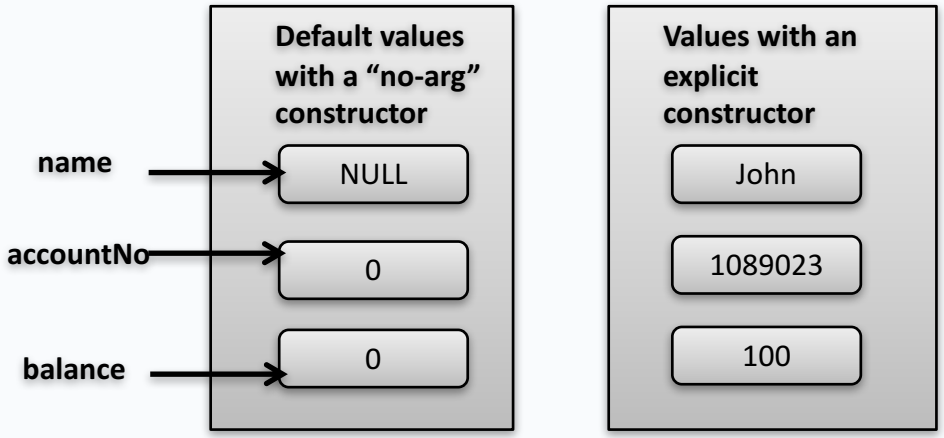
You should know …

- How to 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);
    }
}
```

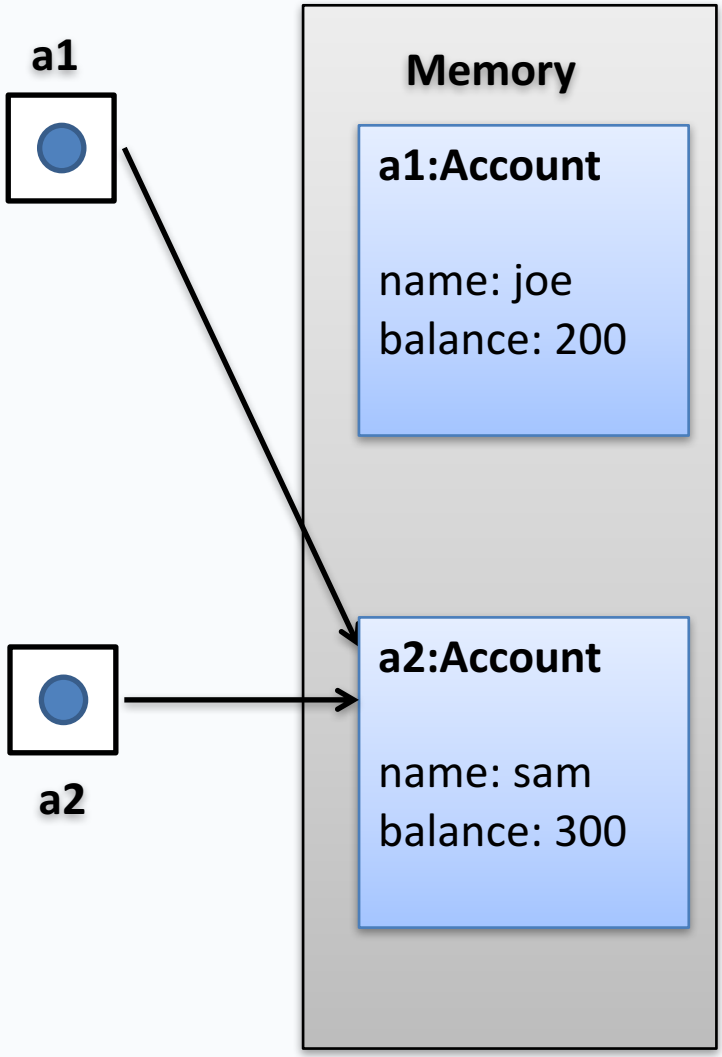


Instance Methods

- 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/)

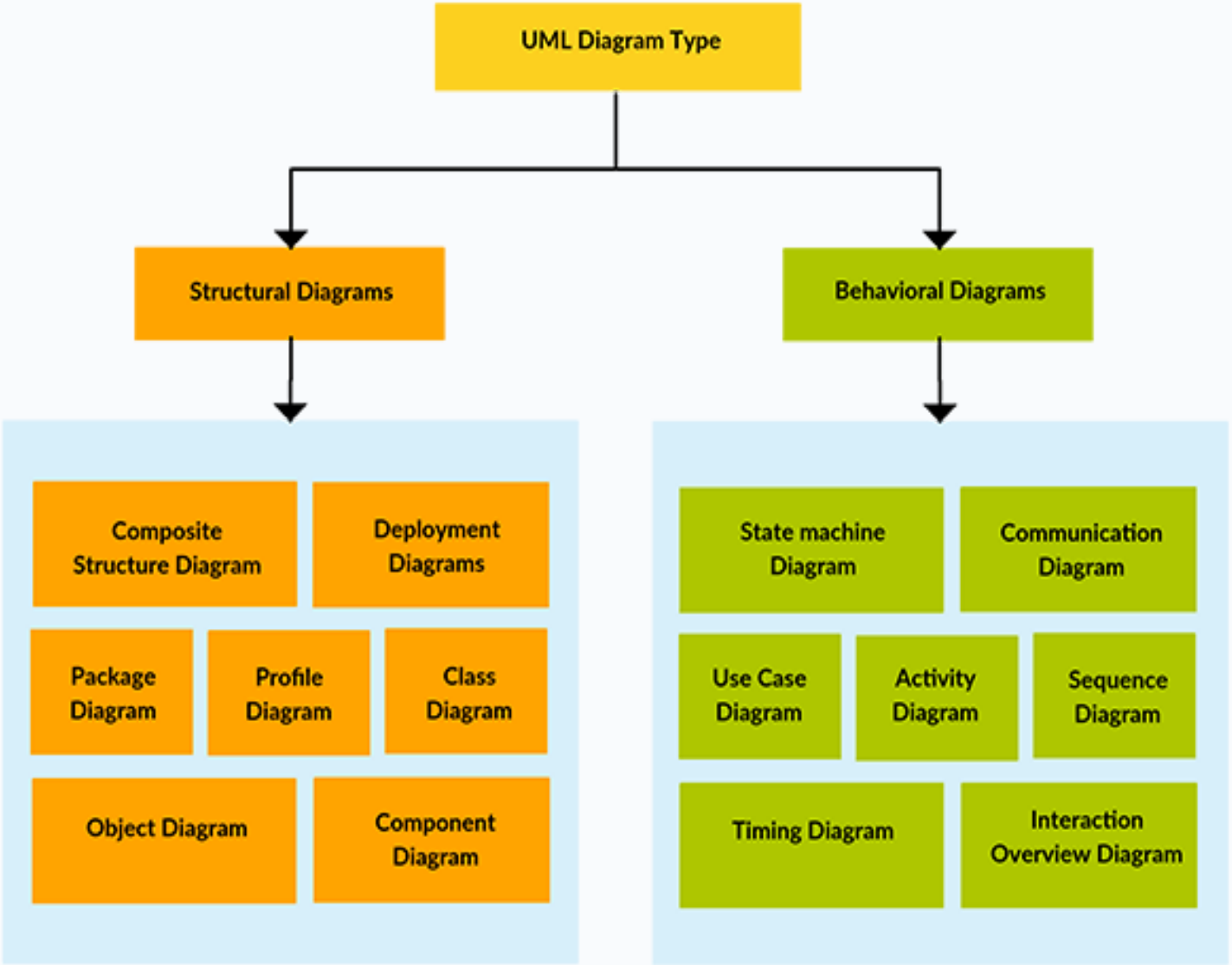
Programming languages not abstract enough for OO design

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

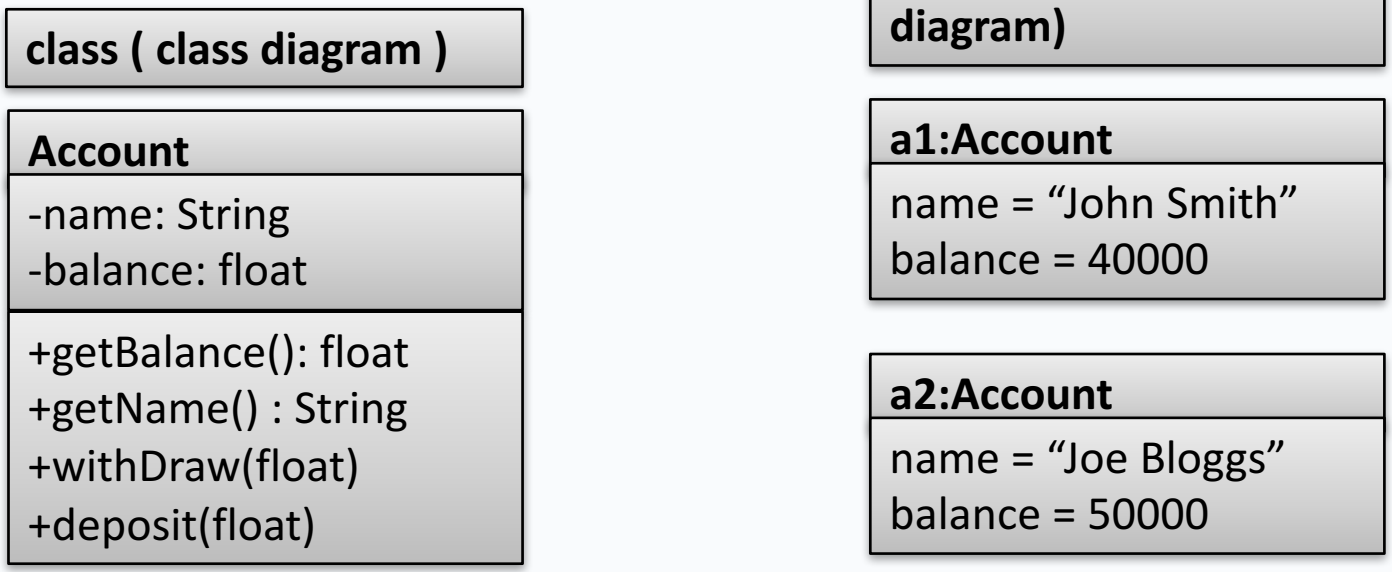
Several uses:

- 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



You should know …

- How to create a java class
  - Use `access modifiers` in Java
  - Define `encapsulation`
  - Define `inheritance`
  - Invoke 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***

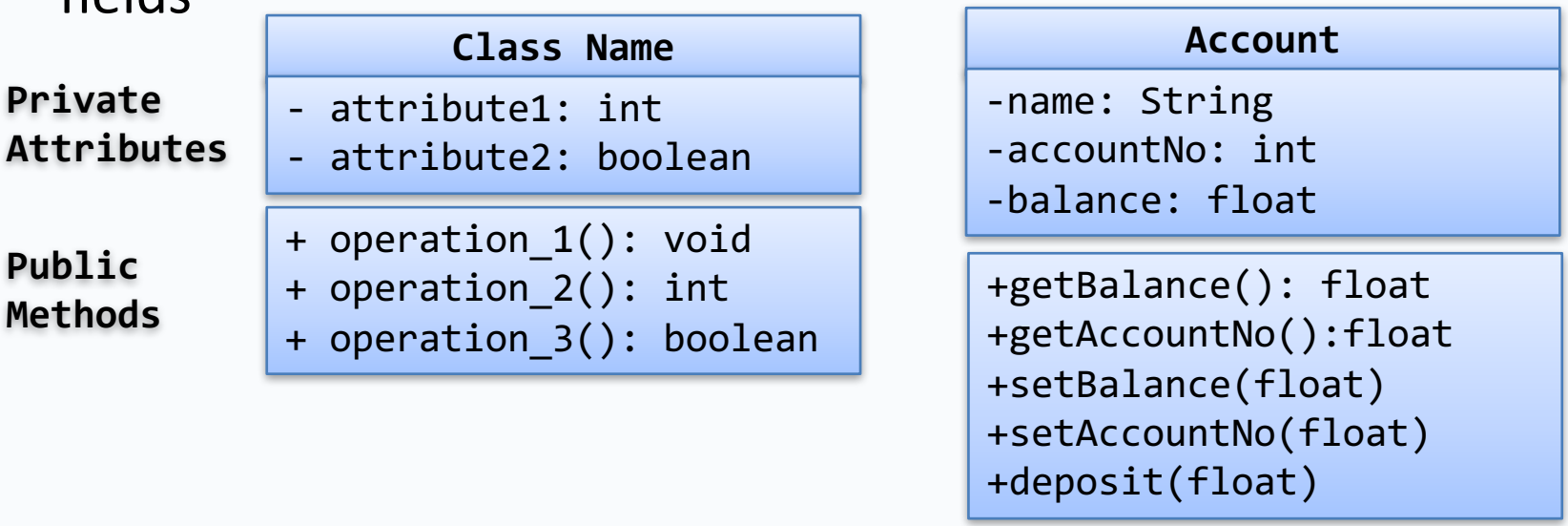
Encapsulation

- 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



UML notation for a Class

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?

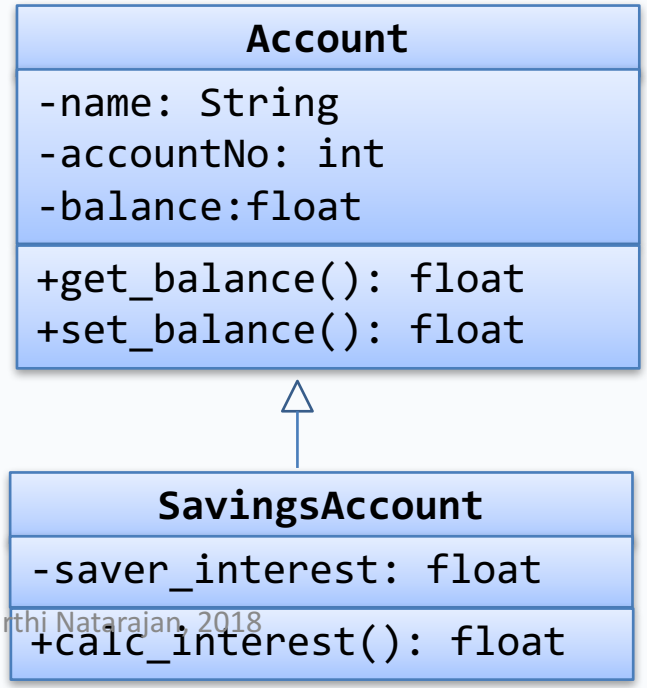
  e.g., if we wanted to store that sports car has spoilers

- 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

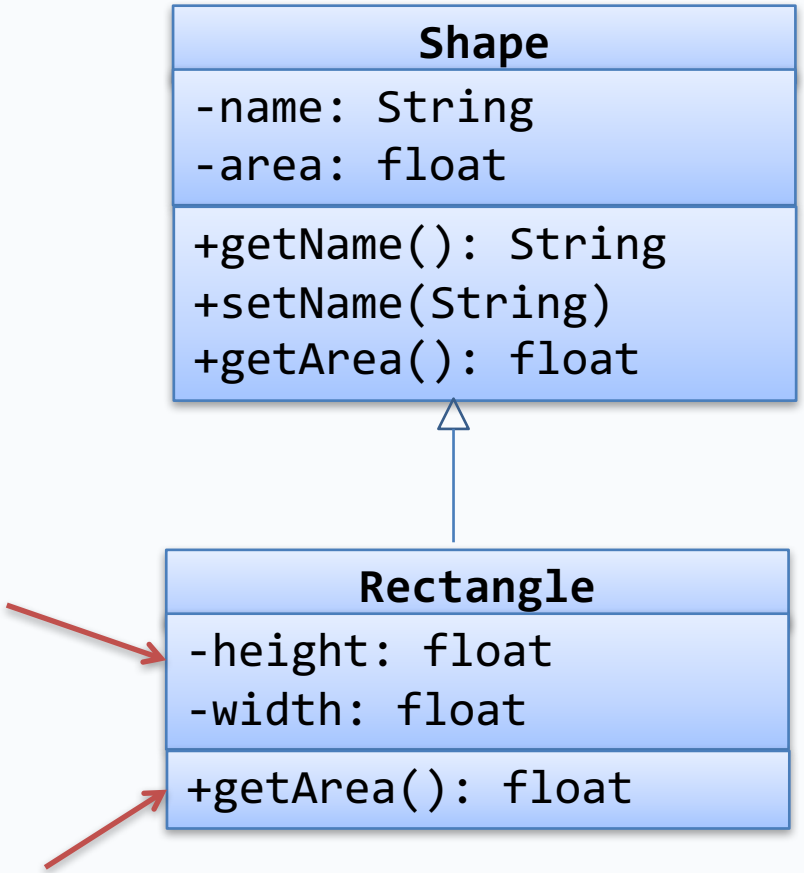
- To implement inheritance, we
  - create a new class (sub-class) , that inherits common properties and behaviour from a base class (parent-class or super-class)
    - We say the child class *inherits/ is-derived from* the parent class
  - sub-class can *extend* the parent class by defining additional 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



class Rectangle *extends* Shape adding attributes *height* , *width*

class Rectangle *overrides* method *getArea()* to provide its own implementation

Next …

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