COMP1007

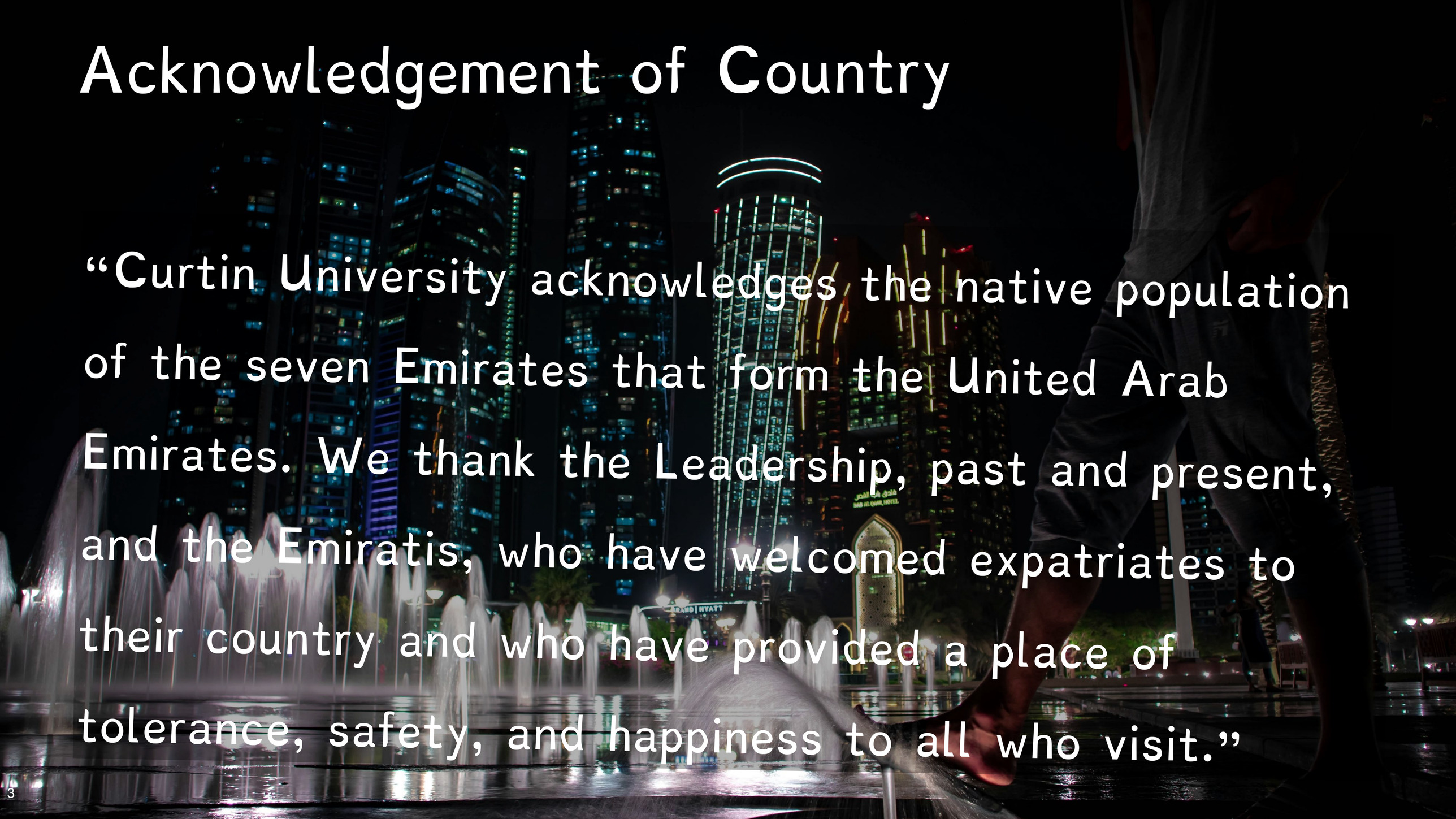
Implementing Object Orientation

Dr David A. McMeekin

Acknowledgement of Country

“I acknowledge the Wadjuk people of the Noongar nation on which Curtin University’s Boorloo Campus sits, and the Wangkatja people where the Karlkurla Campus sits, as the traditional custodians of the lands and waterways, and pay my respect to elders past, present and emerging.”

Acknowledgement of Country



“Curtin University acknowledges the native population of the seven Emirates that form the United Arab Emirates. We thank the Leadership, past and present, and the Emiratis, who have welcomed expatriates to their country and who have provided a place of tolerance, safety, and happiness to all who visit.”

COMP1007 - Unit Learning Outcomes

- Identify appropriate primitive data types required for the translation of pseudocode algorithms into Java;
- Design in pseudocode simple classes and implement them in Java in a Linux command-line environment;
- Design in pseudocode and implement in Java structured procedural algorithms in a Linux command-line environment;
- Apply design and programming skills to implement known algorithms in real world applications; and
- Reflect on design choices and communicate design and design decisions in a manner appropriate to the audience.

Outline

- Revision;
- Constructors & Pseudocode Class in Java;
- OOP vs Non OOP;
- Inheritance;
- More Methods;
- More on Classes;

Revision

How We See the Real World

- We see the world as objects;
- Our world is made up of objects;
- Look around, objects are everywhere.

Objects have Attributes

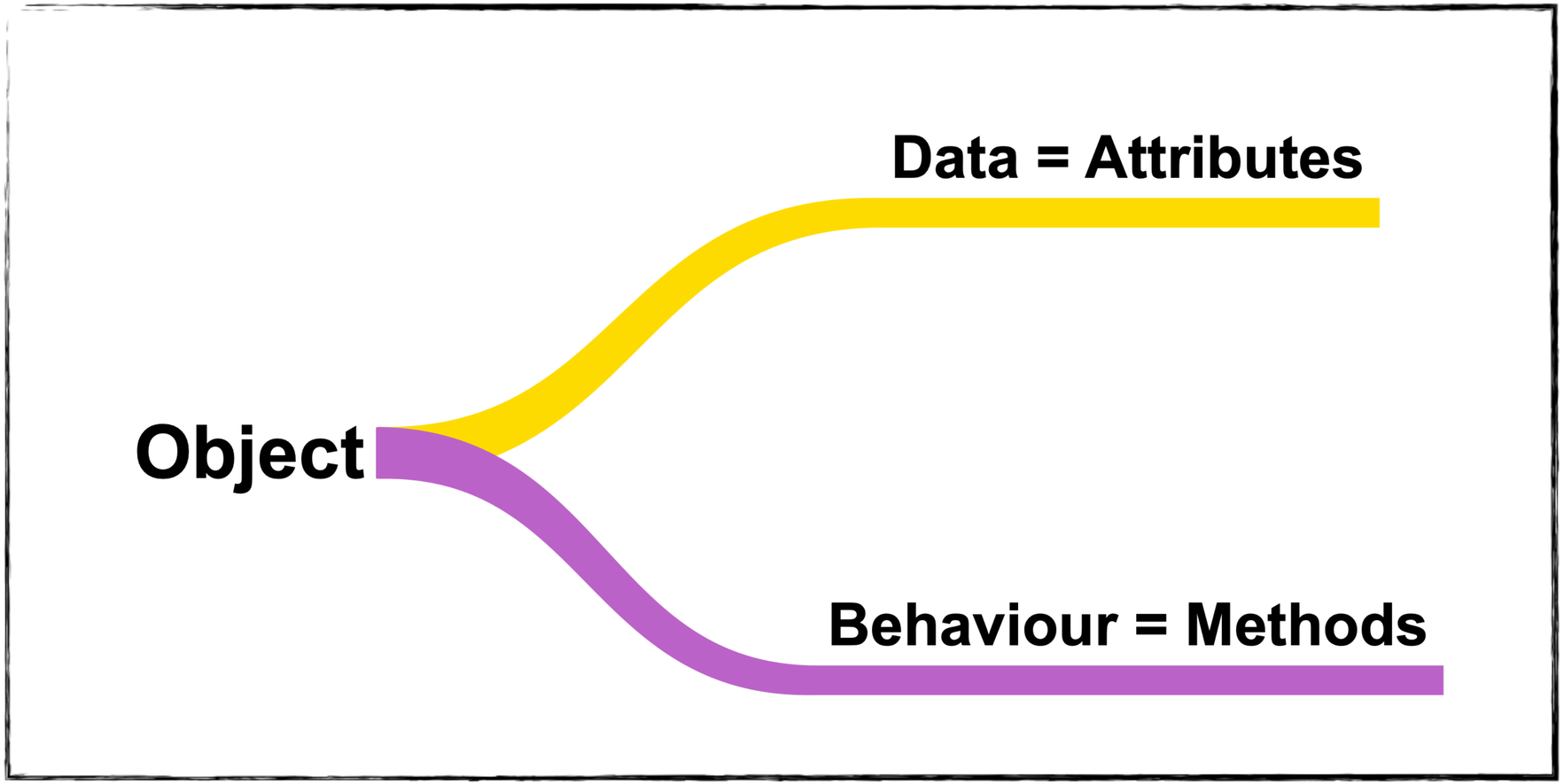
- Each object has attributes;
- A pen has a colour;
- The colour has a value: purple;
- A book has a title;
- The title has a value: “Java Programming”.

Attributes, Values and State

- Attributes: '...the characteristics that define an object...' (Farrell, 2018).
- Values: generally the data in the Attributes;
- State: the set of all the values in the Attributes.

The Basics of Object Oriented Programming

- OOP: about what to manipulate, not how to manipulate it;
- Object Oriented Programming Simplified:



Blueprints/Plans

Created from the Blueprint/Plan

- Built from the plan on the previous slide:

Creating Objects

- Creating something requires a plan or blueprint;
- In OOP, the plan/blueprint is called a class;
- To create an object a class is first needed;
- An object is created from a class.

Constructors & Pseudocode

Creating an Object

- To create a house from the plans, construction occurs;
- The plans have rooms, bathrooms, kitchen, garage...;
- Customise it from the plan with different values;
- Copy it directly from another one;
- Can build straight from plan, default everything; or customise.

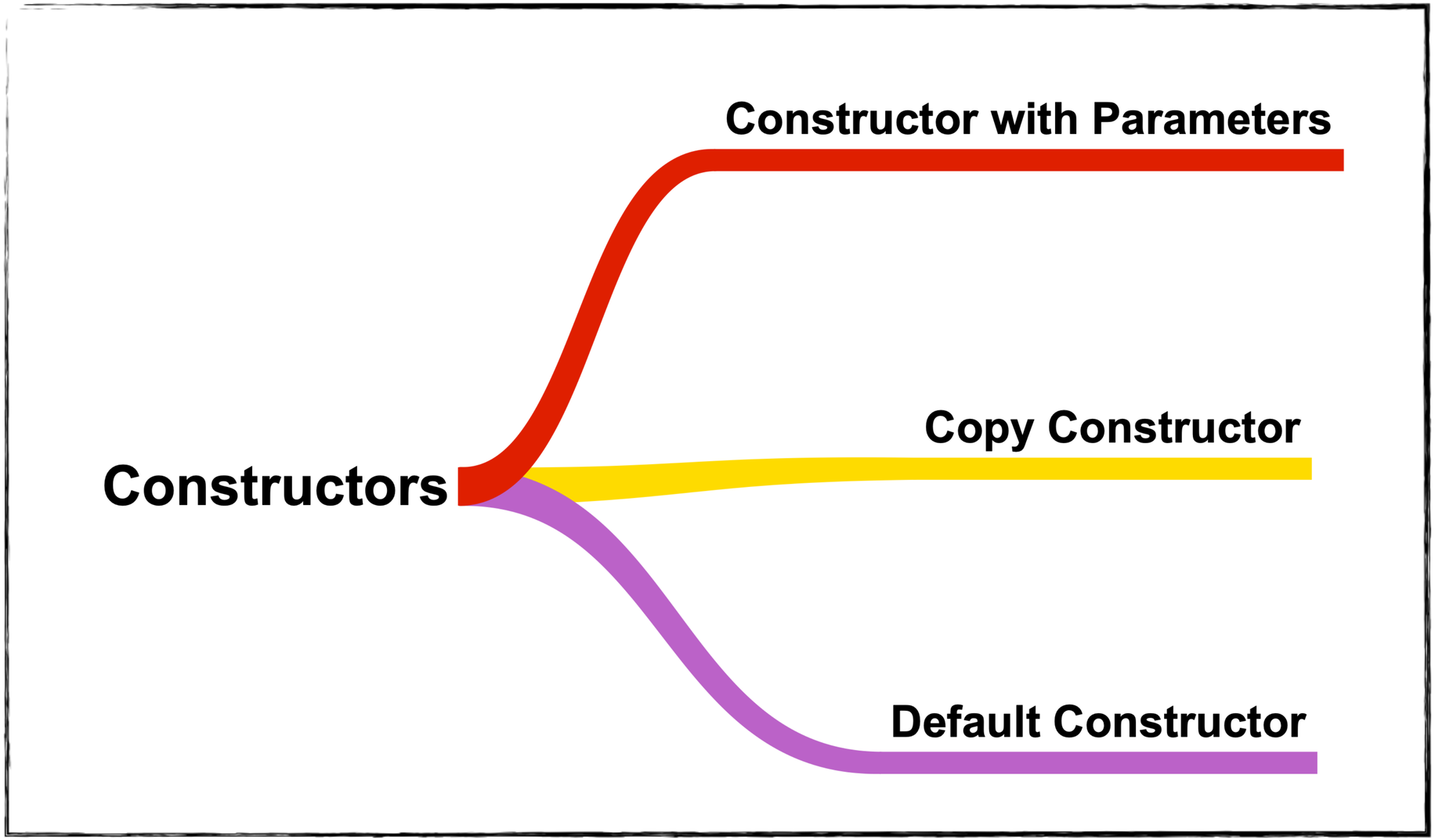
Constructing Objects in OOP

- In OOP objects are constructed;
- Objects are constructed from the Class;
- A Class is instantiated to create an object;
- An object is an instance of the Class.

Constructors

- Methods used to create an object (create an instance of a class);
- Ensure object's are created with valid values;
- Ensure the object is created in a valid state;
- Usually several constructors available;
- Differences between them:
  - what they IMPORT;
  - the information used to initialise the object.

The Three Constructors



The Constructors

- The three main categories of constructors:

  1. Constructor with Parameters: creates an object with IMPORTed data to initialise class fields;

  2. Copy Constructor: creates an object by IMPORTing an object of the same class and initialises class fields to the same values of the IMPORTed object.

  3. Default Constructor: creates an object with default values.

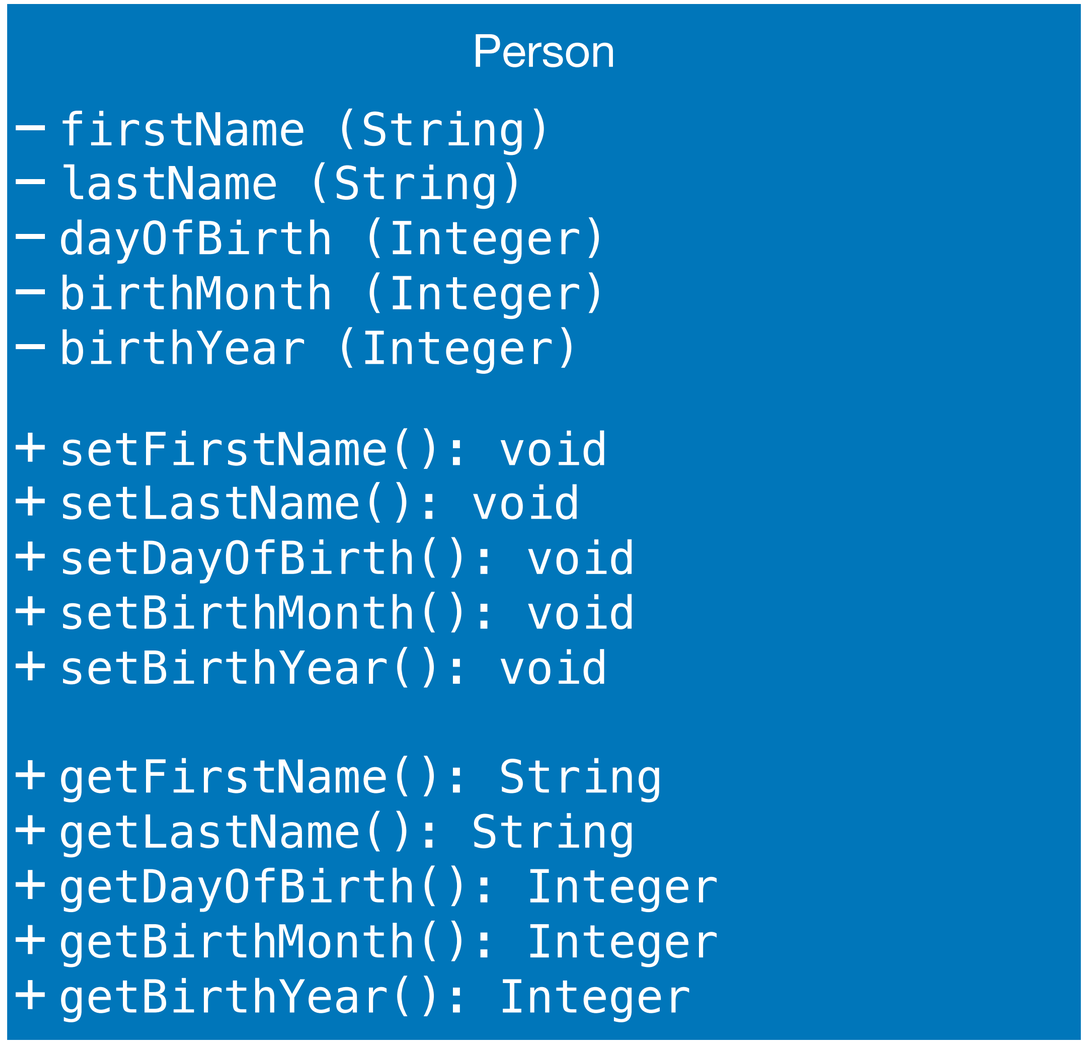
Constructor with Parameters

- **IMPORT** values used to initialise class fields;
- **IMPORT** values should be validated (if possible):
  - If valid: use to initialise class fields;
  - If invalid: fail, or throw an exception (error).
- **IMPORT** information may directly reflect class fields:
  - e.g. **DateClass->pDay**, **pMonth**, **pYear** to initialise class fields day, month and year.

An Example – Person

- Our person has the following attributes:
  - a first name;
  - a family name;
  - a day of birth (dd);
  - a birth month (mm); and
  - a birth year (yyyy).
- A person has other attributes, but this is enough for now.

Person Class UML Diagram



Constructor with Parameters - Pseudocode

```
CONSTRUCTOR with PARAMETERS
IMPORT: pFirstName(String), pFamilyName(String),
        pDayOfBirth (Integer), pBirthMonth (Integer),
        pBirthYear (Integer)
EXPORT:none //Constructors never export
ASSERTION: Created object with imported values
ALGORITHM:
    firstName = pFirstName
    familyName = pFamilyName
    dayOfBirth = pDayOfBirth
    birthMonth = pBirthMonth
    birthYear = pBirthYear
```

- Realistically, the day, month and year values should be validated.
- Why? What could possibly be wrong with them?

Copy Constructor

- IMPORT's an object of the same class;
- Extracts the state information from the IMPORTed object;
- State information: the values in the class fields;
- Uses these values to initialise the new object's class fields;
- The new object has an equivalent state to the IMPORTed object;
- We have copied the IMPORTed object.

Copy Constructor – Pseudocode

```
COPY CONSTRUCTOR
IMPORT: pPerson (Person)
EXPORT:none //Constructors never export
ASSERTION: Created a Copy of the IMPORTed object
ALGORITHM:
    firstName  = pPerson.getFirstName()
    familyName = pPerson.getFamilyName()
    dayOfBirth = pPerson.getDayOfBirth()
    birthMonth = pPerson.getBirthMonth()
    birthYear  = pPerson.getBirthYear()
```

- Each class field in the new object has the value of the equivalent class field from the IMPORTed object;
- Why do IMPORTed objects' class field values NOT need validating?

Default Constructor

- No IMPORT;
- Initialises object state to default values;
- Default values are valid values;
- We decide the default valid values;
- Not an overly useful constructor.

Default Constructor - Pseudocode

```
DEFAULT CONSTRUCTOR
IMPORT:none //Default constructors don't import
EXPORT:none //Constructors never export
ASSERTION: Creates an object with the default values
ALGORITHM:
    firstName = Mohamed
    familyName = Sallah
    dayOfBirth = 15
    birthMonth = 6
    birthYear = 1992
```

- Every Person created using this constructor has these values.

Creating an Object

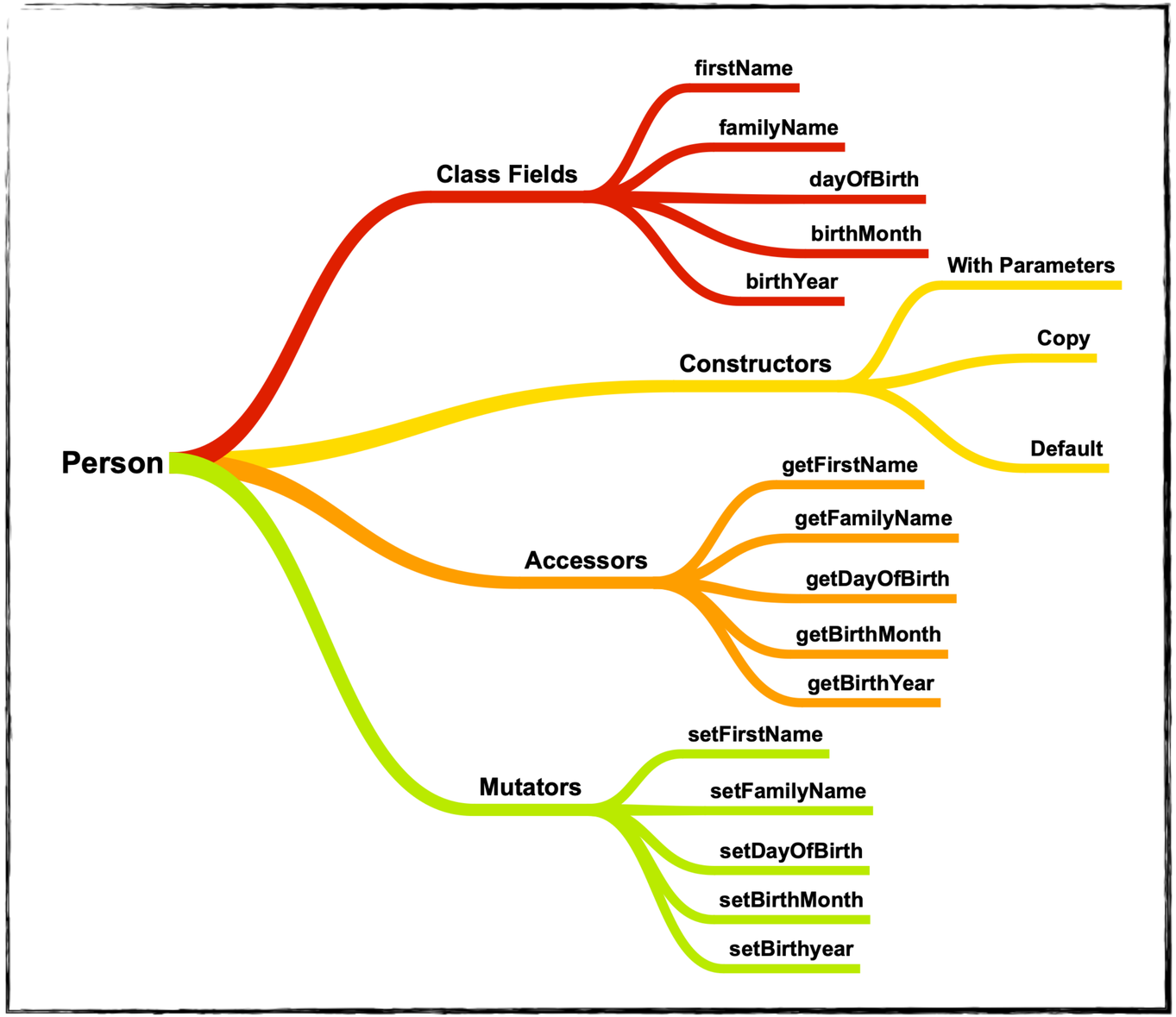
- A person can now be constructed in three ways:
  1. with values defined on construction (constructor with parameters);
  2. a copy of another person object (copy constructor);
  3. a default person.

Java OO

Java and Class Files

- Each class is created in its own `.java` file, same name as the class;
- The Java compiler creates a `.class` file for each `.java` file;
- Many classes that you design and implement in Java could be useful across a number of different applications;
- These general purpose classes should be grouped together in a library which are accessible to all of your applications;
- In Java we call such a library a **package**:
  - You have been using `import java.util.*;` for weeks

Person


Person
Class Fields
firstName
familyName
dayOfBirth
birthMonth
birthYear
Constructors
With Parameters
Copy
Default
Accessors
getFirstName
getFamilyName
getDayOfBirth
getBirthMonth
getBirthYear
Mutators
setFirstName
setFamilyName
setDayOfBirth
setBirthMonth
setBirthyear

Class Fields

- a variable accessible through the entire class (i.e., It is global to the class);
- Good class design means class fields are **NOT** visible outside the object;
  - In Java, this means class fields are declared private;
  - This follows the information hiding principle.
- Constructors initialise class fields;
- Accessors refer to the values stored in class fields; and
- Mutators modify class fields' values.

Example Java Class: Class Field Declarations

```
/*************************************************************
 * Author: David McMeekin                                    *
 * Date: 15 Apr 2021                                         *
 * Purpose: A Person for use in PDI                          *
 *************************************************************/

import java.util.*;

public class Person
{   // Class fields
    private String firstName;
    private String familyName;
    private int dayOfBirth;
    private int birthMonth;
    private int birthYear;
```

- **private** facilitates data encapsulation;
- Class fields are hidden from the outside world.

With Parameters Constructor

```
public Person(String pFirstName, String pFamilyName, int pDayOfBirth, int pBirthMonth, int pBirthYear)
    {
        firstName  = pFirstName;
        familyName = pFamilyName;
        dayOfBirth = pDayOfBirth;
        birthMonth = pBirthMonth;
        birthYear  = pBirthYear;
    }
```

- The `p` denotes it is a method parameter.

Copy Constructor

```
public Person(Person pPerson)
    {
        firstName  = pPerson.getFirstName();
        familyName = pPerson.getFamilyName();
        dayOfBirth = pPerson.getDayOfBirth();
        birthMonth = pPerson.getBirthMonth();
        birthYear  = pPerson.getBirthYear();
    }
```

- The `p` denotes it is a method parameter.

Default Constructor

```
public Person()
{
    firstName  = "Mohamed";
    familyName = "Sallah";
    dayOfBirth = 15;
    birthMonth = 6;
    birthYear  = 1992;
}
```

Accessor Methods

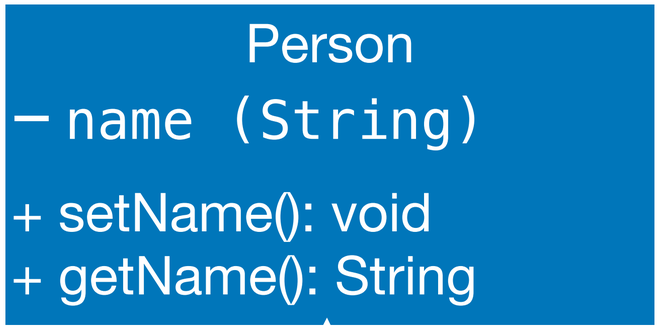
```
public String getFirstName()
{
    return firstName;
}

public String getFamilyName()
{
    return familyName;
}

public int getDayOfBirth()
{
    return dayOfBirth;
}

public int getBirthMonth()
{
    return birthMonth;
}

public int getBirthYear()
{
    return birthYear;
}
```

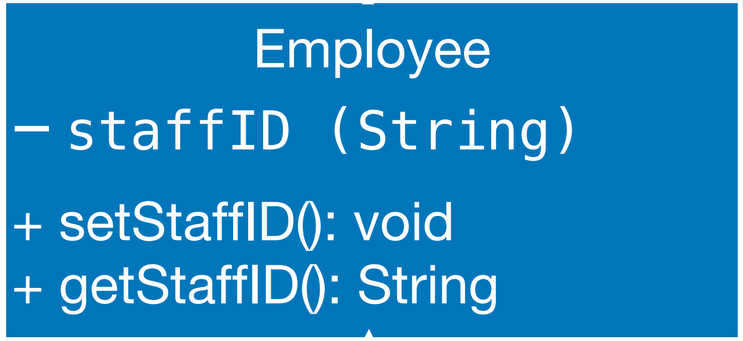
Mutator Methods

```
public void setFirstName(String pFirstName)
{
    firstName = pFirstName;
}

public void setFamilyName(String pFamilyName)
{
    familyName = pFamilyName;
}

public void setDayOfBirth(int pDayOfBirth)
{
    dayOfBirth = pDayOfBirth; // How could this be validated?
}

public void setBirthMonth(int pBirthMonth)
{
    birthMonth = pBirthMonth; // How could this be validated?
}

public void setBirthYear(int pBirthYear)
{
    birthYear = pBirthYear; // How could this be validated?
}
```

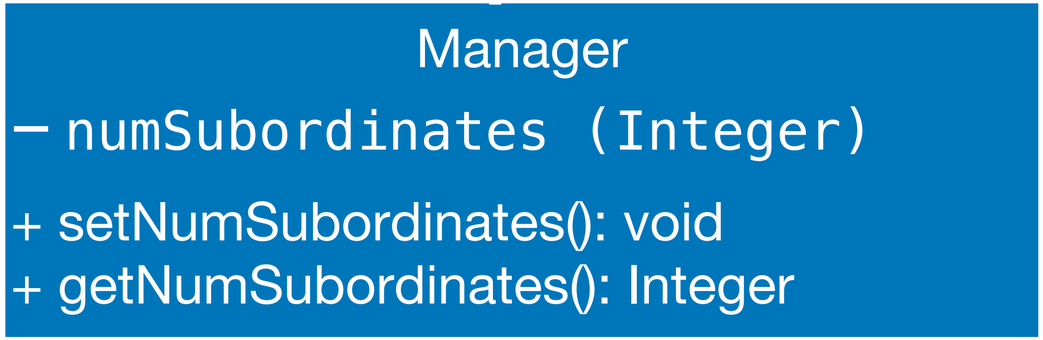
Inheritance

UML Diagram of Inheritance





Employee **is-a** Person



Manager **is-a** Employee

Inheritance in Java

```
public class Person
{
  …
}
```

```
public class Employee extends Person
{
    private String staffID;
    …
}
```

Accessors & Mutators for staffID must be added to the class.

```
public class Manager extends Employee
{
   private int numSubordinates;
   …
}
```

Accessors & Mutators for numSubordinates must be added to the class.

Calling super() in Constructors...

- **super()** calls the equivalent method in the parent class;
- **super()** can be provided with arguments for the super class methods;

```
public Employee(String pFirstName, String pFamilyName, int pDayOfBirth, int pBirthMonth, int pBirthYear, String pStaffID)
{
    super(pFirstName, pFamilyName, pDayOfBirth, pBirthMonth, pBirthYear);
    staffID = pStaffID;
}
// Copy Constructor
public Employee(Employee pEmployee)
{
    super(pEmployee);
    staffID = pEmployee.getStaffID();
}

// Default Constructor
public Employee()
{
    super();
    staffID  = "012345";
}
```

Calling super() in equals() and toString()

```
public boolean equals(Object pObject)
{
    boolean isEqual = false;
    Employee theEmployee = null;

    if(pObject instanceof Employee)
    {
        theEmployee = (Employee)pObject;
        if(super.equals(theEmployee))
        {
           if(staffID.equals(theEmployee.getStaffID()))
              isEqual = true;
        }
    }
    return isEqual;
}
```

```
public String toString()
{
    String tempString = super.toString();
    String staffIDString;
    staffIDString = "StaffID is " + staffID +
                    "\n" + tempString;
    return staffIDString;
}
```

OOP vs Non OOP

OOP Algorithm Design vs Non-OOP Algorithm Design

- Previously, algorithm was defined starting with main, used step wise refinement determining the processing steps, the data types/structures needed;
- Some steps refined into methods;
- Repeat process until design is refined and tested well enough to code.

OOP Algorithm Design vs Non-OOP Algorithm Design

- In OOP things are done slightly differently:
  - Classes are identified;
  - Role(s) or responsibilities assigned to each class;
  - Design the required methods (i.e., Constructors, accessors, etc.);
  - Thoroughly test each class using a test harness.
- Finally, deign the main algorithm and any more required methods making use of the already designed classes.

Where Does the Main Algorithm Go?

- `main` and other required methods have their own class;
- This class:
  - Used to drive the construction and use of objects of other classes;
  - Uses the functionality provided by other classes.

Working Program

- Ensure that **Person.java** is in the same directory as **PersonApp.java**.

- The object(s) can be passed to other methods, and the information used as required.

```
import java.util.*;
public class PersonApp
{
    public static void main(String[] args)
    {
        Person playerOne = new Person("Jordan", "Henderson", 17, 6, 1990);
        System.out.println("playerOne First Name: " + playerOne.getFirstName());

        Person playerTwo = new Person("David", "McMeekin", 7, 3, 1988);
        System.out.println("playerTwo was born in: " + playerOne.getBirthYear());

        Person playerThree = new Person(playerOne);
        System.out.println("playerOne First Name: " + playerThree.getFirstName());

        Person playerFour = new Person();
        System.out.println("playerOne First Name: " + playerFour.getFirstName());
    }
}
```

What Makes Up A Class?

- Each class is used to create objects and contains:
- A set of class fields (variables), which:
  - Are hidden from the outside world;
  - Are accessible anywhere in the class; and
  - Contain the values determining an object's state.

- A set of publicly accessible methods;
- A set of private (or hidden) methods;
- Public methods, usually, concerned with initialising, accessing or modifying the class fields;

Class Roles

- Every class is designed for a specific role;
- A software system's total set of functional requirements is broken down into a set of tasks;
- Tasks are grouped together and mapped to roles, roles are mapped to classes.

Classes and Objects

- A class specifies:
  - The communication with the rest of the system;
  - The method of data representation required;
  - Exactly how the required functionality is to be achieved.

- An object is an instance of a class;
- The class definition provides the template for the object; and
- The object provides the details for a particular instance;
  - Called a model class as it models a “real world thing”;

Static Classes

- Classes can also be a collection of related constants and methods available for use in other classes:
  - Known as a static class;
  - This type of class never has an object instance;
  - The Java Math class is an example.

equals() and toString()

Working with Values in Instance Variables

- Things change, values change;
- Values within instance variables evolve and need to change;
- Values within instance variables must be retrieved for use;
- Accessors (Getters) and Mutators (Setters) are class methods facilitating this.

Other Methods

- It is important to be able to:
  - to test for equality between instance variables' values; and
  - display instance variables' values.
- With `int` equality is determined with: `==`
- By convention all classes have additional two methods:
  1. `equals()`: Tests the equality of two objects; and
  2. `toString()`: Generates a String representation of the state information.

toString() Pseudocode

- It is up to the developer to phrase it in a meaningful way.

```
ACCESSOR: toString
IMPORT: none
EXPORT: personString (String)
ASSERTION: Returns a String representation of the object
    personString = "First name is " + firstName +
                   "Family name is " + familyName +
                   "Date of birth is " + dayOfBirth +
                   "/" + birthMonth +
                   "/" + birthYear
```

equals() Pseudocode

- For 2 objects to be equal, they must be of the same class and the state of each instance variable must be the same;

- That is, the values in each instance variable must be equal.

```
ACCESSOR: equals
IMPORT: inObject (Object)
EXPORT: isEqual (Boolean)
ASSERTION: Returns true if the two objects are equivalent
isEqual = FALSE
    IF inObject IS A Person THEN
      TRANSFORM inObject TO Person NAMED inPerson
      IF firstName EQUALS inPerson.getFirstName() THEN
        IF familyName EQUALS inPerson.getFamilyName() THEN
            IF dayOfBirth EQUALS inPerson.getDayOfBirth() THEN
                IF birthMonth EQUALS inPerson.getBirthMonth() THEN
                    IF birthyear EQUALS inPerson.getBirthYear() THEN
                        isEqual = TRUE
```

toString() Java

```
public String toString()
{
    String personString;
    personString = "First name is " + firstName +
                   "\nFamily name is " + familyName +
                   "\nDate of birth is " + dayOfBirth +
                   "/" + birthMonth + "/" + birthYear;
    return personString;
}
```

equals() Java

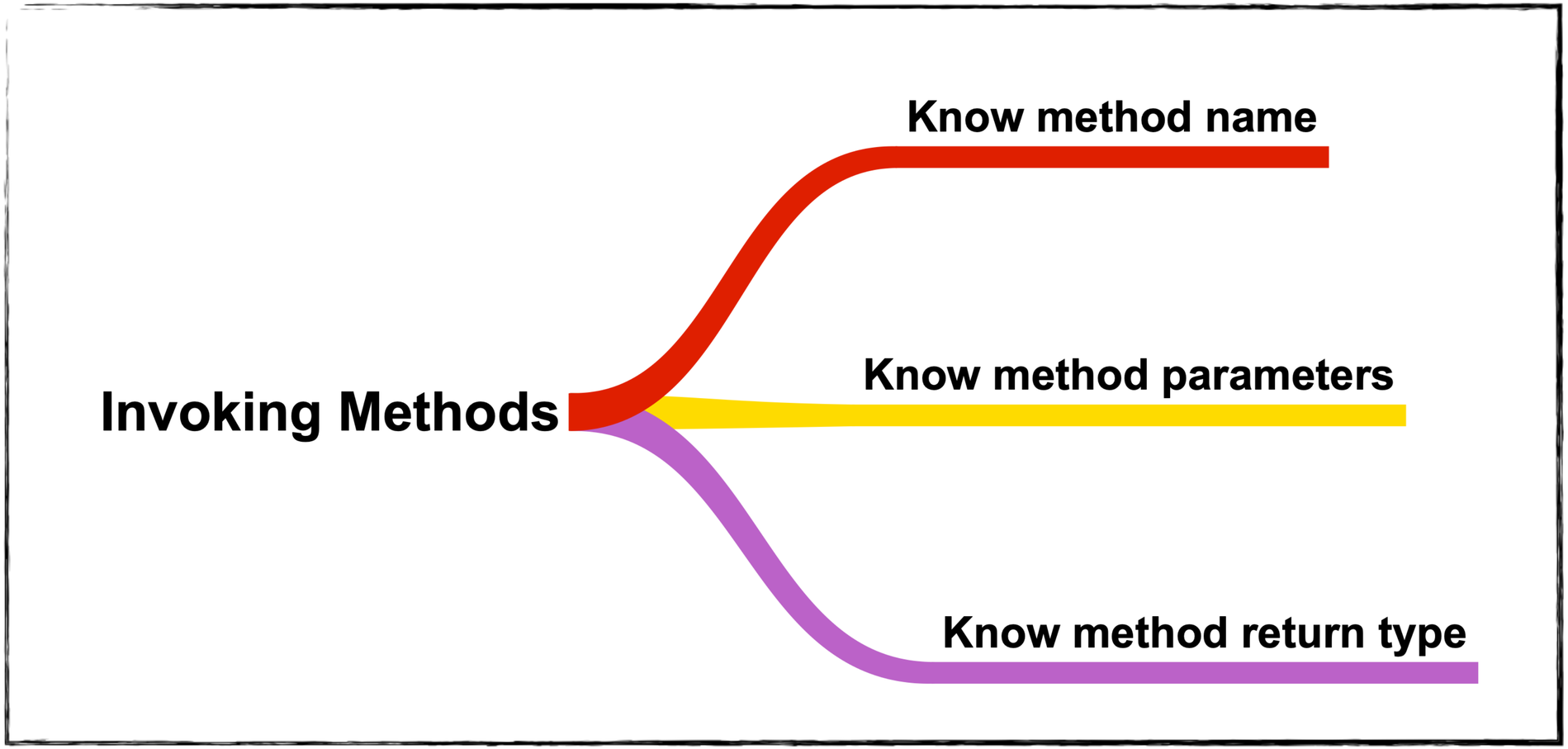
```
public boolean equals(Object inObject)
{
    boolean isEqual = false;
    Person inPerson = null;
    if(inObject instanceof Person)
    {
        inPerson = (Person)inObject;
        if(firstName.equals(inPerson.getFirstName()))
            if(familyName.equals(inPerson.getFamilyName()))
                if(dayOfBirth == inPerson.getDayOfBirth())
                    if(birthMonth == inPerson.getBirthMonth())
                        if(birthYear == inPerson.getBirthYear())
                            isEqual = true;
    }
    return isEqual;
}
```

Using a Class In a Program

```
import java.util.*;
public class PersonApp3
{
    public static void main(String[] args)
    {
        Person personOne;
        Person personTwo;
        Person personThree;

        personThree = new Person("James","Milner",4,1,1986);
        personOne   = new Person("Mohamed", "Sallah", 15, 6, 1992);
        personTwo   = new Person(personOne);

        if(personOne.equals(personTwo))
        {
            System.out.println("They are the same!");
        }
        System.out.println(personOne.toString());
    }
}
```

Using Methods

- When calling/invoking methods only 3 things need to be known:



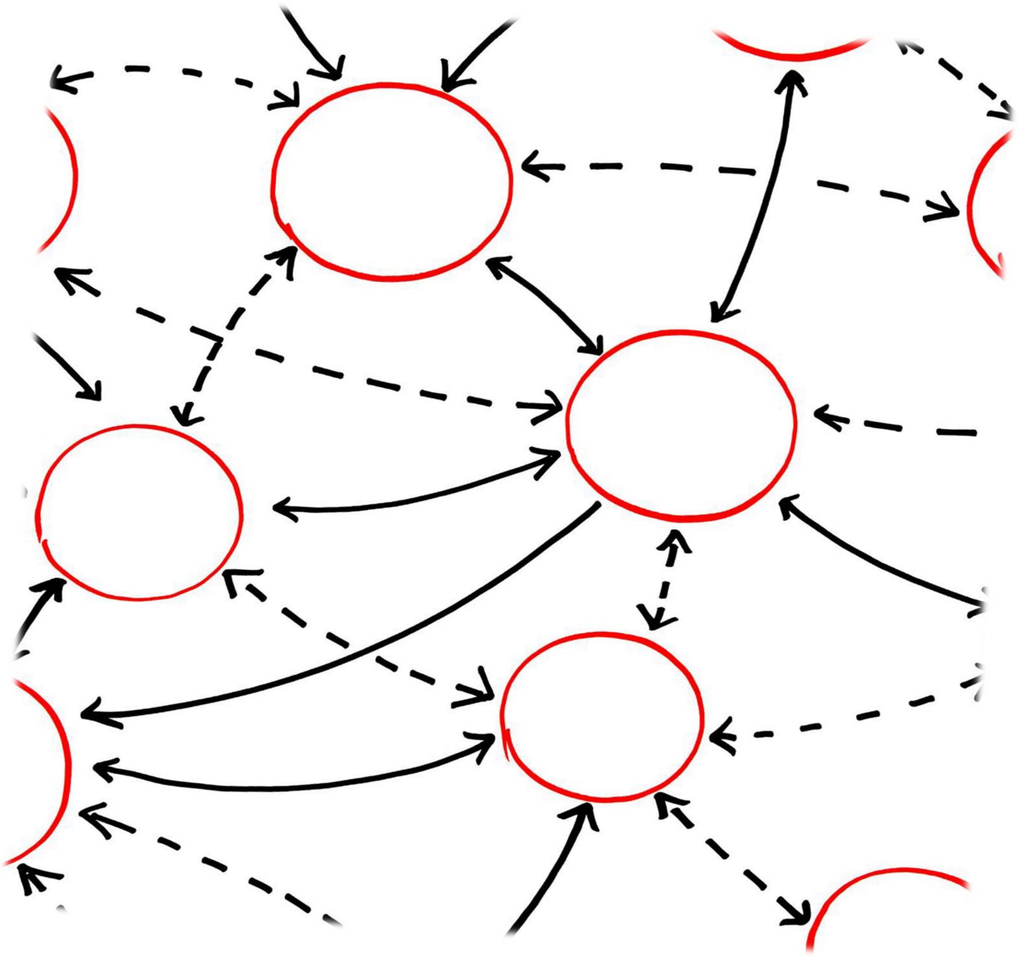
Aggregation

Introduction to Aggregation

- Combines individual components to create a unified whole;
- **Aggregation:** a special form of association, one class is a part of another class.
- Aggregation represents a “has-a” relationship.

Aggregation

- A way of representing relationships between classes and objects in object-oriented programming;
- Forms a ‘has-a’ relationship between them.



Composition & Aggregation

- Two types of relationships between classes and objects.

- **Composition**:
  - implies a strong relationship;
  - the component and the composite object have the same lifecycle.

- **Aggregation**:
  - implies a weak relationship;
  - the component object can exist independently of the composite object.

- **Unique Features of Aggregation:**
  - Aggregation: models a ‘part of’ relationship between classes and objects;
  - Aggregation: one or more objects are part of another object;
  - Composition: models a ‘composed of’ relationship between classes and objects.

A Class with Two Fields

- Create a **`Player`** **CLASS** with its class fields;
- **`playerName`** and **`position`** are of **`String`** data type.

```
CLASS Player:
    CLASS FIELDS:
        playerName (String)
        position (String)
    ...
```

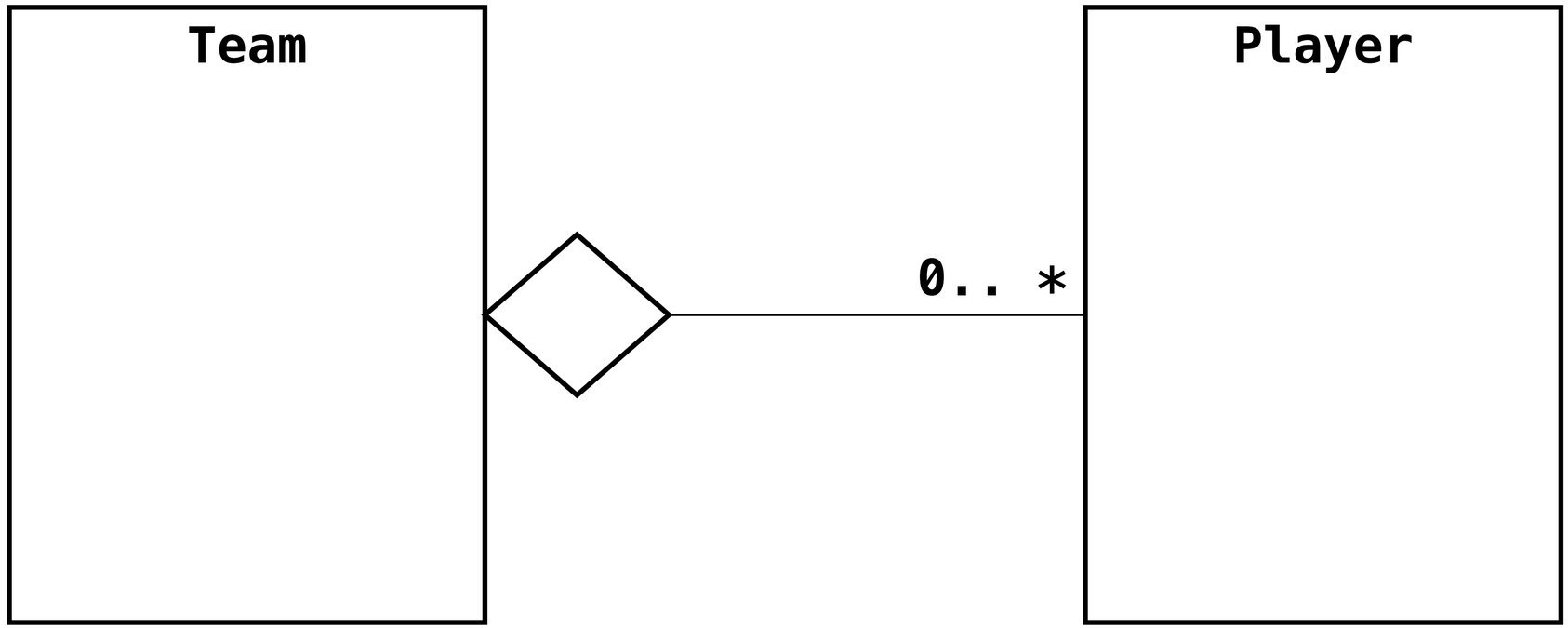
A Class with a Class as a Field

- Create a **Team** **CLASS** with its class fields;
- **teamName** is of **String** data type;
- **players** is a collection of **Player** data types.

```
CLASS Team:
    CLASS FIELDS:
        teamname (String)
        players (collection of Player)
```

Team & Player UML Classes

Team & Player Relationship in UML



- Team ‘has-a’ Player relationship;
- Team can have 0 to many Players

Player Class

- Player has 2 class fields;
- Constructor with parameters;
- Copy constructor;
- Default constructor; and
- All accessors & mutators...

```
1  public class Player
2  {
3      // Class fields
4      private String playerName;
5      private String position;
6
7      // Constructor with parameters
8      public Player(String playerName, String position)
9      {
10         this.playerName = playerName;
11         this.position   = position;
12     }
13
14     // Copy contructor
15     public Player(Player aPlayer)
16     {
17         this.playerName = aPlayer.getPlayerName();
18         this.position   = aPlayer.getPosition();
19     }
20
21     // Default constructor
22     public Player()
23     {
24         this.playerName = "Virgil van Dijk";
25         this.position   = "Defender";
26     }
27     ...
```

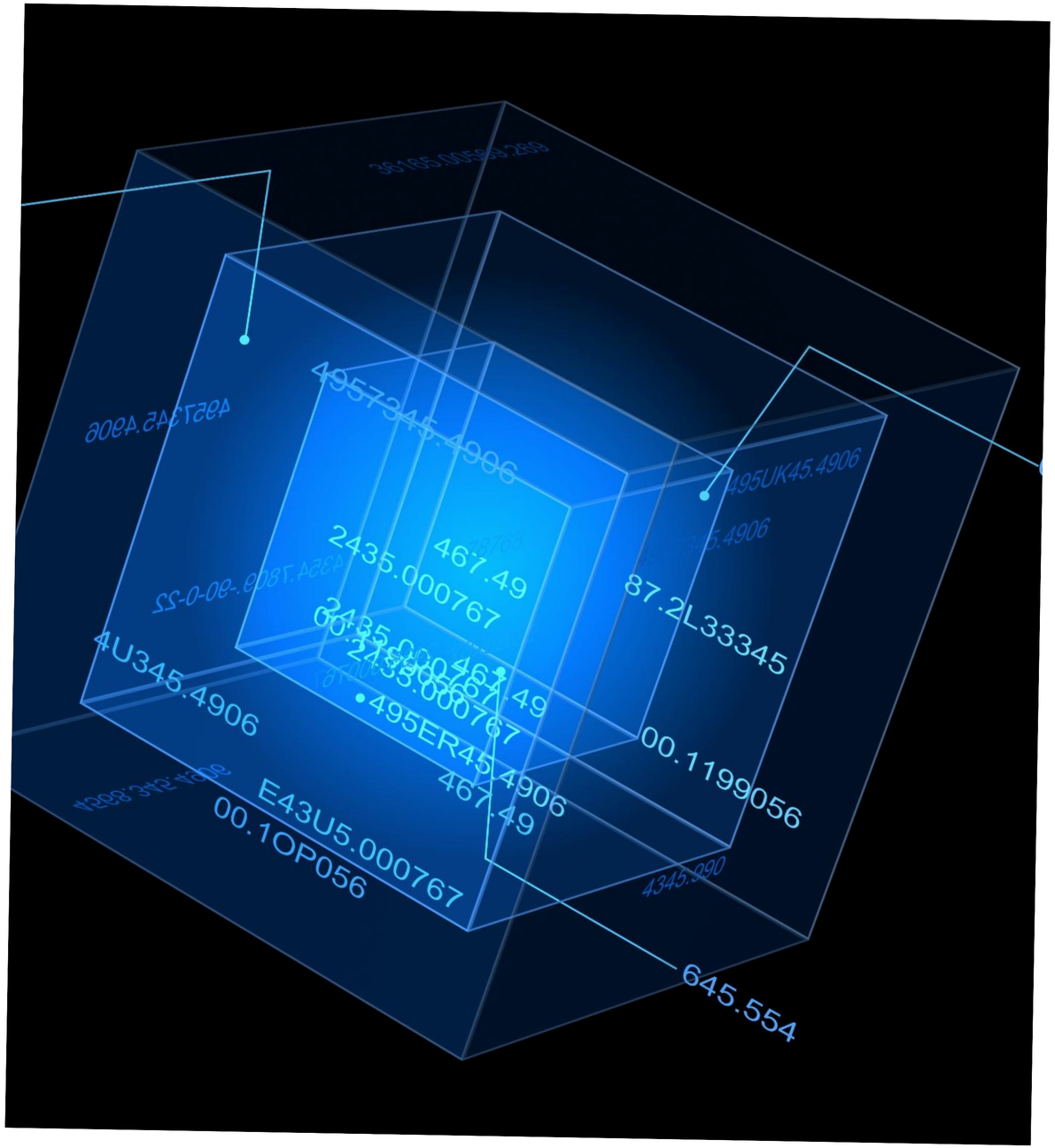
Team Class

- Team class has 2 class fields;
- Constructor with parameters;
- Copy constructor;
- Default constructor; and
- All accessors & mutators...

```java
public class Team
{
    // Class fields
    private String teamName;
    private Player[] players;

    // Constructor with parameters
    public Team(String teamName, Player[] players)
    {
        this.teamName = teamName;
        this.players  = players;
    }

    // Copy constructor
    public Team(Team aTeam)
    {
        // This is interesting as we are dealing with an array.
        // Left blank for you to think about.
    }

    // Default constructor
    public Team()
    {
        this.teamName = "Liverpool Football Club";
        this.players = new Player[16];
    }
    ...
```

Summarising

- Aggregation is an important concept in object-oriented programming;
- It allows objects to be composed of other objects;
- It is necessary to create complex programs that are easy to maintain and update.



A Little Bit More

Class Responsibility

- Each class has a designated role (responsibility) to fulfil when the the system demands it;

- A class is composed of:
- The data required to perform the role: class fields.
- The methods required to perform the role:
  - Methods the system will invoke are declared public:
  - They can be seen and invoked from outside the class;
  - Methods the public methods call when executing their algorithms are declared **private**;
  - **private** methods are not accessible from outside the class.

More Class Responsibility

- Take the requirements for a software application and:
  - Identify the classes required;
  - Assign specific responsibilities to each class;
  - Determine relationships between classes (see DSA COMP1002);
  - Repeat the above steps until the design is correct;
  - Each responsibility to be handled by that class and no other:
    - Example: If a responsibility for keeping track of a person's name is assigned to a class called Person then:
    - No other class should have this information except as an object of Person;
    - Other classes needing this information refer to this class when the information is required.

Object Communication

- Also referred to as message passing;
- When an object of one class calls an object of another class it is
- message passing;
- Public methods provide the functionality required for a class to fulfil its role;
- A class has five categories of methods:
  - Constructors;
  - Accessors (aka getters);
  - Mutators (aka setters);
  - Doing Methods; and
  - Private.

Classes in Java

- Each class in its own `.java` file;
- `main` is in its own class, usually the application name;
- All class fields are `private`;
- Methods are declared `public` or `private`;
- In PDI order your Java code consistently:
  - Declaration of class constants;
  - Declaration of class fields;
  - Declaration of the Constructors; Accessor methods;
  - Mutator methods;
  - Doing methods (`public`);
  - Internal methods (`private`).

Java private

- Declaring a variable or method as **private** means that the variable can only be accessed in the block in which it is declared;
- For methods this means the method is local to the class;
- For **private** variables this means that if:
  - the variable is global to the class then it can be referred to anywhere within the class but is hidden from the outside world;
  - the variable is local to a method, it can only be referred to within the method it's declared in.
- Always explicitly state if a method is **public** or **private**;
- Variables declared within a method block are **private** by default.

private Methods

- These are the methods that are used within the object, but are never seen by the outside world;
- They are hidden so that they can be modified or even replaced without causing problems in any other part of the software;
- To perform a complex task, it is often easier to break the task down into a series of sub-tasks and then refine each sub task independently of the others;
- i.e., The normal process of step-wise refinement still applies.

Doing Methods

- Are used for performing some required task;
- For example, generating a full date from the `dayOfBirth`, `birthMonth` and `birthYear` class fields, could be classified a doing method.