Classes

Ramaseshan Ramachandran

DATA TYPES

- A datatype describes the representation of an object with limited operations $+, -, /,$ comparison operations
  - Representation + operations
    - $i = i + 1$
    - name = ‘Ram’
    - cost = 12.5

$$\left.\begin{array}{l} i = i + 1 \\ \text{name = ‘Ram’} \\ \text{cost = 12.5} \end{array}\right\} \longrightarrow \text{Composition of type constructor + base operations}$$

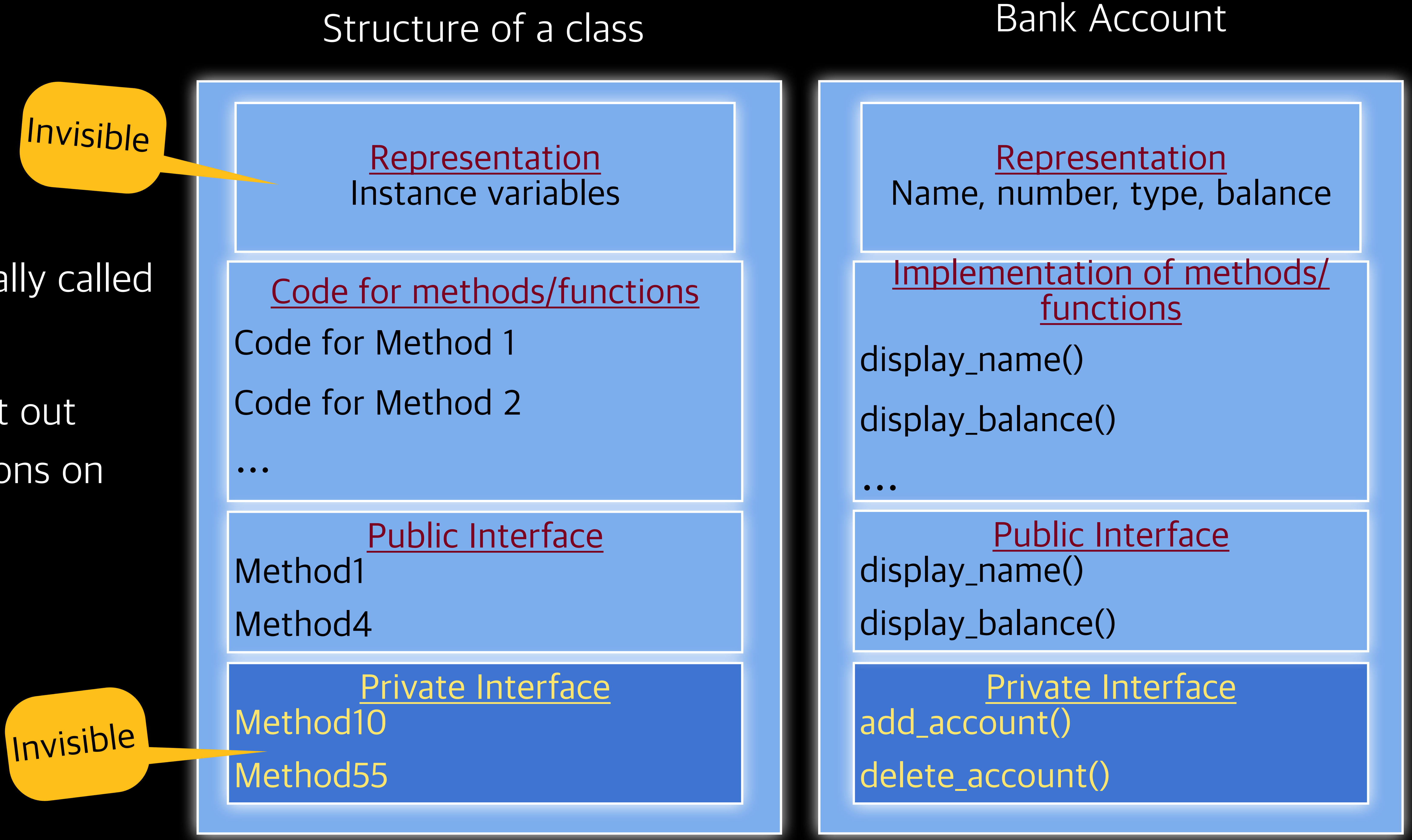
All these operators have associative, commutative, and distributed properties

ABSTRACT DATA TYPE

- ADT provides an additional mechanism – clearly separates the interface and the implementation of the data type
  - Representation – selection of data structure
  - Operation – choosing the algorithm to operae on
- Programming languages like Python provide language defined ADTs –
  - list, dictionary/map, vector,…

CUSTOM DEFINED ADTS - CLASSES

- User defined ADTs are usually called as classes
- Interface to classes brought out through associated operations on data types



Most modern languages allow the benefit of ADT to every custom object

ADVANTAGES OF CUSTOM CLASSES

Encapsulation – Enforces the access and the update of objects through public interfaces

In C or Pascal, this is not strictly enforced
Can be manipulated by any function/procedure
with in the scope of the data type

QUESTION?

What happens to all methods when the internal data structure changes?

Example – Data type representing balance changes to a class object from float

What happens to the methods that access/modify it?

All the methods/functions that access and modify the data structure must be modified

It is possible to safely modify the implementation without affecting the public/private interface

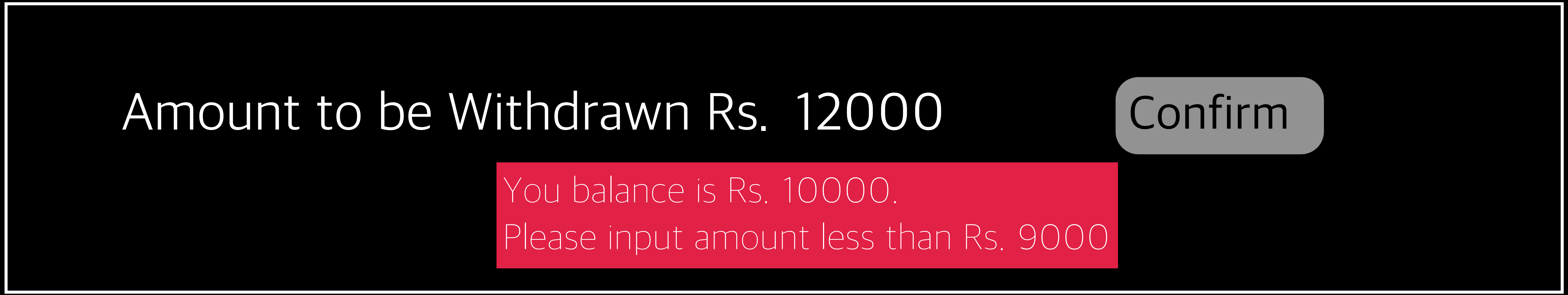
First priority while modifying – almost care is taken to retain the interface

How about adding a new public method/function?

MESSAGE PARADIGM

Objects make up the computational paradigm

Every object is capable to processing requests/messages



Amount to be Withdrawn Rs. 12000

You balance is Rs. 10000.
Please input amount less than Rs. 9000

Message 1 – Confirm withdrawal amount

Message 2 – Check balance

Message 3 – If withdrawn > balance

Send message to button object– change state to disabled

Send message to ErrorMonitor – display error appropriate error message

MESSAGE PARADIGM

Different from the traditional Model

It consists of collection of objects that uses a set of message to exchange/modify data

Objects communicate via messages

Objects listen to messages in the channel

If appropriate message is seen, takes action accordingly

Objects in this model have internal states

External messages cause objects internal states to change

Object in this paradigm is always in an active state

Behavioural specifications are added to the traditional objects

MODELLING THE REAL WORLD

Conventional programming is Procedure-based

Data objects are central to the Object-oriented programming

Objects have well structured behaviour

Encapsulated algorithms implement the behaviour of the objects

Implementation of the behaviour of the objects is quite natural unlike procedure-based decomposition

Objects provide direct representation of the real world

Clarity, robustness, debugging and maintenance are improved

AUTONOMY

Autonomy refers to the degree of independence an object has from external control

OOP allows objects to behave in an autonomous way

Each object has specific authority and responsibilities in handling the member variables

Objects can make their own decisions based on internal state and well-defined rules, potentially reacting to events or stimuli without direct instructions

Techniques

Event-driven programming

Objects can register for events and react autonomously when they occur, reducing dependence on direct calls.

Callbacks

Objects can pass functions as arguments, empowering them to trigger specific actions within other objects

Asynchronous Programming

Objects can initiate operations without waiting for responses, allowing them to proceed concurrently and independently.

ROBUST CODE BASE

- Helps in building robust code base
  - Stubbing
  - Isolation of Errors
  - Exception-checking and error-handling
- Reusable

GRAMMAR SPECIFICATION

```
class_def:
    | decorators class_def_raw
    | class_def_raw

decorators: ('@' named_expression NEWLINE )+

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ]
':' block
```

```
type_params: '[' type_param_seq  ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound]
    | '*' NAME ':' expression
    | '*' NAME
    | '**' NAME ':' expression
    | '**' NAME
```

SUMMARY

- Allows better conceptualisation and modelling of real-world scenarios
- Enhances robustness
- Enhances performance
- Modularization – localises the operational aspects of the attributes
- Separates specifications from implementation
- Allows extensibility – easy to develop and maintain