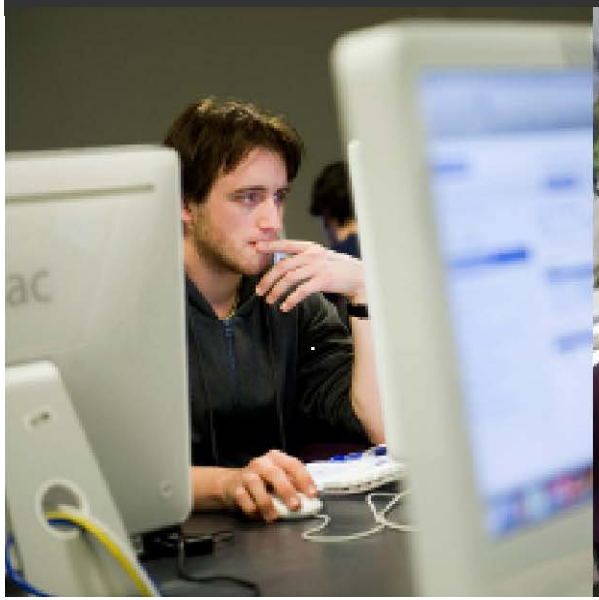
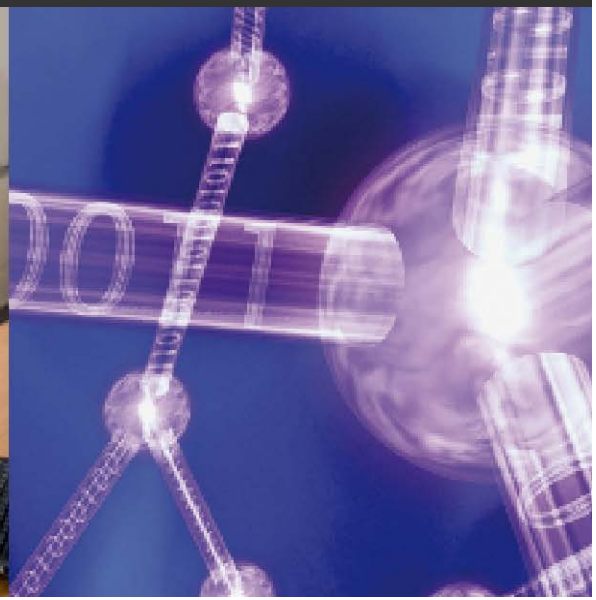
MONASH University

Information Technology

Module 4:
Object Orientation

FIT2034 Computer Programming 2
Faculty of Information Technology





Object Orientation

- An approach to solving (computing) problems that reasons about objects and their interactions
  - Contrasts to an approach which is purely task oriented
- Objects represent things or concepts of the real world
- Objects maintain a state
- Objects send messages between each other to invoke behaviours

Definitions (1)

- **Encapsulation** – the grouping of related ideas into one unit, which can thereafter be referred to by a single name (Page-Jones 2000)

- **Object Encapsulation** – the packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only via the interface provided by the encapsulation (ibid.)

Definitions (2)

- **Attribute** – a data item, a value, which has significance to the encapsulated unit
- **State** – a description of how an object is at a particular moment in time.
  - Defined by the current values of its attributes.
- **Operation** – something an object is capable of doing, a behaviour offered by the object
- **Method** – The actual code which specifies what processing is to be done in the CPU to achieve an operation's purpose

Definitions (3)

- **Class** – a logical grouping of a set of objects that all have the same set of defined attributes and provide the same set of operations, and which behave in the same manner when in same states with same triggers
- **Object** – one particular existent item of a class.
  - We say that an object is an **instance** of a class

Objects (Instances)

- An object has an **identity** – i.e. it has a unique existence apart from other objects of the same class/type
  - i.e. attribute values are personal
- An object may be **associated** or related to other objects of any class/type
  - Temporarily (for one method call),
  - Periodically (for a while, beyond one method call),
  - Everlasting (for whole existence)
- Associated Objects can interact by sending messages to each other

Every object has..

- A unique identity and a current state

dog3

Rusty
red
40cm
45 cm

Sprint
brown
1m
75 cm

sprint

Dog
name colour length height
run chase bark

1
Dog
class

myDog

Scott
black
30cm
20cm

3 objects of the one class

Information Hiding

- The principle of hiding the details of the implementation, and revealing publicly only an interface by which to interact with the object
- It is “the use of encapsulation to restrict from external visibility certain information or implementation decisions that are internal to the encapsulation structure” (Page-Jones)
- Enforced in Java through use of **public** and **private** modifiers
  - If something is private, only code within the same class can refer to it
  - If something is public, any code can refer to it (through using the dot operator)

Using Public and Private

- All fields (attributes/instance variables) should be private
  - Prevents other classes from tampering with data
  - Encapsulates the data inside the object
- Some methods should be public
  - Collectively referred to as the “public interface” of the class
  - Allows us to control access to fields through procedural code.
    - The method can be written to ‘veto’ an action if the object deems it inappropriate at the moment. E.g. trying to extend a library loan after the loan has become overdue may be prevented.
- Some methods can be private
  - Helper methods that are called within the public methods

Interaction between objects

- **Message** – a request from one object (**caller**) to another object (**recipient** or **called**) for the recipient to perform an operation
  - Recipient can be same object as caller (self-messaging)
- **Interrogative** Message – asks the target object to reveal something about itself.
  - A response to the caller is required.
- **Imperative** – requests (demands) that the object takes some action on itself, or another object, or the environment in which it exists.
  - A response to the caller may not be required, but could occur.
- **Informative** – tells the target object something which may be of interest to it.
  - No response is expected to be sent back to caller, but target may perform some actions anyway.

Types of *operations* (*implementation of messages*)

- **Constructor** – create a new object in memory, with a sensible initial state
- **Destructor** – some languages provide this (**not Java**). Allows for clean-up before memory is de-allocated.
- **Accessors** – report the state of an object's attributes.
  - Supports interrogative messages. (see next slides)
- **Mutators** – modify the state of an object's attributes.
  - Supports imperative messages
- **Notification** – take note of interesting events.
  - Supports the informative messages
  - Useful in GUIs

Accessor and Mutator Methods

- Instance variables (attributes) are declared with the private visibility modifier
  - Making them unreachable directly by code outside the class
  - This is encapsulation + information hiding in action
- We can code public methods to allow restricted access to these instance variables
  - Whatever restrictions we want (including none)
  - These methods should be the only access to the instance variables
- Accessor
  - Public methods that return the current value of an instance variable
- Mutators
  - Public methods that overwrite the current value of an instance variable with their formal parameter value (supplied by an actual parameter value in the call to the Mutator)

Accessor/Mutator - Example

```
public class Circle {

   private float radius;

   public Circle(float initRadius){
      if (initRadius > 0)
         radius = initRadius;
      else
         radius = 0;
   }

   // radius Accessor
   public float getRadius(){
      return radius;
   }

// radius Mutator
   public void setRadius(float newRadius){
      if (newRadius > 0)
         radius = newRadius;
   }
}
```

private instance variable

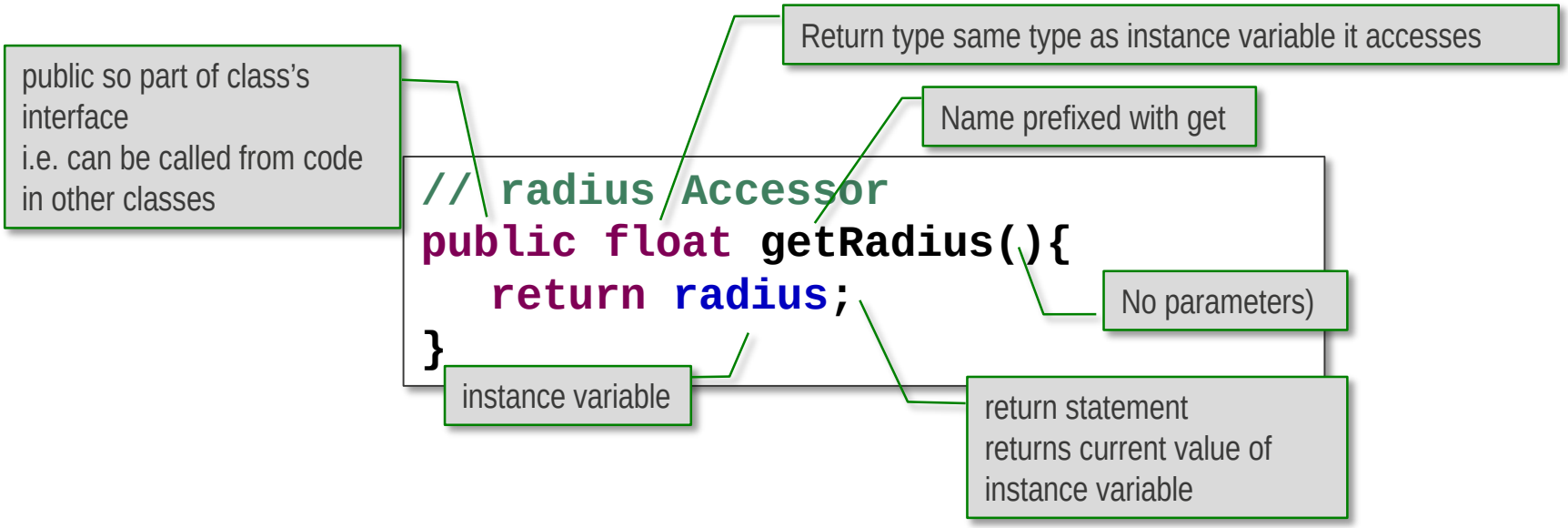
public Constructor:
Same name as class, no return type.
With validation of initial value of radius.

public Accessor:
Returns current value of radius.
No validation which is typical because the value of radius is not being changed.
This method can be called from inside or outside the Circle class

public Mutator:
Radius set to new value with validation which is typical because the value of radius is potentially being changed using a formal parameter value which was initialised with an actual parameter value in a call to the Mutator method from inside or outside the Circle class

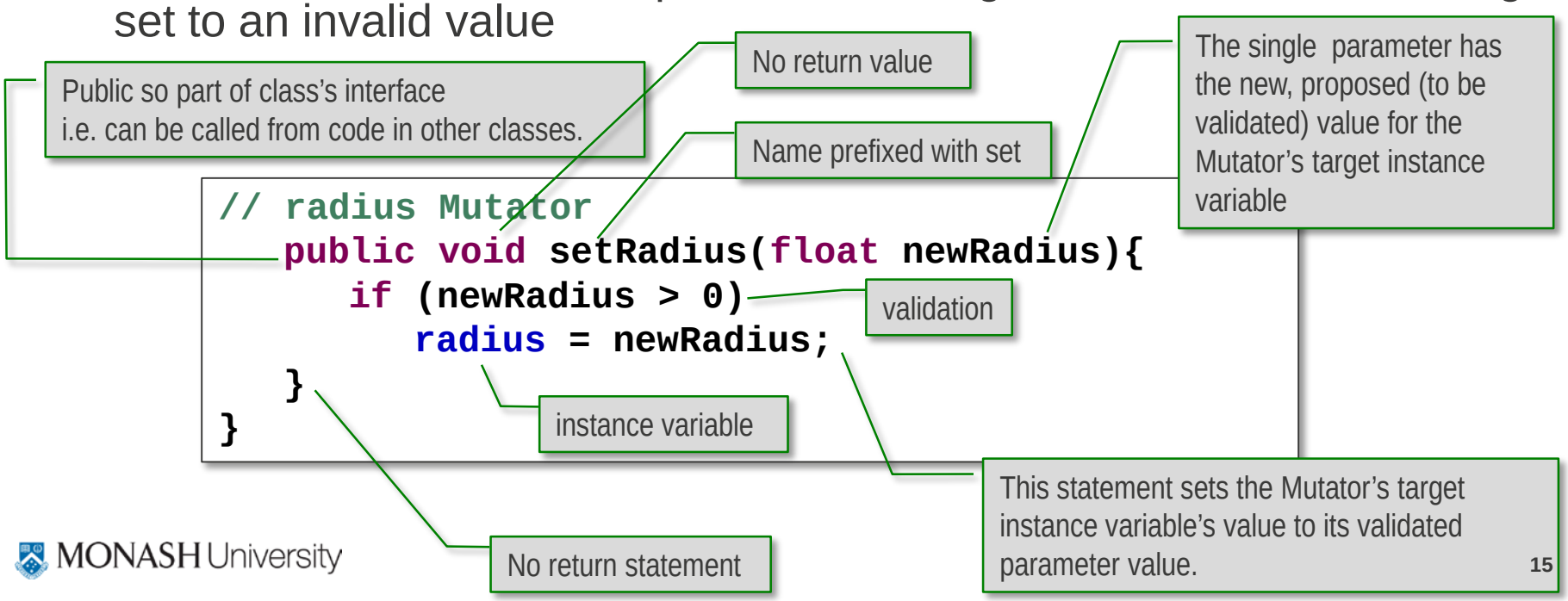
Accessors

- Accessors should:
  - Return (get) the current value of an instance variable
  - Have a return type and return statement but no parameters
    - Accessors provide a data out channel not a data in channel
  - Have the name format: get*InstanceVariableName*
  - Be public so code in other classes can call them



Mutators

- Mutators should:
  - Change (set) the value of an instance variable in a controlled way
  - Have a parameter that carries the new, proposed (but to be validated) instance variable value but no return value
    - Mutators provide a data in channel not a data out channel
  - Have the name format: set*InstanceVariableName*
  - Be public so code in other classes can call them
  - Contain validation code to prevent their target instance variable being set to an invalid value



Mutators

- It's not uncommon for Mutators to have a boolean return value
  - To indicate the success or failure of changing their target instance variable's value
    - i.e. did the new instance variable value they propose pass the Mutator's validation
  - This can be important for the calling code so it can react appropriately
  - e.g.

```
public boolean setRadius (float newRadius){
   if (newRadius > 0) {
      radius = newRadius;
      return true;
   }
   else
      return false;
}
```

Accessors and Mutators - Example

```
public class Rectangle {
    private int height;
    private int width;

    public Rectangle(int initHeight, int initWidth){
        height = initHeight;
        width = initWidth;
    }

    public int getHeight(){
        return height;
    }

    public int getWidth(){
        return width;
    }

    public void setHeight(int newHeight){
        height = newHeight;
    }

    public void setWidth(int newWidth){
        width = newWidth;
    }

    public int computeArea() {
        return height * width ;
    }

    public int computePerimeter() {
        return  2 * (height + width);
    }
    :
}
```

Instance variable declared inside the class but outside all methods. Their scope is the entire class i.e. All the class's method code.

This Constructor initialises all instance variable values without validation in this case (NOT recommended).

Accessors returning (getting) an instance variable's value.

Mutators setting an instance variable's values without validation in this case (NOT recommended!).

Plain methods.
They do not simply get or set an instance variable's value although, as you can see in this case, they may still get (they should not set) the value of several instance variables to perform a calculation then return the value of that calculation.

Accessors, Mutators – When?

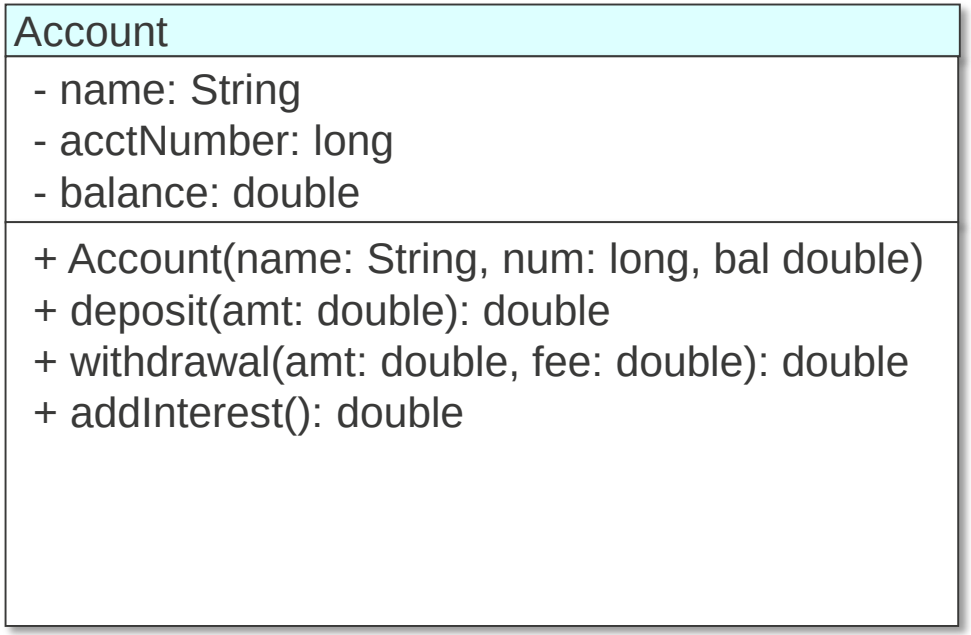
- In general
  - All instance variables should have an Accessor and Mutator BUT
- No Mutator
  - When instance variable is read-only
  - e.g. an identifier that should not be changed after instantiation
    - e.g. a bank account object's account number instance variable
- No Accessor
  - Instance variable is write-only
  - e.g. this would allow objects of a class to have secret data although it would not be secret for any code that uses a Mutator to set this data
    - This code knows what value it set the instance variable to
    - Example: a password kept in the object, checked by a public method
  - Write-only instance variables are uncommon

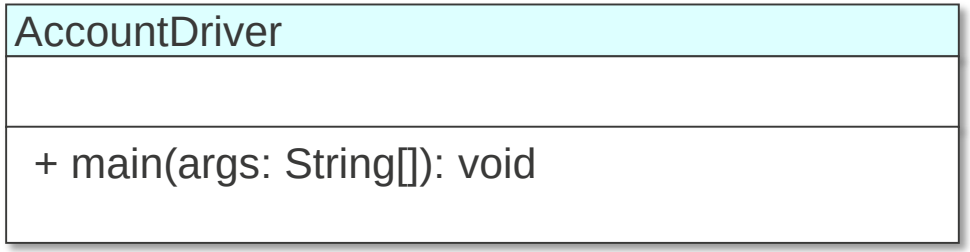
Scope and Lifetime

- Mutator methods set the values of instance variables
- The lifetime of instance variables is the lifetime of their object instance which extends beyond the execution time of their Mutators
- Be sure you understand that:
  - Although local variables of a method (including its formal parameters) are erased from memory when their method ceases execution
  - Any changes made by Mutator methods to instance variables are not erased when the Mutator method ceases execution

Classes and Methods - Example

- Account class
  - A Class that represents a bank account with basic services
- AccountDriver class
  - A Driver that can be used to test the Account class





in Account.java

A Class that represents a bank account with basic services

```
import java.text.NumberFormat;
public class Account {
    private static final double RATE = 0.035;  // interest 3.5%
    private String name;
    private long acctNumber;
    private double balance;

    public Account (String owner, long account, double initial){
        name = owner;
        acctNumber = account;
        balance = initial;
    }

    public double deposit (double amount){
        if (amount > 0)
            balance = balance + amount;
        return balance;
    }

    public double withdraw (double amount, double fee){
        if (amount + fee > 0 && amount + fee < balance)
            balance = balance - amount - fee;
        return balance;
    }

    public double addInterest (){
        balance += (balance * RATE);
        return balance;
    }
} // end Class
```

Constructor. This one initialises all instance variables

Method. Validate amount to deposit, then deposit it only if valid, return new balance.

Method. Validate amount to withdraw, withdraw it only if valid, return new balance.

Method. Calculate interest and add it, return new balance.

```
public double getBalance (){
    return balance;
}

public String getName (){
    return name;
}
```

Accessors

in AccountDriver.java

```
public class AccountDriver {

    public static void main (String[] args){
        Account acct1 = new Account ("Ted Murphy", 72354, 25.59);
        Account acct2 = new Account ("Angelica Adams", 69713, 500.00);
        Account acct3 = new Account ("Edward Dempsey", 93757, 769.32);

        acct1.deposit(44.10);  // return value ignored

        double adamsBalance = acct2.deposit(75.25);
        System.out.println ("Adam's balance after deposit: " +
                            adamsBalance);

        System.out.println ("Adam's balance after withdrawal: " +
                            acct2.withdraw(480, 1.50));

        acct3.withdraw(-100.00, 1.50);  // invalid – no trans performed

        acct1.addInterest();
        acct2.addInterest();
        acct3.addInterest();

        System.out.println();
        System.out.println(acct1);
        System.out.println(acct2);
        System.out.println(acct3);
    }
}
```

Driver Class to demonstrate the creation and use of Accounts

So it's syntactically valid to do this.

What will these output !!!

toString() method

- All classes, including any we write, automatically include a (hidden) toString() method
  - If we do not include our own version of this method when writing our own classes then the default result it will return is a String with the following format:
    - ClassName@HexadecimalAddressOfObject
- It's conventional to write a toString method for each class you write
  - It should return a String which contains the State of the object (all the object's attribute names and their values) formatted in a way for easy reading
  - If there are many attributes, the returned String should include just the most important/useful attribute values

toString() Method – Example

```
public class Rectangle {
   private int height;
   private int width;

   public Rectangle(int initHeight, int initWidth){
      height = initHeight;
      width = initWidth;
   }

   public int computeArea() {
      return height * width ;
   }

   public int computePerimeter() {
      return  2 * (height + width);
   }

   public String toString(){
      return   "Height = " + height +
               " Width = " + width;
   }
}
```

toString() Method – Another Example

```
import java.text.NumberFormat;
public class Account {
   private static final double RATE = 0.035;  // interest 3.5%
   private String name;
   private long acctNumber;
   private double balance;

   ...

   public String toString()
   {
      return "Account " + acctNumber +
         ", balance: $" + balance +
         ", name: " + name;
   }

} // end Class
```

We only include the instance variables

toString() Method - Invoking

- The toString() method can be invoked explicitly on any object as required
  - Because all objects are instantiated from a class and all classes automatically include a hidden or coded toString() method

- However, you don't need to invoke it explicitly in one of its most common syntax contexts:
  - The following statements are equivalent:
    - `System.out.println(rectangle1);`
    - `System.out.println(rectangle1.toString());`
  - The compiler automatically inserts the call for the first

Static Variables

- Instance Variables
  - Class-level variables (rather than local to a method)
  - Each instance (object) of a class gets an independent set of these variables to store their particular attribute values
- Static variables
  - Class-level variables
  - All instances (objects) of a class share the same variable
    - If one object alters the value, it is changed for all objects, of that class
- Syntax
  - e.g.

Trying to declare these inside a method would be a syntax error

```
public class Person {
    private static int totalPersons;

    private int age;
    private String name;
```

static variable

instance variables

Static Variables - Example

```
public class Person {
    private static int totalPersons;

    private int age;
    private String name;

    public Person(){
        age = 99;
        name = "nobody";

        totalPersons++;
    }

    public String toString(){
        return name + " is " + age + " years old.";
    }

    public static int getTotalPersons(){
        return totalPersons;
    }
}
```

static variable

int variable so auto-initialised to 0 (when?)

Constructor executes every time a person instance is created. Each time the static variable shared by all instances is incremented.

static method (see next slide)

```
public class PersonDriver {

    public static void main(String[] args) {
        Person aPerson = new Person();
        Person bPerson = new Person();
        Person cPerson = new Person();

        System.out.println(
            "Number of persons instantiated = " +
            Person.getTotalPersons());
    }
}
```

static method applied to class NOT instance (object) of class (see next slide)

When this driver code completes there are:
3 instances of Person in memory.
Therefore:
3 instances of the instance variable age (one per Person instance).
3 instances of the instance variable name (one per Person instance).

1 instance of the static variable totalPersons (shared by all instances of Person)

Number of persons instantiated = 3

Static Methods

- Can be invoked on their class's name
  - Therefore do not have to instantiate an object of the class to use a static method
  - This makes for efficient coding and processing
- Cannot set/get instance variable values
  - Which object's instance variables would it set/get???
  - Remember its invoked on the class name not a reference to an object of the class
- Can set/get static variable values
  - There is no ambiguity here since there is only one of each static variable in a class
- So if a method only needs to set/get static variables of a class
  - It's good practice to make it a static method

Static Variables – Scope & Lifetime

- Scope
  (i.e. code that can set and get their values)
  - Same scope as instance variables
  - The entire class (i.e. all class methods)
- Lifetime
  (i.e. time period they exist in memory)
  - Not the same as instance variables since they are independent of instance variables
  - Lifetime starts when the Class is first used for any reason
  - Specifying the end is more complicated but it's certainly equal or later than the end of the lifetime of all the instance variables of all the instances of the Class

Instance and Static Methods

- When you begin Java programming
  - You code the main method in a single class
    - It's a static method because execution needs to begin without instantiating an object of the class to invoke main on
  - Eventually you code more methods in the same single class to be called from code in main
    - These must also be static because you can't call an instance (non-static) method from a static method (i.e. from main)*
- But when we begin writing our own classes
  - We need to code instance (non-static) methods for the classes we write
    - Code from other classes can then instantiate objects of our class and invoke instance (non-static) methods on these objects
  - To code an instance (non-static) method simple omit the static keyword in the method header
  - e.g.

```
public int computeArea() {
    return height * width ;
}
```

static keyword omitted

* Exception
Unless you instantiate an object of the class in the class main (possible but not usual)

Instance and Static Methods

- The following table summarises the relationship between
  - Static and non-static (instance) methods and variables

	Reference an instance variable?	Reference a static variable?	Contain a non-static method call?	Contain a static method call?
Non-Static method	Yes Object selected	Yes Object independent	Yes Object selected	Yes Object independent
Static method	No* Which object's instance variables?	Yes Object independent	No* Which object is the method to invoked on?	Yes Object independent

* Exception: Unless an object of some class is instantiated in the method

Principles for designing classes

Discovering classes

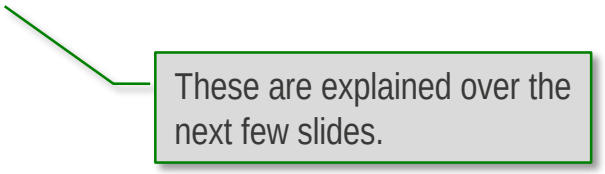
- In Object Orientation, the work will be done by objects in the computer, so:
  - We need to identify what objects are required.
  - We need to assign responsibilities to the various classes
  - We need to work out how the objects will interact to achieve processing tasks
  - We should aim for an easily-understood solution to the problem
    - Helps for debugging, and in team development situations
  - Principles need to be followed to develop a quality solution

Cohesion and Coupling

- **Cohesion** – A measure of how closely related the elements of an encapsulated unit are.
  - Cohesive methods – focus all their energy on just one task
  - Cohesive class – one whose methods are closely related to one goal
- **Coupling** – A measure of how dependent one thing is on another
  - A change in the implementation of one class could impact on how other classes will use it.
  - Methods are coupled if one is called by the other
  - Classes are coupled if one uses the other

Domains of object classes

- Classes should be designed to fit into one particular domain.
- By **domain** we mean an place within a classification scheme
- The four domains of classes are:
  - Foundation domain
  - Architecture domain
  - Business domain
  - Application domain



Domains of object classes (2)

- **Foundation Domain** – classes (and primitive types), which form the basic building blocks from which we can make all other classes.
  Examples:
  - String, int, float, char, boolean.
- **Architecture Domain** – classes which are designed to support a particular computer architecture
  Examples:
  - User-interface classes, such as windows, buttons, text boxes
  - Database access classes, which communicate with a database system to store and retrieve records
  - Networking classes, which communicate data between machines across the Internet

Domains of object classes (3)

- **Business/Problem Domain** – classes which are designed to address the needs of a particular business / industry, such as Banking, Customer Service, Education. They may be suited to a range of different problems within those settings.
- **Application Domain** – classes which are specific to a particular problem and not re-usable in other problems, for example:
  - Driver classes (the class providing the **main()** method)
  - Event Handler classes - classes which respond to specific buttons being clicked on the screen
- We will only be coding for the above two domains

Re-usability of objects – a Holy Grail?

- If an object is designed perfectly, it may be suitable for use in other software. This could save time and errors when developing that other software.
- **Re-usability** is where a class is written once, and used in different programs.
- Foundation domain classes and Architecture domain classes are highly re-usable
- Business domain classes are often re-usable, but to a lesser extent (because only for particular contexts)
- Application domain classes are generally not re-usable at all

The key to developing a re-usable class

- When designing a new class, it should fit in one particular domain.
  - This is achieved by being clear about the responsibility of the objects of that class
- Aim for high cohesion within the class
  - All attributes and methods are related to some common aspect of the system, typically what the class is named
- Aim for low coupling between classes
  - Minimize relationships - Explained further next week

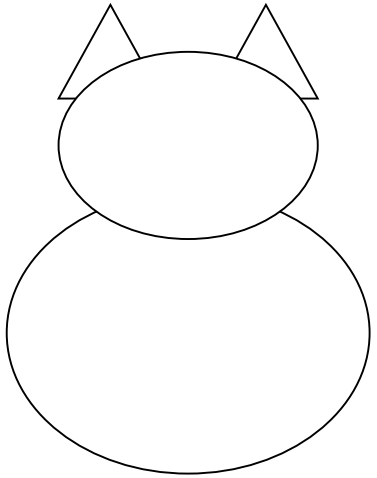
Object Responsibility

- Each object should be responsible for a refined set of tasks that contributes towards solving the problem.

- Example responsibilities that an object could have:
  - To **remember** information
  - To **coordinate groups** of other objects
  - To act as a **central-authority** for messages to other objects
  - To **coordinate the steps** required to achieve a business task
  - To **present information** to the user, or **obtain data** from the user, or files or other locations where the data may reside.

The process and outcome of abstraction

- **Abstraction** is “the process of ignoring details irrelevant to the problem at hand and emphasizing essential ones. To abstract is to disregard certain differentiating details” (Niño, J. & Hosch, F., 2005, p. 6)
- ***An abstraction*** is a description of the essence of the thing: key attributes, and behaviors
- Example: A visual abstraction of a Cat.
  - It just shows the typical shape of all cats.
  - Describes what they “do”
- We can see that a cat has a belly, a face and two ears



Abstraction as a process of design

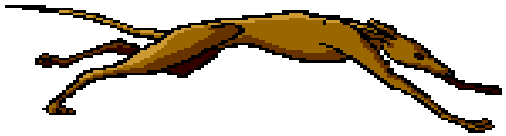
- In OO program design, we perform abstraction to find out what are *the essential*, *common things* about different possible objects.
  - Start by considering the individual actual things of the real world
  - Gradually eliminate the specifics of the individuals, to be left with the essence of them
- We then develop models to represent this remaining information.
  - Use UML Class Diagrams to document it

Abstraction Demonstrated (1)

- We have a set of Dogs and want to abstract their essential features and abilities:

Rusty is red
40 cm long
45 cm high

He likes to run, chase things, and bark



Sprint is brown
1m long and
75 cm high

He likes to run and bark as well

Scott is brown
20cm long and
30 cm high

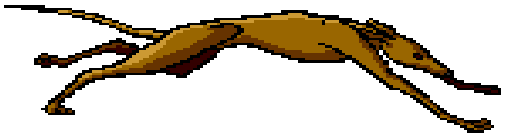
He likes to trot and chase things. He has been known to bark

Abstraction Demonstrated (2)

- Each dog seems to be a different colour. This could indicate that ***colour*** is an essential feature about a dog.

Rusty is red
40 cm long
45 cm high

He likes to run, chase things, and bark



Sprint is brown
1m long and
75 cm high

He likes to run and bark as well

Scott is brown
20cm long and
30 cm high

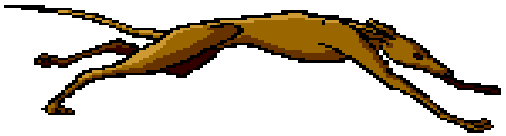
He likes to trot and chase things. He has been known to bark

Abstraction Demonstrated (3)

- Each of the dogs seems to have 2 eyes. This could indicate that ***number of eyes*** is an essential feature about a dog.
- But none of the dogs have more or less than 2 eyes. So our abstraction of a dog can include the fact this is always 2.

Rusty is red
40 cm long
45 cm high

He likes to run, chase things, and bark



Sprint is brown
1m long and
75 cm high

He likes to run and bark as well

Scott is brown
20cm long and
30 cm high

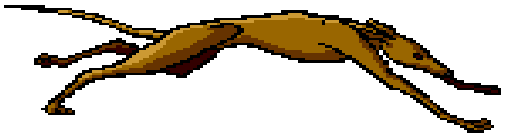
He likes to trot and chase things. He has been known to bark

Abstraction Demonstrated (4)

- Each dog seems to be a different length. This could indicate that ***length*** is an essential feature about a dog, to become an instance variable.

Rusty is red
40 cm long
45 cm high

He likes to run, chase things, and bark



Sprint is brown
1m long and
75 cm high

He likes to run and bark as well

Scott is brown
20cm long and
30 cm high

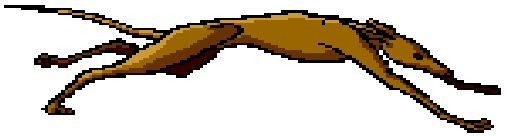
He likes to trot and chase things. He has been known to bark

Abstraction Demonstrated (5)

- Two dogs can run. The other can trot. How should we abstract this behaviour?
- We could abstract it as an ability to ‘move’

Rusty is red
40 cm long
45 cm high

He likes to run,
chase things,
and bark



Sprint is brown
1m long and
75 cm high

He likes to run
and bark as well

Scott is brown
20cm long and
30 cm high

He likes to trot
and chase things.
He has been
known to bark

- What other behaviors or attributes can be in our abstraction?

Abstraction Demonstrated (6)

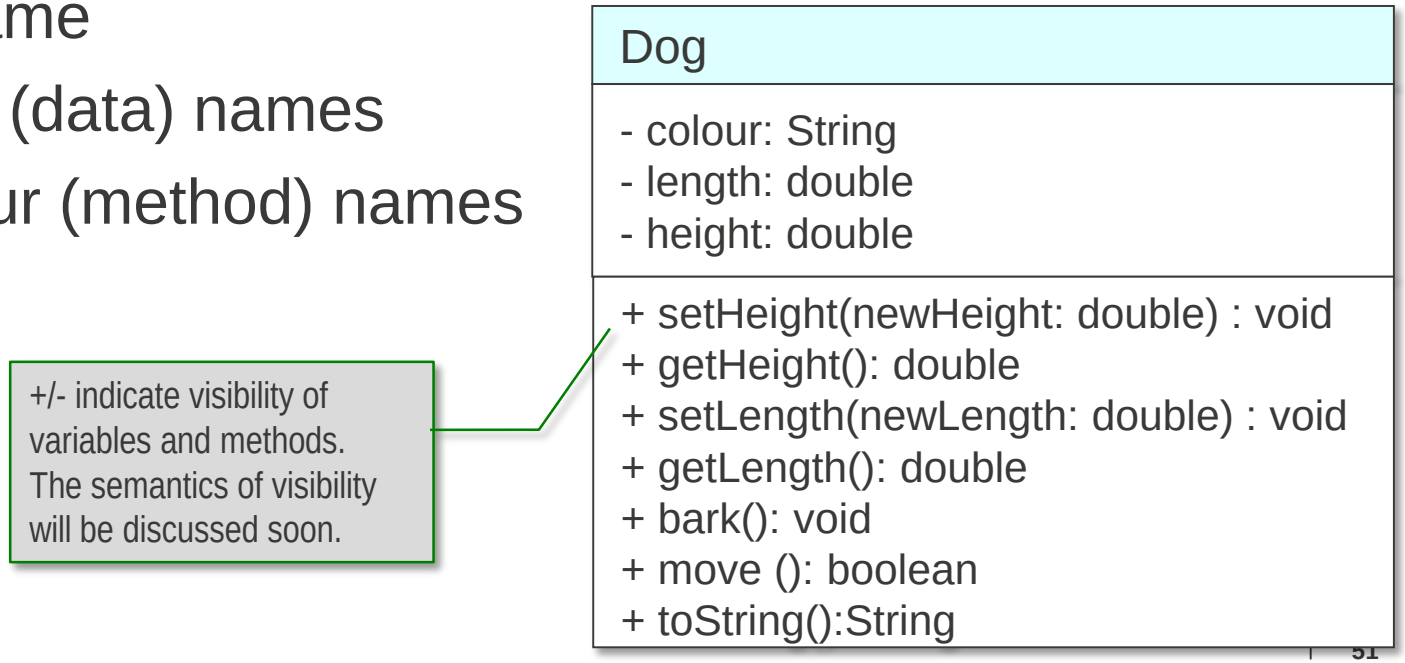
- We can consider all aspects of a dog. Some features will seem more important than others.
- If features are considered important, they need to be included in the abstraction of the dog.
- If features seem irrelevant, they can be excluded from the abstraction of the dog.
- So our abstraction of a dog is all the essential features:
  - A dog has two eyes (always)
  - A dog is with some colour (which may vary)
  - A dog has a length (which may vary)
  - A dog has a height (which may vary)
  - A dog can move (a behavior)
  - A dog can bark (another behaviour)

Modelling classes in OO Systems

- We give a generic name to the thing – *name* for class
- The features which are deemed essential and which may vary in particular *value* become *attributes* of the class
- Features which are essential but which do not vary become *constants*, or may be left out of the system if they are implicit.
- Behaviours of the class become *methods*

Documenting a Model using Class Diagrams

- A Class Diagram shows a graphical summary of key details of the classes that make up a system
  - i.e. name, attributes, behaviours
- In a Class Diagram, a class is represented as a rectangle containing three sections:
  - Class name
  - Attribute (data) names
  - Behaviour (method) names



Determining appropriateness for inclusion

- How do we know which features about an object are appropriate to keep in the abstraction?
- For OO modelling and programming, we only consider those things which have *relevance* in the setting of the computer program
- For example: a system that keeps track of customer orders needs to maintain information about the customers – but which things about customers are relevant from the following list?
  - Name, Age, Address, Place of Birth, Phone number, Mother's name, Father's name, Hair colour, number of children, driver's licence number, credit card number, Weight

Determining appropriateness for inclusion (2)

- Very Relevant to a Customer Ordering system:
  Name, Address, Phone number.

- Not relevant:
  Age, Weight, Place of Birth, Hair colour, Mother's name, Father's name, number of children

- Possibly relevant:
  driver's license number (for security),
  credit card number (for charging)

These may be essential in another system, e.g. the hospital's system.

These may not be relevant in another system, e.g. the hospital's system.

Example of a re-usable Customer class

- Class Name: Customer
- Attributes (fields):
  - Name
  - Address
  - Contact Number
- Operations
  - Create new
  - GetName
  - GetAddress
  - SetNewAddress
  - GetContactNumber
  - SetNewContactNumber

Example of a less re-usable Customer class

- Class Name: Customer
- Attributes (fields):
  - Name
  - Address
  - Contact Number
  - Client Number
  - Accumulated Frequent-Points
  - Monthly Discount Rate
  - WantsToReceiveWeeklyNewsletters
- This class is less re-usable, because some program-specific attributes are present, which may be irrelevant in some other customer management system