CLASS DEFINITIONS

CITS1001 week 2
REVISION OF CONCEPTS FROM WEEK 1

After studying the lectures, lab and reading Chapter 1, you should be familiar with the concepts of class, object, state and method
1. Objects are created by classes
2. Object state is represented by fields
3. Objects (usually) do something when we invoke a method
WEEK 2
Looking inside classes
Concepts (1)

This week we will learn to understand class definitions by looking inside Java classes.

• fields
• constructors
• comments

• Reading: *Chapter 2* of Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling
Classes vs. objects

A *class* is a group of *objects* that have similar characteristics and that exhibit similar behaviour.

An *object* is a specific *instance* of a *class*.

- Classes represent all objects of a certain kind
  - e.g. Car, Lecturer, Student
- Objects represent ‘things’ from the real world, or from some problem domain
  - e.g. the red car down there in the car park
  - e.g. the lecturer talking to you now
  - e.g. you!
Classes vs. objects

- Classes exist at compile-time
  - The source code we write in Java is basically a description of a set of classes
  - You can regard each class as a **design** or a **specification** of objects belonging to that class

- Objects exist at run-time
  - When we start up a program, no objects exist
  - The program creates objects as it runs, and invokes the methods of those objects to enact the behaviour of the program

- This relationship is illustrated in this week’s lab sheet
Why do we use classes?

- To reduce complexity
- Often, we know how to deal with an object based purely on knowing its *class*, without knowing anything specifically about that particular *instance*
- For example, if we encounter a dog – i.e. an instance of the class Dog – we already have a basic understanding of how to deal with it, even if we have never previously met that *particular* dog
  - We know that it might bark, or bite, or wag its tail, based purely on knowing that it is a Dog
  - Barking, biting, and tail-wagging are best viewed as features of the *class* Dog, not of any individual dog
Source code

- In Java, classes are defined by text files of source code
- Source code is designed to be both
  - Human readable, and
  - Machine readable
- Source code must specify everything about how objects belonging to a class behave
  - Computers are very fast but also very stupid!
- In this lecture we will use as a running example the third class in the CITS1001 code listings on the Resources page of the web-site
  - TicketMachine.java (the correct version!)
A very important point

• Program code is designed to be human readable
  • Familiar words are used for programming constructs
    (if, else, while, repeat, for)
  • Indented format is similar to paragraphs and sections in text
  • Meaningful variable names suggest what is intended
    (e.g. price, mark, studentName)

• AND program code is also executed by a computer
  • The computer will do exactly what it is TOLD to do
  • The RULES of the language determine EXACTLY what happens when the program is run

THE COMPUTER DOES NOT KNOW WHAT YOU INTENDED THE PROGRAM TO DO
What is a programming language?

• A program for a computer to follow must be expressed completely unambiguously
• There are many different *programming languages* in which programs can be written
• In order to write a working program, you need to learn
  • the *vocabulary* and *syntax* of the language, so you can write statements that make sense
  • how to make *sequences* of legal statements that do simple tasks
  • how to express what you want the computer to do in a simple enough way to translate into the programming language
• Similar to learning the *words*, how to form *sentences*, and how to *write a story*, in learning a human language
“Programming can be difficult at first. It is annoying when your program doesn’t work, and you spend ages trying to figure our why. Bugs can seem to come from nowhere, for no reason. But there is always a logical reason behind a bug. It is incredibly satisfying when your program does work.”
Ticket machines – an external view

• An external view of a class means considering
  • What objects of the class do
  • How we create and use those objects

• For example, machines accept money, and supply tickets at a fixed price
  • How is that price determined?
  • How is ‘money’ entered into a machine?
  • How does a machine keep track of the money that has been entered?

• This is the view relevant to the user of a class
Ticket machines – an internal view

• An internal view of a class means also considering
  • How it stores information
  • How it does things
• Looking inside allows us to determine how behaviour is provided or implemented
• This is the view relevant to the writer of a class
• All Java classes have a consistent internal view
The four components of a class

- A class definition has four components
  - Its **name** – what is the class called?
  - Its **fields** – what information do we hold for each object, and how is it represented?
  - Its **constructors** – how are objects created?
  - Its **methods** – what can objects do, and how do they do it?

- It is (usually) easiest to consider the four components in this order, whether you are writing your own class, or reading someone else’s
Basic class structure

public class ClassName
{
    Fields
    Constructors
    Methods
}

Reserved words

Curly brackets

The inner contents (the body) of the class
Keywords

• Words with a special meaning in the language:
  • public
  • class
  • private
  • int

• Also known as reserved words.
Syntax

• Reserved words and curly brackets are our first encounter with **Java syntax**

• Source code must be structured in a certain way, as determined by the rules of the language

• Reserved words are words with a special meaning in Java
  • e.g. `public`, `class`, `private`, `int`
  • There are many, many others
  • Also known as **keywords**

• Brackets (of all types) are everywhere in many languages
  • Here, they delimit the contents of the given class
Fields

- Fields store values for an object.
- They are also known as instance variables.
- Fields define the state of an object.
- Use `Inspect` to view the state.
- Some values change often.
- Some change rarely (or not at all).

```java
public class TicketMachine {
    private int price;
    private int balance;
    private int total;

    Further details omitted.
}
```

- visibility modifier
- type
- variable name

private int price;
The fields of TicketMachine

```java
private int price;
private int balance;
private int total;
```

• Each field is described by a **variable**, which has
  • A **visibility modifier**, which denotes who can access it (more later)
  • A **type**, which denotes what values it can store (more later)
  • A **name**, chosen to make its use clear to human readers
• Additionally, and crucially, each field has a **meaning**
  • A sense of what information it stores
  • This should apply to every variable in every program you ever write
• Collectively, the fields denote the **state** of an object
Exercise 2.12 What do you think is the type of each of the following fields?

private int count;
private Student representative;
private Server host;

Exercise 2.13 What are the names of the following fields?

private boolean alive;
private Person tutor;
private Game game;

Exercise 2.14 From what you know about the naming conventions for classes, which of the type names in Exercises 2.12 and 2.13 would you say are class names?
Comments

• The other thing you will see in the source file TicketMachine.java is comments

• Comments are ignored by the computer; they exist simply to make the code easier for people to understand

• Comments come in three principal types

• Comments starting with //
  • In this case, the computer ignores everything up to the end of the line

• Comments starting with /*
  • In this case, the computer ignores everything up to the first occurrence of */ , which acts like a closing bracket for the comment

• Javadoc comments start with /** and end with */
  • We will discuss these later in the unit
Constructors

public TicketMachine(int cost) {
    price = cost;
    balance = 0;
    total = 0;
}

• Initialize an object.
• Have the same name as their class.
• Close association with the fields.
• Store initial values into the fields.
• External parameter values for this.
• Constructors initialise an object.
Constructors

- The principal job of a constructor is to initialise the fields of the object
- Initial values may be
  - Set as **defaults** (e.g. balance, total), or
  - Derived from data passed in as **parameters** (e.g. price)
- Syntactically, the constructor is a special method
  - It has the same name as the class
  - It has no return type
- Note that there may be more than one constructor
**Exercise 2.18** To what class does the following constructor belong?

```java
public Student(String name)
```

**Exercise 2.19** How many parameters does the following constructor have, and what are their types?

```java
public Book(String title, double price)
```

**Exercise 2.20** Can you guess what types some of the `Book` class's fields might be, from the parameters in its constructor? Can you assume anything about the names of its fields?
Aside: Default initialisation

- In Java, all fields are automatically initialised to a default value if they are not explicitly initialised.
- For integer fields, this default value is zero.
- However, we prefer to write the explicit assignments anyway.
- There is no disadvantage, and it serves to document what is actually happening.
Concepts (2)

- Parameters
- Scope of a variable
- Lifetime of a variable
- Assignment statements
Passing data via parameters

Parameters are used by constructors and methods to receive values from outside. Parameters are another sort of variable.
Parameters

- **Parameter names** inside a constructor or method are called **formal parameters**

- **Parameter values** outside are called **actual parameters**

- So `cost` is a formal parameter, and a user-supplied value such as `500` is an actual parameter

**Scope**
- The scope of a variable defines the section of source code from which the variable can be accessed.

**Lifetime**
- The lifetime of a variable describes how long the variable continues to exist before it is destroyed.
Parameter vs Fields

• The **scope** of a **formal parameter** is restricted to the body of the constructor or method in which it is declared.

• The **scope** of a **field** is the whole of the class definition – it can be accessed from anywhere in the same class.

• The **lifetime** of a **formal parameter** is limited to a single call of the constructor or method.

• The **lifetime** of a **field** is the same as the lifetime of the object it belongs to.

Examples: see TicketMachine code
Choosing variable names

• There is a lot of freedom over choice of names. Use it wisely!
• Choose expressive names to make code easier to understand:
  • price, amount, name, age, etc.
• Avoid single-letter or cryptic names:
  • w, t5, xyz123
Assignment

• Values are stored into fields (and other variables) via assignment statements:

\[
\text{variable} = \text{expression;}
\]
\[
\text{price} = \text{cost;}
\]

• A variable stores a single value, so any previous value is lost.
Exercise 2.21  Suppose that the class Pet has a field called name that is of the type String. Write an assignment statement in the body of the following constructor so that the name field will be initialized with the value of the constructor’s parameter.

    public Pet(String petsName)
    {
    }

Exercise 2.22  Challenge exercise The following object creation will result in the constructor of the Date class being called. Can you write the constructor’s header?

    new Date("March", 23, 1861)

Try to give meaningful names to the parameters.
Concepts (3)

- Methods
- Accessor methods
- Mutator methods
Methods

• Methods implement the behaviour of objects
• Methods have a consistent structure comprising
  • a header, and
  • a body
• Methods can implement any form of behaviour, as required by the class being implemented
Method structure

• The header provides the method’s signature:
  • `public int getPrice()`

• The header tells us:
  • the name of the method
  • what parameters it takes
  • whether it returns a result
  • its visibility to objects of other classes

• The body encloses the method’s statements.
Accessor (get) methods

```java
public int getPrice()
{
    return price;
}
```
Accessor methods

- An accessor method returns a value (*result*) of the type given in the header.
- Usually it just looks up the current value of one of the object’s fields.
  - Sometimes it does some minor calculation on that value.
- An accessor method always has a return type that is not *void*.
- The method will contain a *return* statement to return the value.
  - NB: returning is *not* printing!
Mutator methods

```java
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

- visibility modifier: `public`
- return type: `void`
- method name: `insertMoney`
- parameter list: `(int amount)`
- field being mutated: `balance`
- assignment statement: `balance = balance + amount;`
Mutator methods

- They have the same method structure
  - Header and body
- They are used to *mutate* (i.e. change) an object’s state
  - Achieved through changing the value of one or more fields
- They usually have the return type `void`
- They typically contain assignment statements
- They often receive data through parameters
set mutator methods

• Fields often have dedicated set mutator methods
• These have a simple, distinctive form
  • void return type
  • method name related to the field name
  • single parameter, with the same type as the type of the field
  • a single assignment statement
A typical set method

```java
public void setDiscount(int amount) {
    discount = amount;
}
```

- We can infer from this that `discount` is a field of type `int`, i.e.

```java
private int discount;
```
Protective mutators

- A set method does not have to simply assign the parameter to the field
- The parameter may be checked for validity and rejected if inappropriate
- Mutators thereby protect fields
- Mutators support *encapsulation*
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
String concatenation

- $4 + 5$
  - 9
- "wind" + "ow"
  - "window"
- "Result: " + 6
  - "Result: 6"
- "# " + price + " cents"
  - "# 500 cents"
Quiz

- System.out.println(5 + 6 + "hello");

- System.out.println("hello" + 5 + 6);
Exercise 2.31  How can we tell from just its header that `setPrice` is a method and not a constructor?

    public void setPrice(int cost)

Exercise 2.32  Complete the body of the `setPrice` method so that it assigns the value of its parameter to the `price` field.

Exercise 2.33  Complete the body of the following method, whose purpose is to add the value of its parameter to a field named `score`.

    
    
    /**
     * Increase score by the given number of points.
     */
    public void increase(int points)
    {
        ...
    }
Method summary

• Methods implement all object behaviour
• A method has a name and a return-type
  • The return-type may be `void`
  • A non-`void` return type means the method returns a value to its caller
• A method might take parameters
  • Parameters bring values in from outside for the method to use
• *Accessor methods* provide information about an object
• *Mutator methods* alter the state of an object
• Other sorts of methods can accomplish a variety of tasks
Concepts (4)

- conditional statements
- string concatenation
- local variables
Reflecting on the ticket machines

• Their behavior is inadequate in several ways:
  • No checks on the amounts entered.
  • No refunds.
  • No checks for a sensible initialization.
• How can we do better?
  • We need more sophisticated behavior.
Making choices in everyday life

- If I have enough money left, I will go out for a meal
- Otherwise, I will stay home and watch a movie

```java
if(I have enough money left) {
    // go out for a meal;
}
else {
    // stay home and watch a movie;
}
```

- The result depends on the amount of money available at the time the decision is made
Making choices in Java

```java
if (perform some test) {
    Do these statements if the test gave a true result
} else {
    Do these statements if the test gave a false result
}
```

- **‘if’ keyword**
- **‘else’ keyword**
- **boolean condition to be tested**
- **actions to perform if condition is true**
- **actions to perform if condition is false**
public void insertMoney(int amount) {
    if(amount > 0) {
        balance = balance + amount;
    }
    else {
        System.out.println(  
            "Use a positive amount: " +  
            amount);
    }
}
```java
class TicketMachine {

    public void printTicket()
    {
        if(balance >= price) {
            System.out.println("#
```
Exercise 2.52  After a ticket has been printed, could the value in the **balance** field ever be set to a negative value by subtracting **price** from it? Justify your answer.

Exercise 2.53  So far, we have introduced you to two arithmetic operators, + and −, that can be used in arithmetic expressions in Java. Take a look at Appendix C to find out what other operators are available.

Exercise 2.54  Write an assignment statement that will store the result of multiplying two variables, **price** and **discount**, into a third variable, **saving**.

Exercise 2.55  Write an assignment statement that will divide the value in **total** by the value in **count** and store the result in **mean**.

Exercise 2.56  Write an if-statement that will compare the value in **price** against the value in **budget**. If **price** is greater than **budget**, then print the message “Too expensive”; otherwise print the message “Just right”.

Exercise 2.57  Modify your answer to the previous exercise so that the message includes the value of your budget if the price is too high.
Variables – a recap

- Fields are one sort of variable:
  - They store values through the life of an object.
  - They are accessible throughout the class.

- Parameters are another sort of variable:
  - They receive values from outside the method.
  - They help a method complete its task.
  - Each call to the method receives a fresh set of values.
  - Parameter values are short lived.
Local variables

• Methods can define their own, *local* variables:
  • Short lived, like parameters.
  • The method sets their values – unlike parameters, they do not receive external values.
  • Used for ‘temporary’ calculation and storage.
  • They exist only as long as the method is being executed.
  • They are only accessible from within the method.
Scope highlighting

```java
/**
 * Print a ticket if enough money has been inserted, and
 * reduce the current balance by the ticket price. Print
 * an error message if more money is required.
 */
public void printTicket()
{
    if(balance >= price) {
        // Simulate the printing of a ticket
        System.out.println("#" * 10);
        System.out.println("# The Blue Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("#" * 10);
        System.out.println();

        // Update the total collected with the price.
        total = total + price;
        // Reduce the balance by the price.
        balance = balance - price;
    }
    else {
        System.out.println("You must insert at least: " +
                        (price - balance) + " more cents.");
    }
}
/**
 * Return the money in the balance.
 * The balance is cleared.
 */
```
Scope and lifetime

- Each block defines a new scope.
  - Class, method and statement.

- Scopes may be nested:
  - statement block inside another block inside a method body inside a class body.

- Scope is static (textual).

- Lifetime is dynamic (runtime).
Local variables

```java
public int refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```
Local Variables’ scope and lifetime

• The scope of a local variable is the block in which it is declared.
• The lifetime of a local variable is the time of execution of the block in which it is declared.
• The scope of a field is its whole class.
• The lifetime of a field is the lifetime of its containing object.
How do we write 'refundBalance'?

- Return all the money left in the machine (balance) to the customer.
- And clear the balance to 0
refundBalance method

```java
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```
The Debugger
The debugger

• Useful for gaining insights into program behavior ...
• ... whether or not there is a program error.
• Set breakpoints.
• Examine variables.
• Step through code.

Watch this introductory video:
Using the Debugger in BlueJ with Java
https://www.youtube.com/watch?v=w_iy0jmMmkA
The debugger
Review (1)

- Class bodies contain fields, constructors, methods and comments.

- **Field**
  - Fields store data for an object to use. Fields are also known as instance variables.

- **Constructors**
  - Constructors allow each object to be set up properly when it is first created.

- **Comment**
  - Comments are inserted into source code to help human readers. They have no effect on the functionality of the code.
Review (2)

- **Scope**
  - The scope of a variable defines the section of source code from which the variable can be accessed.

- **Lifetime**
  - The lifetime of a variable describes how long the variable continues to exist before it is destroyed.

- **Assignment**
  - Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.
• Methods
  • Methods implement the behavior of objects.

• Accessor methods
  • Accessor methods return information about the state of an object.

• Mutator method
  • Mutator methods change the state of an object.

• println
  • The method `System.out.println` prints its parameter to the text terminal.
Review (4)

- **Conditional**
  - A conditional statement takes one of two possible actions based upon the result of a test.

- **Boolean expression**
  - Boolean expressions have only two possible values: true and false. They are commonly found controlling the choice between the two paths through a conditional statement.

- **Local variables**
  - A local variable is a variable declared and used within a single method. Its scope and lifetime are limited to that of the method.

- **Debugger**
  - A debugger is a SW tool that helps in examining how an application executes. It can be used to help find bugs.