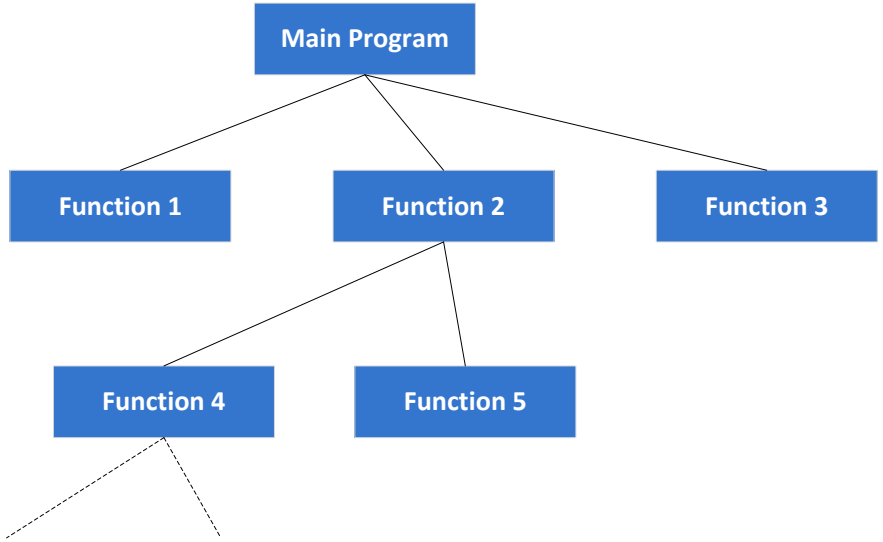
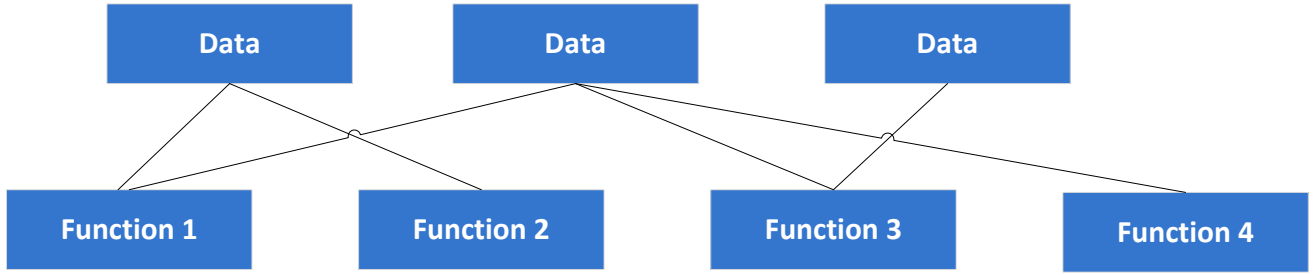
OBJECTS & CLASSES

Procedural Programming

- Program broken down into functions

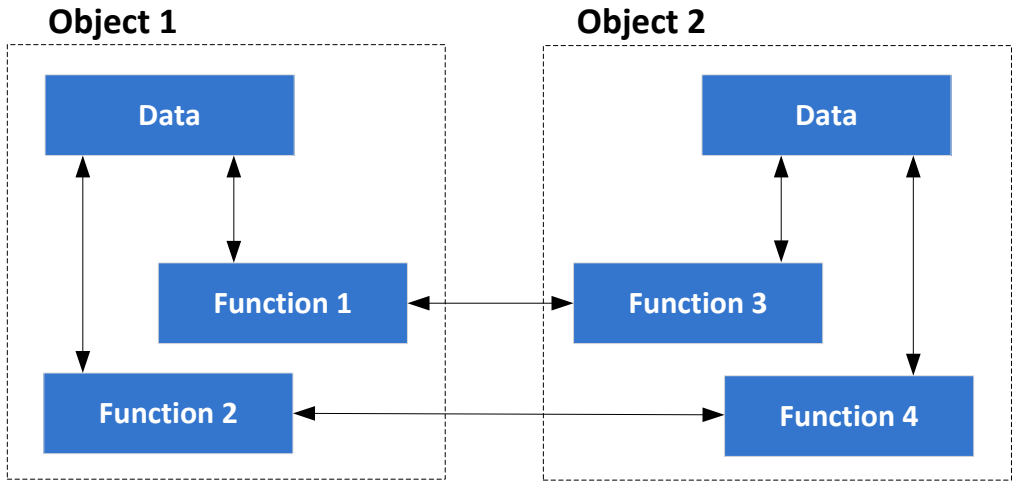


- Focus is on actions, not data
  - Data access is unstructured



Object-Orientation

- A style of software based on objects



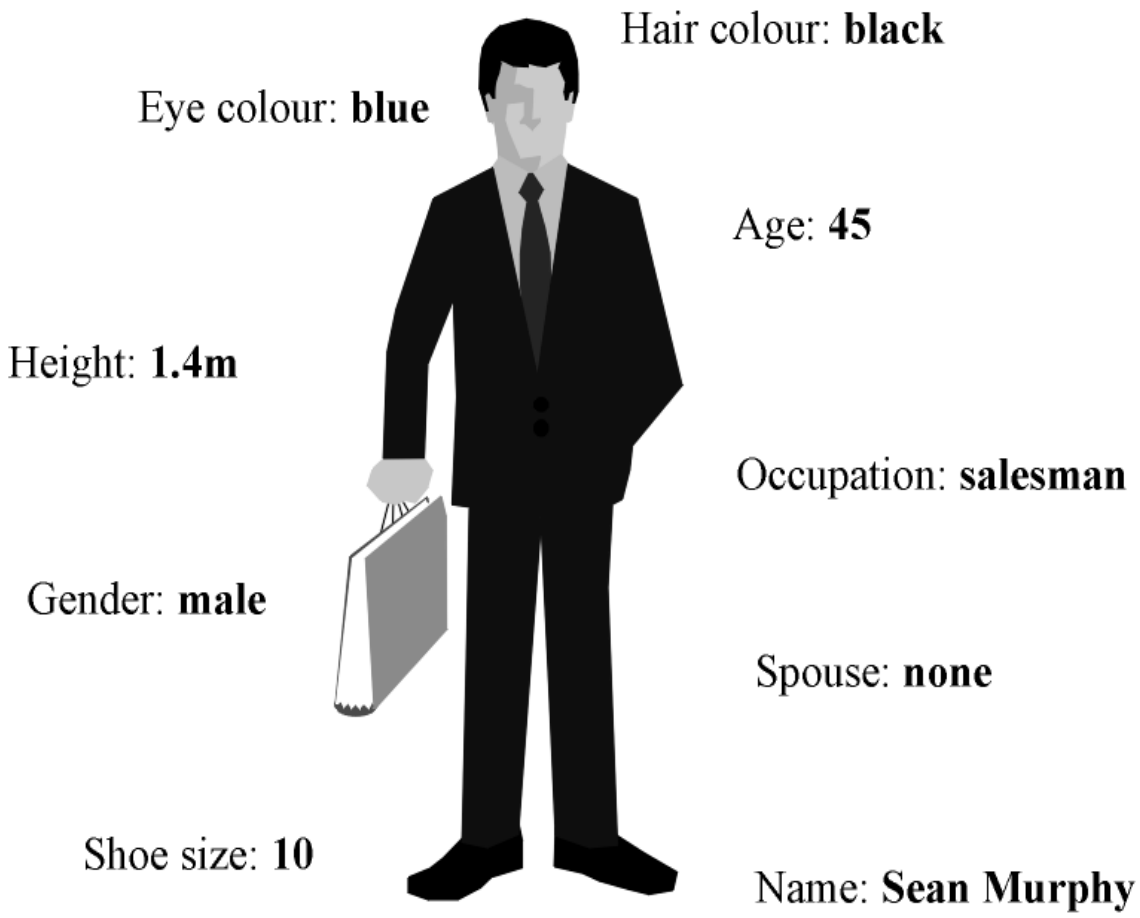
- Objects represent things of relevance to the problem
- Program defines, creates and manipulates objects

Object

- An object has:
  - Identity
  - State
  - Operations to manipulate state
- Examples of objects
  - People
  - Cars
  - Bank accounts
  - Printer Spooler
  - Some Java objects we've already met
    - String
    - HWPrinter

Object Attributes

- Objects have attributes
- Attributes have values
- Attributes can be references to other objects
- There can be many objects of the same type
- Diagrams courtesy of Paddy Nixon, Dublin



Object Operations

- Objects have operations (methods)
- Operations access and/or manipulate the state of an object

Grow

Dye hair

Change jobs

Marry

Die

Class

- Defines a set of characteristics of a thing
  - a template which needs to be filled in
- Its attributes
  - fields
  - properties
- Its behaviour
  - operations (methods)
  - features
- A class describes a general *type* of something

Object

- An instance of a class
- The class ***Person*** defines all possible people
  - Attributes: name, age, hair colour, spouse, etc.
  - Behaviour: grow older, dye hair, change job, etc.

- The object ***michael*** describes me
  - Name = Michael Young
  - Age = …
  - Hair colour = brown
  - Spouse = null
  - Can grow older
  - Can dye my hair
  - Can change my name (!)
  - etc.

- The object ***michael*** is an *instance* of the Person class

Examples

Class	Object (Instance)
Person	Alan Turing
Car	BD51 SMR
Boat	QE2
University	University of St Andrews

CONSTRUCTORS

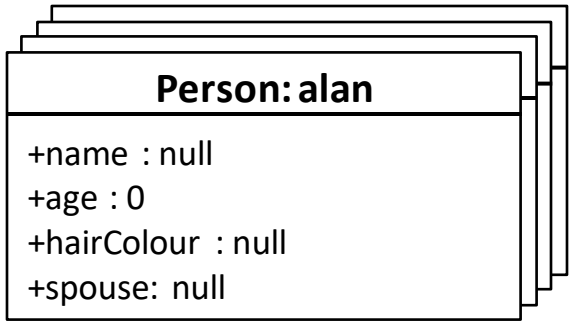
Classes in Java

```
public class Person {

    public String name;

    public int age;

    public String hairColour;

    public Person spouse;

    ...

}
```

Objects in Java

- Construct instance of a class

```
Person alan = new Person();
Person p = new Person();
```

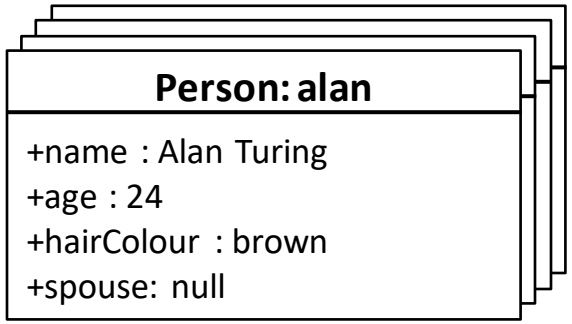


- Constructor is a special method to create an object
  - Used to set up the state of an object
- Constructor creates space in memory and initialises attributes
  - Attributes are given their default value to start:
    - Primitive types set to 0 (or false), reference types set to null.
- The constructor has same name as the class
  - e.g. Person class has a Person() constructor

Objects in Java

- Initialise public fields

```
alan.name = "Alan Turing";
alan.age = 24;
alan.hairColour = "brown";

p.name = "Ada Lovelace";
p.hairColour = "red";
```



- Messy
  - Object created and initialised in several lines
  - May forget to initialise all fields
  - Uncontrolled access to fields

Custom Constructors

- Use parameters to initialise to non-default values

```
public class Person {
    ...
    public Person (String fullname, int years, String hair) {
        name = fullname;
        age = years;
        hairColour = hair;
    }
}
```

- Construct the object & initialise fields in one line

```
Person alan = new Person("Alan Turing", 24, "brown");
```

- Cannot forget to initialise fields
- Can prevent uncontrolled access to fields
  - Allows us to *encapsulate* object state – more on this later!

Default Constructor

- If you do not write your own constructor java supplies the default constructor
  - e.g. **`Person()`**
  - Takes no arguments
  - Sets members to default values

- Default constructor is only available if you do not write your own.
  - You can re-add the default constructor:

```
public class Person {
    public Person() { }
    …
}
```

METHODS & ATTRIBUTES

How they interact

Defining Instance Methods

- So far, object has state, but no behaviour

```
public class Person {
    private String name;
    private int age;
    private String hairColour;
    private Person spouse;

    public Person (String fullName, int years, String hair){
        name = fullName;
        age = years;
        hairColour = hair;
    }

    public void updateAge(){
        age++;
    }

    public void dyeHair(String newHairColour){
        hairColour = newHairColour;
    }
}
```

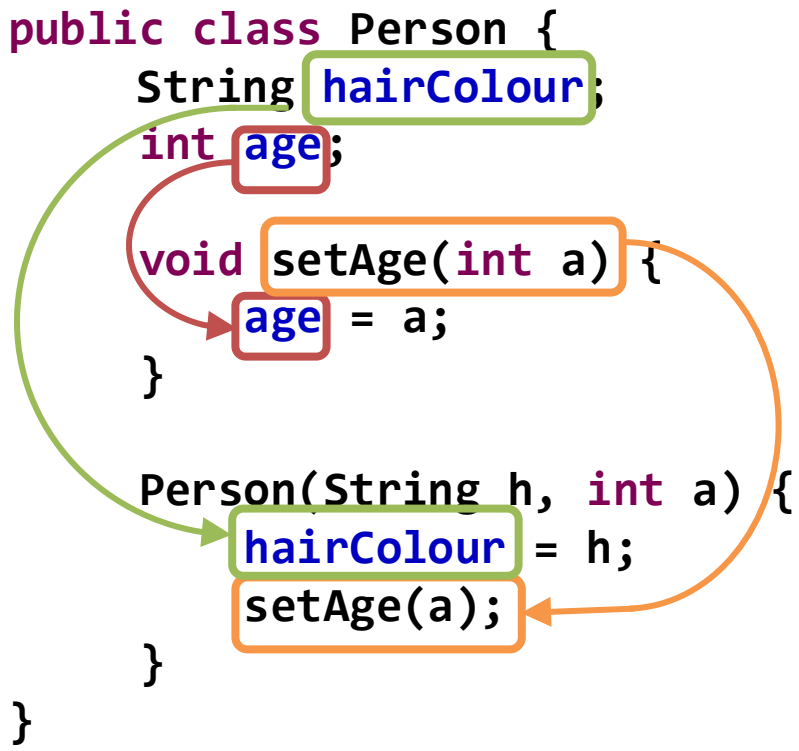
Overloading

- Overloaded methods and constructors
  - Same name but different number or types of parameter

```
public void updateAge() {
    age++;
}
public void updateAge(int years) {
    age = age + years;
}

public Person(String name, int age, String hair) { ... }
public Person(String name, int age, String hair, Person spouse) {  ... }
```

Referring to Members

- You can refer to members, and member functions, within a class simply by using their name.



The *this* keyword

- Refer to field of the current object if the name is 'shadowed'.

```
public class Person {
      String hairColour;
      int age;

       Person(String hairColour, int age) {
           this.hairColour = hairColour;
           this.age = age;
       }
}
```

- Refer to a constructor in the current Class

```
public class Rectangle {
     private int x, y;
     private int width, height;

     public Rectangle(int width, int height) {
         this(0, 0, width, height);
     }

     public Rectangle(int x, int y, int width, int height) {
         ...
     }
```

STATIC & NON-STATIC

Class vs Instance

The Static keyword (recap)

- A method or variable can apply to a specific instance (object), or to a class.
  - Instance method/variable
    - One copy per object
    - Most methods/variables should be instance methods/variables
  - Class method/variable
    - One copy per class, shared across all instances
    - A class method can be called without creating an object first
- The *static* keyword identifies which of these categories a method or variable falls into.

The Static keyword: methods

- Class method:

```
public static double getPi(){
    return 3.14;
}
Math.getPi();
```

- Instance method:

```
public void updateAge(){
    age++;
}
jon.updateAge();
```

The Static keyword: variables

- Class variable:

```
public static int numberOfPeople;
Person.numberOfPeople;
```

- Instance variable (attribute):

```
public int age;
alan.age;
```

REFERENCES & NULL

Null Pointers

- Instance methods are called on specific instances of objects. For example:

```
Person michael = new Person("Michael Young", 30, "brown");
michael.dyeHair("green");
```

- When called on an object which hasn't been instantiated a NullPointerException is thrown

```
Person nemo = null;
nemo.dyeHair("pink");
```

- If you do not give reference types a value, they default to null.

Reference Types (Recap)

- Java includes primitive types and reference types

- Primitive Types
  - *int*
  - *double*
  - *boolean*
  - *char*

- Reference Types
  - All objects are reference type
  - For example, String, arrays, Person

Standard Java Classes

- Listed in JavaDoc online
  - https://docs.oracle.com/en/java/javase/17/docs/api/
- Examples:
  - String
  - Integer
  - Math
  - Exception
  - Scanner

INHERITANCE

OO Principles

- Encapsulation
  - Information hiding
- Abstraction
  - Ignoring details of implementation
- Inheritance
  - Hierarchical relationships between objects
- Polymorphism
  - ‘many forms’

- All aim to promote code reuse, maintainability, flexibility, extensibility

Inheritance

- So far we can describe basic relationships between objects
  - A person's spouse
  - The car a person owns
- But we may also want to describe hierarchical relationships
  - Students and staff members are subsets of the *Person* class

Inheritance [2]

Staff
- name: String - age: int - payroll_number: int
+ Staff(name: String, age: int, pay_no: int) + getName(): String

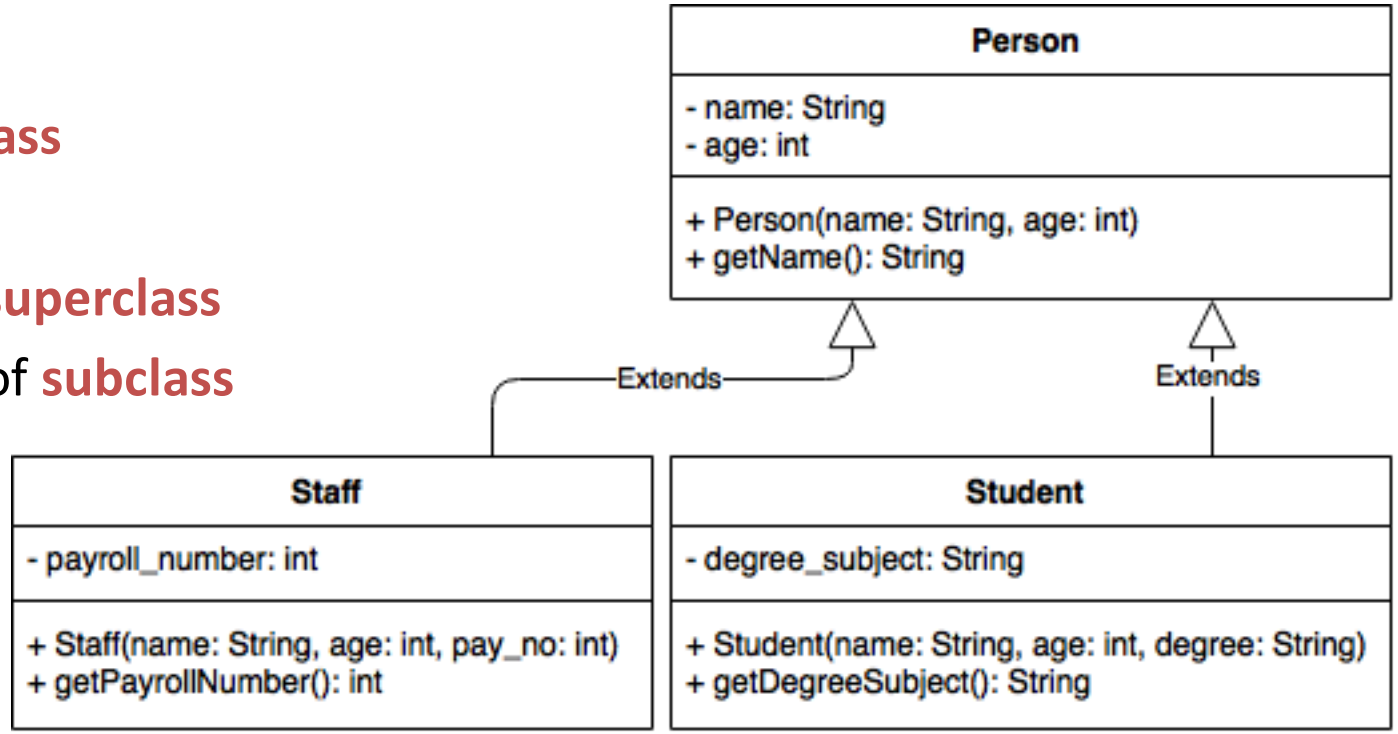
Student
- name: String - age: int - degree_subject: String
+ Student(name: String, age: int, degree: String) + getName(): String

- Both classes contain duplicate information. This is:
  - a waste of space
  - harder to manage

- Reducing duplication is desirable in general

Inheritance [3]

- One class **inherits** attributes and methods from another
  - Inherited attributes and methods are automatically part of the class
  - Don't need to be specified explicitly
- Terminology
  - **Subclass inherits from superclass**
  - **Subclass extends superclass**
  - **Subclass** is a **specialisation** of **superclass**
  - **Superclass** is a **generalisation** of **subclass**



Inheritance in Java

```
public class Person {
    private String name;
    private int age;

    public Person (String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
}

public class Staff extends Person {
    public int payrollNumber;
    ...
}

public class Student extends Person {
    public String degreeSubject;
    ...
}
```

- Only include attributes unique to this subclass
- **DON'T** repeat attributes from superclass

Constructors and Inheritance

- The *super* keyword can be used to call a constructor (or method) of a super class

```
public class Student extends Person {

    public Student(String name, int age, String degreeSubject) {
        super(name, age);
        this.degreeSubject = degreeSubject;
    }

    public void print() {
        super.print();
        System.out.println("Degree intention: " + degreeSubject);
    }
}
```

Inheritance Relationships?

- Pet, Dog
- Kitchen Appliance, Toaster
- Bicycle, Wheel
- Person, Woman
- Person, Man
- Class, Student
- Aeroplane, Jumbo Jet
- Team, Player
- Vehicle, Car
- Employee, Lecturer
- Wall, Brick
- Organisation, University

Inheritance in Java

- Every class that doesn't include the 'extends' syntax extends *Object*.

- Some of the methods in *Object*
  - toString()
  - equals()
  - hashCode()

- See *java.lang.Object* for more details

OVERRIDING & POLYMORPHISM

Overriding

- Overridden methods
  - method is redefined in a subclass to perform different actions

```
@Override
public String toString() {
```

- Note the `@Override` notation - not required by Java but helpful
  - Check superclass contains a method with that signature at *compile* time
  - Picks up mistakes early - did you write “tostring” by accident?
  - Is required by some coding styles
- Useful for documentation

Overriding vs Overloading

- Overriding
  - Redefine a method in a subclass
  - Method in subclass & superclass have same signature
  - Actual type of object determines which is called

- Overloading
  - Define multiple methods with same name in a class
  - Methods have same name but different parameters
  - Method name & actual parameters determine which version is called

Polymorphism

- ‘Many forms’
- A variable can reference many types of object
  - Object of subclass can be used where object of superclass is expected
- Many definitions associated with the same method signature
  - Overriding

```
public class Worker {
   public void name() {
      System.out.println("Worker");
   }
}

public class Manager extends Worker {
   @Override
   public void name() {
      System.out.println("Manager");
   }
}
```

```
...

Worker w = new Manager();
w.name(); // prints 'Manager'

...
```

Polymorphism [2]

- Different data types can be accessed through a uniform superclass (or interface shared by all classes)
- Example:
  - A program stores details of Persons, Students and Staff
  - The Person class defines a printDetails method
  - Both Student and Staff extend Person
    - Only Student overrides the printDetails method in Person
  - Program deals with a collection of objects of type Person
  - The collection can contain Person, Staff and Student objects
  - The program can safely call the printDetails method on any of the objects regardless of whether they are a Student, Staff, or Person.
  - See code on StudRes:
    - https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W02_OO/CS5001_PersonStudentStaffExample/
    - https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W02_OO/CS5001_PolymorphismPersonExample/

ABSTRACT CLASSES

Abstract Classes

- Class that cannot be instantiated directly
- Useful when it is not logical to create an instance of that class
- For example, in a university people-management system
  - Define classes Person, Student, Staff
  - Someone is either a student or a staff member
  - No-one should be represented by just a ‘Person’ object

Abstract Classes [2]

- Abstract classes can have attributes
- Abstract classes can have ‘concrete’ methods
  - There is sensible ‘default’ behaviour
- Abstract classes can also have abstract methods
  - No method body
  - There is no sensible ‘default’ behaviour
  - All subclasses must define their own behaviour for the method

Abstract Example

```
public abstract class Shape {
  private int x, y;

  public void setX(int x) {
    this.x = x;
  }

  public abstract void calcArea();

}
```

```
public class Triangle extends Shape {
  public Triangle () {
    ...
  }

  public void calcArea(){
    ...
  }
}
```

- Shape is abstract and cannot be instantiated
- The `calcArea` method must be overridden for each subclass
  - Since it is abstract, it is not implemented in Shape
- Another example:
  - https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W02_OO/CS5001_PersonAbstractExample/

ENCAPSULATION

Encapsulation

- An object's fields are (should be) **encapsulated**
  - the fields (attributes) are declared to be ***private***
  - they can't be seen directly by other objects
  - they can be accessed by that object's methods
    - often provide public *accessor* methods (getters/setters)

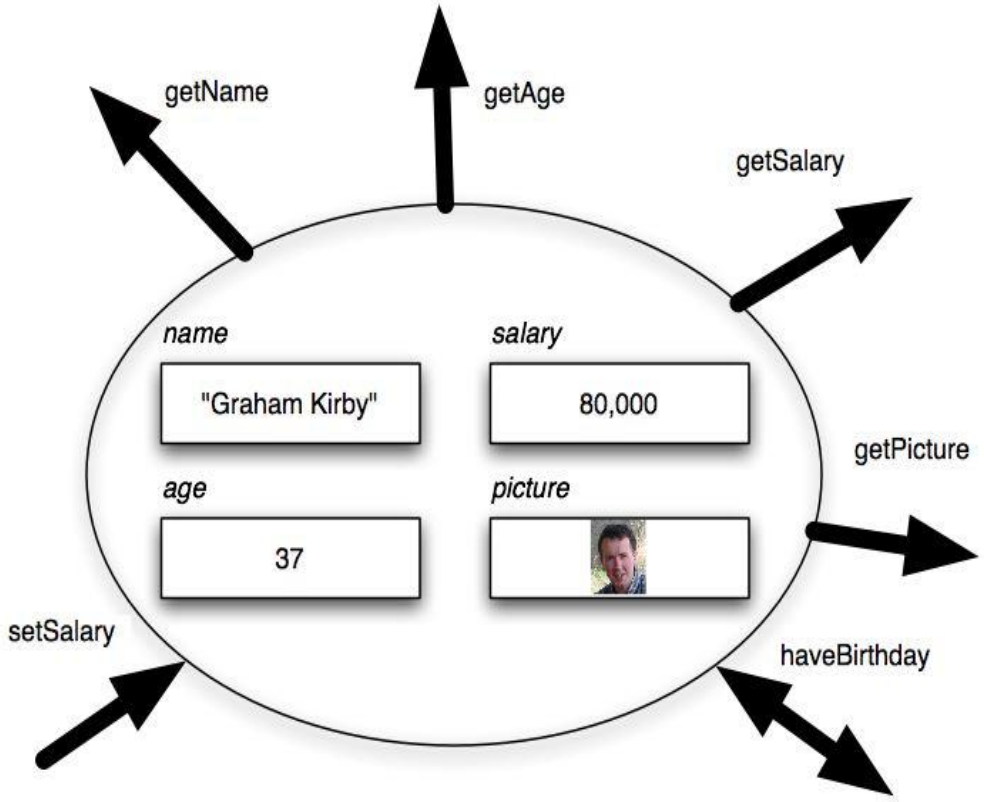
- Improves control over how fields are read and updated

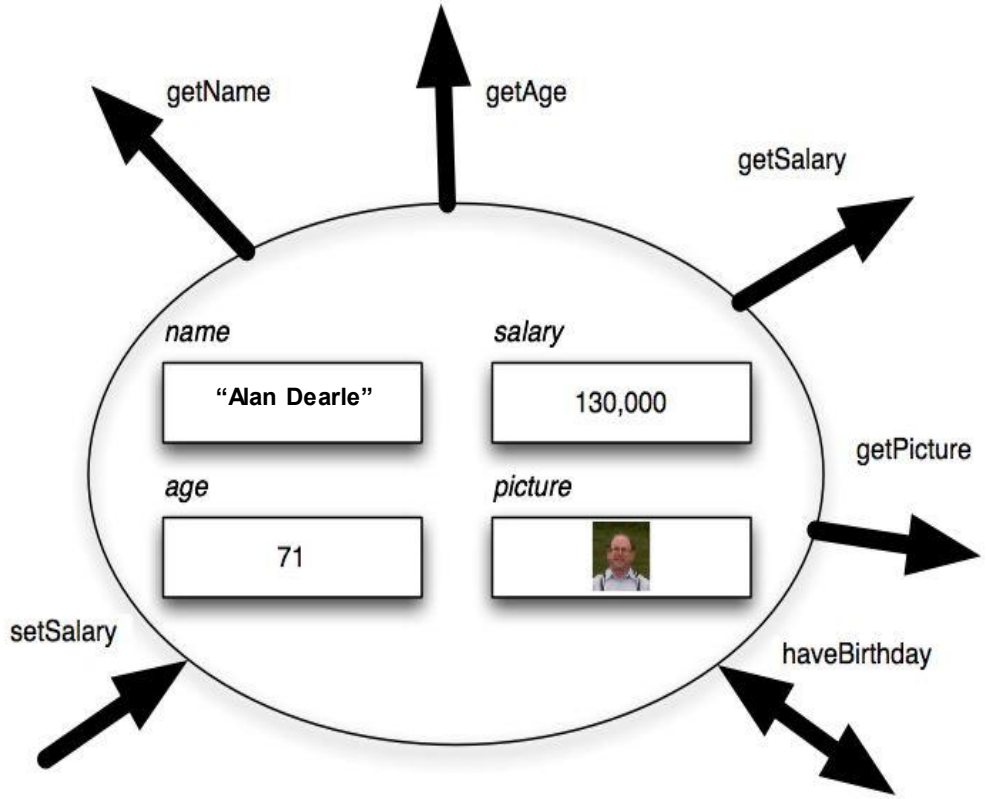
- Aids maintainability of software
  - Can change implementation details (name or type of field/method implementation)
  - So long as publicly accessible method signatures stay the same software relying on this class continues to work

Encapsulation Example

```
public class Person {
   private int age;
   ...
   public String setAge(int age) {
      if ( age < 0 ) {
         throw new IllegalArgumentException("Age must be non-negative");
      }
      else {
         this.age = age;
      }
   }
}
```

Encapsulated Objects





Packages

- Hierarchal namespace
  - A way to organise Java files
  - java.* & javax.*- Standard java classes
  - java.io.* - Classes related to I/O
- Package hierarchy corresponds to directory structure
  - Classes in *cs5001.samples* must be in the *cs5001/samples/* directory
- De facto standards
  - Packages named using hostname and project name e.g.
    - *uk.ac.standrews.cs.mct25.cs5001*
- Stops implementations clashing
  - *mct25.cs5001.Person* is distinct from *ozgur.cs5001.Person*

Controlling Access to Members

		Accessible From			
		Within Class	Within Package	Within Subclass	World
Modifier	public	Y	Y	Y	Y
	protected	Y	Y	Y	N
	no modifier (package)	Y	Y	N	N
	private	Y	N	N	N

- Modifiers can be applied to: methods, classes, class variables, instance variables

Code Reuse and OO

- Benefit of OO and Encapsulation
  - The implementation details and attributes are not exposed
  - Objects (defined by a Class) contain all the operations (methods) required to interact with them
  - Easy to share Classes among different software projects

- Imagine someone has written a WebServer Class
  - You don't need to know how it has been implemented, you just need to know how to use:

```
WebServer ws = new WebServer(80); // port 80
ws.start(); // start web server running
```

PROGRAM DESIGN/MODELLING

Program Design/Modelling

- How do you choose what should be a class, object or method in your program?

- Design new classes
  - to allow creation of new kinds of objects
  - to store particular kinds of information
  - to process information in a particular way

- Steps
  - determine classes
  - determine fields
  - determine methods and parameters
  - determine relationships between classes
  - [implement methods]

Program Design

- Noun/verb identification
  - Nouns for classes or attributes
  - Verbs for operations
- Use sentence structure to determine which class an operation belongs in and what parameters it might take

- Example: a person can buy a car
  - Nouns: person, car
  - Verbs: (can) buy – operation of person takes car as parameter
- Verb identification/sentence structure is not perfect
  - System descriptions often miss out or obscure the critical verb or structure

Modelling Example: Plan Classes

"The system should store information about the modules offered by Schools and Departments in the University. For each module it should record course code and description, the staff teaching the module, and the students taking it."

- What classes should be used?
  - noun identification

Modelling Example: Plan Classes [2]

"The system should store information about the modules offered by Schools and Departments in the University. For each module it should record course code and description, the staff teaching the module, and the students taking it."

- Discard inappropriate candidates (and duplicates)
  - terms that are too general
  - synonyms
  - plural/singular

Modelling Example: Plan Classes [3]

"The ~~system~~ should store ~~information~~ about the modules offered by Schools and ~~Departments~~ in the ~~University~~. For each ~~module~~ it should record course code and description, the staff teaching the ~~module~~, and the students taking it."

- Discard:
  - system, information, university, departments (synonym of Schools), module (singular of modules)
- Potential Classes:
  - Module, School, CourseCode, Description, Staff, Student
    - naming conventions: singular, initial capital

Modelling Example: Plan Attributes

- What information should be stored for each class?

- Module
  - Course code
  - *Description*
  - School offering module
  - Enrolled Students
  - Teaching Staff
- School
  - Modules on offer
  - *Name*
  - *Staff*

- Be aware of any two-way relationships in model
  - e.g. between *Module* & *School*
  - Can be difficult to maintain
- Consider any implicit relationships

Modelling Example: Plan Attributes [2]

- CourseCode
  - ?
- Description
  - ?
- Staff
  - Taught Modules
  - *Name*
  - *Payroll number*
  - School
- Student
  - Modules taken
  - *Name*
  - *Student ID*

- Re-evaluate Classes
  - CourseCode & Description have no sensible attributes
  - May not be Classes, just attributes of Module

Modelling Example: Plan Operations

"The system should store information about the modules offered by Schools and Departments in the University. For each module it should record course code and description, the staff teaching the module, and the students taking it."

- What operations should be used?
  - Verb identification

Modelling Example: Plan Operations [2]

"The system should store information about the modules offered by Schools and Departments in the University. For each module it should record course code and description, the staff teaching the module, and the students taking it."

- System
  - stores information
- School
  - offer Modules
- Staff
  - teach Modules
- Student
  - take Modules
- Module
  - record course code
  - record description
  - record Staff teaching it
  - record Students taking it

Modelling Example: Plan Operations [3]

- Give the operations more sensible names and specify parameters

- School
  - addModule (Module)
  - listModules
- Staff
  - assignModule (Module)
- Student
  - enroll (Module)

- Module
  - setCourseCode (String)
  - setDescription (String)
  - setTeacher (Staff)
  - enroll (Students)

- You should also be able to add appropriate accessors