MONASH University

Information Technology

Module 3:
Modularity with Methods and Classes

FIT2034 Computer Programming 2
Faculty of Information Technology

Part 1:

Methods

Modularisation – Why?

- Real programs are commonly thousands of lines long
  - Programmers need to be able to create, understand, change, test and debug these programs
  - To do these tasks quickly and successfully programmers need to be able to
    - Navigate the code (e.g. to make changes or find errors)
    - Focus on a small section of code without having to worry about its effect on the rest of the program
- Alternatives
  - A single monolithic block of code
    - Data is Global, any statement can potentially access any data
  - A hierarchy of named small modules that call (execute) each other and pass data between themselves
    - Data is local, each module has its own data which no other module can access

Modularisation – Why?

- Monolithic
  - Navigation?
    - There are no navigation aids
  - Data Scoping?
    - Since any statement can potentially access any data, to safely create/change any statement all the other statements (possible thousands) must be understood
- Modularised
  - Navigation?
    - The module hierarchy can be quickly navigated using the names of the modules from top level modules that perform general tasks to lower level modules that perform more specific tasks
  - Data Scoping?
    - The code of a module (on average around 10-20 lines of code) can be created/maintained in relative isolation

Modules and Modularisation

- Module
  - Any unit which is both small enough and large enough to be self-contained and useful
- Modularisation
  - Breaking-down (decomposing) logic and therefore the program that implements the logic into modules
- OOP - Two levels of Modularisation:
  - Classes (later)
  - Methods (next slide)
    - Methods are contained within Classes

Classes and Methods
Several methods often perform a set of tasks related by the fact that they manipulate the same data set in various ways e.g. open account, close account, deposit to account withdraw from account. The data set here is all the data associated with an account. The account data and the methods which manipulate it can be bundled together to create a self-contained Class

Methods

- In Java, code Modules are called Methods
- Method
  - Named self-contained block of statements
  - Should perform a single, coherent task at some level of detail
  - Can be called (executed) from the code of other Methods which allows
    - A method call hierarchy to be built
    - Elimination of code duplication
      - Duplicate code should really have been written once and called multiple times as required
  - Allows for a form of Abstraction (see next slide)

Duplicate code is both inefficient to create and maintain and an accident waiting to happen if maintained inconsistently

Abstraction

- Abstraction is:
  - the process of "identifying essential characteristics of a thing … and omitting details that are unimportant from a certain viewpoint"
    (J. Rumbaugh, et al.)
  - "The process of ignoring details irrelevant to the problem at hand and emphasizing essential ones.
    To abstract is to disregard certain differentiating details"
    (J. Niño & F. Hosch)
- Methods allow you to write code by thinking in terms of larger/broader tasks to be done without worrying about the details of those tasks as you code these larger/broader tasks
  - Subsequently Methods can be coded to perform the details of each of the larger/broader tasks

Abstraction: Classes and Methods

- Classes and Abstraction
  - Normally abstraction is used to describe an aspect of *classes*
    - A Class abstracts the essential attributes and behaviours of a class of real world objects essential to their use in the particular piece of software being created
    - e.g. the essentials of a car in a racing game and a crash simulator are very different.
- Methods and Abstraction
  - Here, however we are talking about abstracting the essence of a task by coding it in a method and giving the method a name
    - Now we just use the name to perform the task and forget about the details of how the task is performed
    - This allows us to incorporate it and other similarly abstracted tasks into a larger task without becoming overwhelmed with details

Modularisation - Example

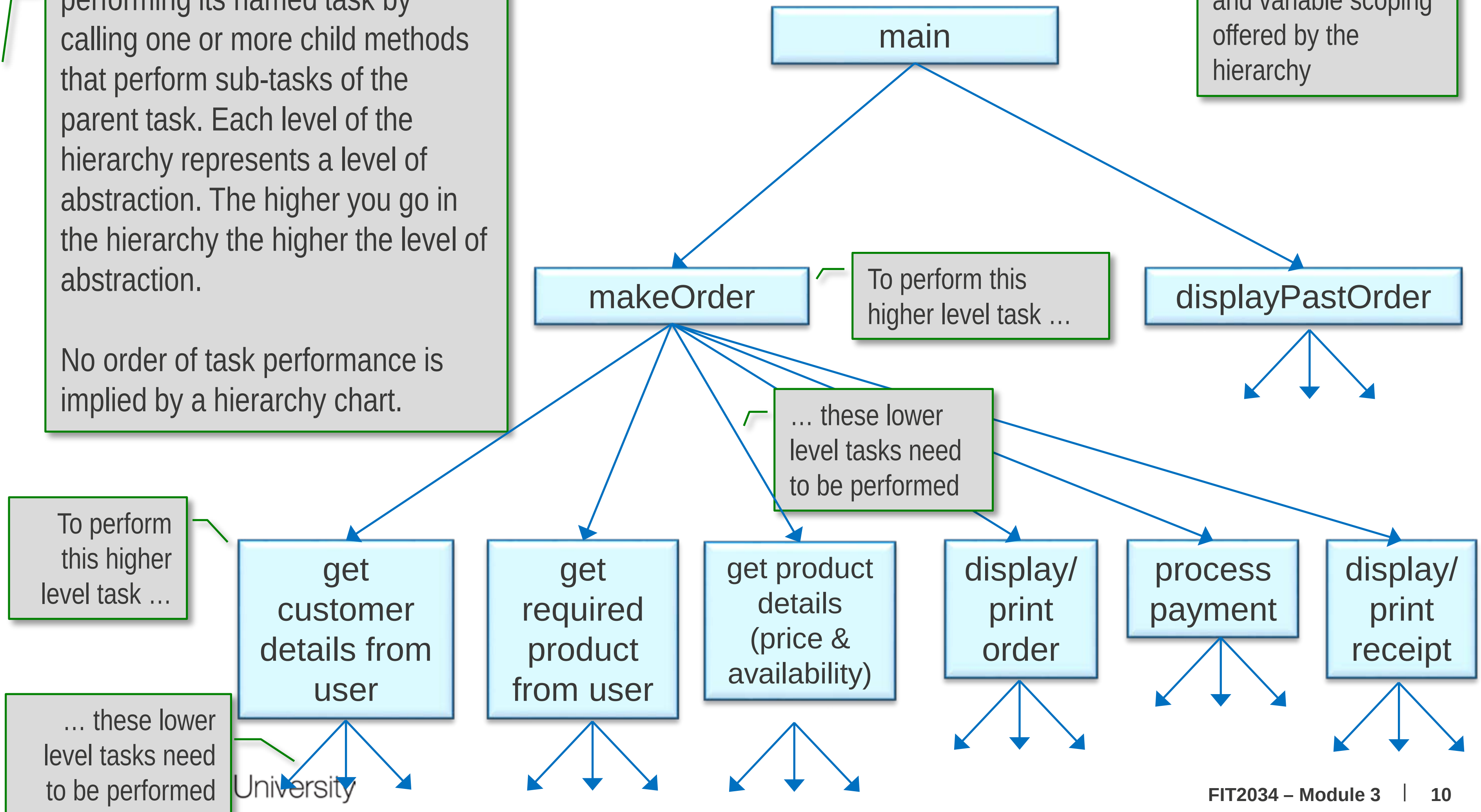
- Consider the requirements:
  - A system is required to keep track of orders made by customers for a retail company
  - The system should capture details of orders including customer details, products ordered, and fulfillment status
  - The system should calculate prices based on stored prices
  - Payments must be processed before fulfillment can proceed
  - The system should generate receipts
  - The system should display the details of all orders made to date
  - The system will be controlled by means of a menu that lists possible actions for the user

Method Hierarchy - Example

The STRUCTURE is hierarchical. Implemented by a parent method performing its named task by calling one or more child methods that perform sub-tasks of the parent task. Each level of the hierarchy represents a level of abstraction. The higher you go in the hierarchy the higher the level of abstraction.

No order of task performance is implied by a hierarchy chart.

Note the possibilities for code navigation and variable scoping offered by the hierarchy



Methods and Abstraction

- A Method's name
  - Should describe what the method will do
  - This is very important for navigating the Method hierarchy
- Examples
  - getProduct
    - A method to get (from the user) the product and quantity they wish to order
  - getProductDetails
    - A Method to get product price and stock levels from a database
  - determineCustomerDiscount
    - Called from getProductPrice which is called from getProductDetails assuming the discount varies depending on the customer and the product
    - This is a sub-sub-task of getProductDetails

Designing a Method

- Focus on the primary responsibility of the method
- It often helps to think about the following aspects of the Method
  - Input
    - What data does the Method require to perform its task
    - Could come from a user or a Method that called this Method
  - Processing
    - What processing must the Method perform to complete its task
    - Often (but not always) involves processing input data to create output data
  - Output
    - What data should be returned by this Method (if any)
    - Could be to a user or a Method that called this Method
  - Coordination
    - What Methods should this Method call to perform the details of its task

Designing a Method - Pseudocode

Method's name textually
Delimits start of method

```
MakeOrder
    total = 0
    Call Input customer details

    While there are more products to purchase
        Call Input next product
        Call Get product details
        total = total + price

    Call Display Order
    Call Accept payment
    Call Produce receipt
End
```

This indicates a call to another
Method to perform a sub task

Note:
How we do NOT concern ourselves with lower level detail which improves our concentration when getting this level's logic right

Textually delimits
end of method

This is pseudocode (not Java code)
Remember this is structured English with a relaxed syntax designed to help us focus on logic not Java language features and syntax.

Notice the appearance of control structures not present in the hierarchy chart

Java Method - Example

- Actually we have already seen a Method in Java code
  - The main method
    - This is a special method for Java because there is only ever one method called main in a Java program (application) and the Java virtual machine begins execution by executing this method
- Here is an example of a Java method other than main
  - What are the similarities and differences between main and this method (Name? Parameters?)

```
public static void introduction(){
    String name = "Thomas";
    int age = 40;

    System.out.println( "My name is " + name +
                        " and I am " + age +
                        " years old.");
}
```

```
public static void main(String[] args) {

}
```

Java Method - Syntax

- Here is a common syntax for a Method
  - There is a method header containing:
    - The method name
    - A specification of input data (in a parameter list – see later)
    - Some “plumbing” which we do not focus on now
  - There is a method body
    - Enclosed in braces (curly brackets)
    - Containing Java statements to perform the method’s task

For now, this is “required plumbing”. You need to code it but not understand it at this stage.

Input data the method requires to perform its task

method body starts

```
public static void methodName(parameterList){
   methodBody
}
```

Statements that perform the method’s task

method body ends

method header

Java Method Calls - Invocation

- A method is always called by some other method
  - i.e. by the code of some other method
  - Thus the calling method calls the called method
- We say a method call has occurred in the calling method
  - There are two distinct mechanisms which will be described soon
- Invocation
  - Invocation = call, invoked = called
- Flow of Control
  - The calling method suspends its execution at time of call until the called method has started and finished executing

Java Method Call - Example

```java
public static void main(String[] args){
 1 System.out.println("Before Call to introduction()");

 2 introduction();

 6 System.out.println("After Call to introduction()");
}
```

method call

Flow of control

1 ------> 6

1 Call

1 Non-executable

Before Call to introduction()
My name is Thomas and I am 40 years old.
After Call to introduction()

Note: it's clear from this output that at the time of call main(…) suspends execution, introduction() then starts and finishes execution, then main(…) resumes execution and finishes execution.

```java
public static void introduction(){
 3 String name = "Thomas";
 4 int age = 40;

 5 System.out.println( "My name is " + name +
                      " and I am " + age +
                      " years old.");
}
```

Declaration are not usually regarded as executable

Java Methods – Returning Data

- Methods - Data In
  - Via parameters (soon)
- Methods - Data Out
  - Often (but not always) methods need to return a value usually as a result of their processing
    - e.g. the result of a calculation they perform based on their input data
    - e.g. a boolean true or false depending on the success or failure of the method's task (so the calling method can react appropriately)
- How?
  - It's easy enough to calculate some value in a called method but how is that value to be returned to the calling method?
  - Look at the previous slide
    - If the method introduction() returned a value how could that value be recovered in the calling method (main in this case)?

Java Methods – Returning Data

- Syntax Notes
  - In the called Method
    - The method header replaces the Java keyword void with the type of the value to returned
      - So void indicates that a method has no return value
    - A return statement specifies the value to be returned
      - It should be the same type as that promised in the method header
  - In the calling Method
    - The returning value replaces the method call
    - For this value to be used in must appear in an expression
      - Not as a statement containing just the method call

Java Methods – Returning Data

```
public static void main(String[] args){
 1 String name;

2 7 name = introduction();
  8 System.out.println("Hello " + name);
}
```

A method call embedded in a very simple, single term String expression.
The method executes and its call is replaced by its return value when the call is encountered during normal expression evaluation.

Please enter your name
David
Hello David

Specifies type of value being returned

```
public static String introduction(){
 3 String myName = "test";
 4 Scanner console = new Scanner(System.in);

 5 System.out.println("Please enter your name");
 6 myName= console.next();
   return myName;
}
```

Return value is of type String – same as return type promised in the method header

Flow of control

1 -----> 6

1 Call

1 Non-executable

Returning Data – Flow of Control

- Flow of control - Step 2 and 7 2 7
  - This is more complicated than the case where the method does not return a value
  - If a method call is encountered during the normal evaluation of an expression
    - The evaluation pauses while the called method executes
      - Calling method suspends execution, Called method starts execution
    - Called method calculates a return value and returns it
      - Called method finishes execution, calling method resumes execution
    - The return value replaces the method call
    - Evaluation of the expression continues and completes
    - The statement the expression is in continues and completes
    - The calling method continues with the next statement

Java Methods – Returning Data (version 2)

```
public static void main(String[] args){
1 6 System.out.println("Hello " + introduction());
}
```

Please enter your name
Stephen
Hello Stephen

A method call embedded in a String expression.
The method executes and its call is replaced by its return value when the call is encountered during normal expression evaluation.

Specifies type of value being returned

```
public static String introduction(){
  2 String myName = "test";
  3 Scanner console = new Scanner(System.in);

  4 System.out.println("Please enter your name");
  5 myName= console.next();
    return myName;
}
```

Return value is of type String same as return type promised in the method header

Flow of control

1 - - - - > 6

1 Call

1 Non-executable

Flow of Control

- i.e. the order in which statements are executed
  - *Begins with the first executable statement of the main method*
    - Declarations are not normally regarded as executable
  - Continues sequentially subject to Selection and Repetition control structures and calls to other Methods
  - Method Call
    - When a method call is encountered the flow of control immediately redirects to the first executable statement of the called method (the calling methods suspends execution)
    - The called method executes completely
    - Flow of control returns to the calling method at the point of call
      - Exact flow depends on whether the call was a single statement or embedded in an expression (see previous slides)
    - The calling method continues to execute

Code Modularisation - Example

- Consider the following program
  - We will modularise this program
    - With such a small example this may seem pointless but we will separate two independent tasks implemented as methods both called from main which coordinates their activities
      - Displaying a welcome message
      - Displaying a times table

```
This program displays a times table
Which times table (enter an integer)? 5
1*5=5
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50
```

```java
import java.util.Scanner;

public class Modularisation1{
    public static void main(String[] args){
        int x, cur;
        Scanner scan = new Scanner(System.in);

        System.out.println("This program displays a times table");
        System.out.print("Which times table (enter an integer)? ");
        x = scan.nextInt();

        for (cur = 1; cur <= 10; cur++)
            System.out.println(cur + "*" + x + "=" + (cur * x) );
        }
}
```

Code Modularisation - Example

```
import java.util.Scanner;

public class Modularisation2 {
    public static void main(String[] args){
        welcome();
        timesTable();
    }

    public static void welcome(){
        System.out.println("This program displays a times table");
    }

    public static void timesTable(){
        int x, cur;
        Scanner scan = new Scanner(System.in);

        System.out.print("Which times table (enter an integer)? ");
        x = scan.nextInt();

        for (cur = 1; cur <= 10; cur++)
            System.out.println(cur + "*" + x + "=" + (cur * x) );
    }
}
```

Neither called method has a return value therefore it would not make sense to embed them in an expression. Instead their Call is a statement.

Local Variables

- Variables declared in a Method
  - Are called local variables
  - Their scope (code that can access them) is the method they are declared in
    - This means their values can only be set or changed by the method's code and not by any other method's code
    - This means programmers can create and maintain a method without fear of accidentally interacting with the data of other methods
      - This dramatically increases the efficiency of programmers and lowers their error rates

More precisely from their declaration statement to the end of their method

- e.g.
  - Any variables declared in main are local to main
  - The timesTable() method has 3 local variables

2 slides back

Scope

- A variable's Scope is all the code that can set or get its value
- So far we have seen that local variables have the scope of the method they are declared in from their declaration statement to the end of the method

This has implications for the argument about where variables should be declared. At the top of a method or as required throughout the method. The latter makes for more chaotic scoping.

Parameters

- Parameters allow us to pass data into methods thereby making them much more versatile
  - e.g. which is the more versatile method
    - myAccount.deposit100();
    - myAccount.deposit(amount);

- Parameters allow a calling method to supply a called method with input data

Parameters

- The header of a Method's definition
  - Specifies the number and type of parameter data items the method will accept
  - In a comma separated list enclosed in parenthesis following the Method's name
  - e.g.

```
public static char calc(int num1, int num2, String message){
…
```

FORMAL PARAMETERS LIST

  - This parameter list is called the **Formal Parameter** list and it specifies target variables where the passed parameter values are to be stored

Parameters

- Every invocation (call) of the method calc must include a parameter list that corresponds to the Formal parameter list
  - Same number of parameters with the same type in the same order
  - e.g.

```
c = calc(3, count, "Hello");
```

ACTUAL PARAMETERS LIST

  - This parameter list is called the **Actual Parameters** list and it specifies the actual parameter values, as expressions of the correct type, for this particular call to the method.

Parameters

- At the time of a parameterised method's call
  - Before its statements begin executing
  - A copy of each evaluated actual parameter value is assigned to its corresponding formal parameter variable
    - Formal parameters must be variables (storage locations, destinations for data)
      - In fact they behave just like local variables of their method
    - Actual parameters must be expressions that evaluate to a value of the type of their corresponding formal parameter (sources of data)

Parameter Passing

o

```
public static void main(String[] args) {
    int count = 4;
    char c;

    c = calc(1, count, "Hello");


    System.out.println(c);
}
```

CALLING METHOD

CALL

1

4

Hello

```
Public static char calc(int num1, int num2, String message){
    char retValue;

    if (num1 + num2 <= message.length())
        retValue = message.charAt(num1 + num2 - 1);
    else
        retValue = ' ';

    return retValue;
}
```

CALLED METHOD

This method is just for demonstration purposes. Its functionality could easily be achieved using just charAt(...)

Actual Parameters

- Actual Parameters are expressions
  - e.g. any valid combination of literals, variables, operators AND METHOD CALLS (at least those with return values)
  - They must evaluate to the type of their corresponding formal parameter
- In the following call to calc the actual parameters are more complex expressions

```
int count = 4;
String s = "GOODBYE";
char c;

c = calc(1, (count + 5) / 4, s.substring(2, 5).toLowerCase());
```

b

Simple literal

Expression involving method calls with return values

Expression involving literals, variables and numeric operations

Formal Parameters are Local

- Formal parameters are like automatically declared and initialised local variables
  - They are initialised with their corresponding actual parameter's value
- Scope of local variables

For formal parameters this is the entire method

  - From their declaration to the end of their method
- Lifetime (time period in memory) of local variables
  - Begins when the method starts execution
    - We say they are born
  - Ends when the method finishes execution
    - We say they die (are erased from memory)
  - If a method is called twice the two lifetimes of each of the local variables are completely unrelated
    - No values are remembered between lifetimes

Information Technology

Part 2

Writing Classes

Objectives – Part 2

- Know the basic structure and content of a class definition and be able to create a basic class in Java
- Explain the concepts of attribute and behaviour;
- Explain the difference between an object and a class
- Understand and use Instance Variables
- Explain and be able to code:
  - Class Constructors
  - Method invocations
  - Test Driver classes

Why Write Your Own Classes?

- Why Object-Oriented Programming?
  - It has been found that representing real world objects and concepts as classes in programs produces applications that are dramatically easier to create, understand, test, debug and change
    - In addition these Classes can be reused in new applications
  - Why? Classes enclose data and the code that manipulates that data. They are context-free, therefore they can be created and maintained in isolation without having to consider the rest of an application’s code and reused in new contexts in new applications

Objects

- An object:
  - Is an “instance” of a Class
    - e.g. an actual student is an instance of the class of Student
    - Any number of student instances can be created from this Student class to represent all the students taking FIT1002 for instance
  - The class is a recipe of attributes (data, not data values) and behaviours (methods) that each instance of the class is created with
  - Attributes
    - Each instance of a class has the same list of attributes but can have different values for those attributes
  - Behaviours
    - Each instance of a class has the same list of behaviours but these behaviours are invoked on particular instances of the class as required

e.g. you all have names (an attribute) but not the same name (attribute value)

e.g. you can all yawn but only a few of you are ☺

Examples of Objects

Many instances of the coin class

- Here are pictures of three objects (instances) of three different Classes
  - Dog, Clock, and Coin
- What attributes (data) might a Dog class have? A Clock class? A Coin class?
  - Does this depend on the application the class is to be used in?

Example of Objects

- Here are pictures of three distinct, but similar, objects
  - Could they all belong to just one Clock class?
- Can you think of some common attributes?
  - What about an attribute that is not common?
- Can you think of some common behaviours?
  - What about a behaviour that is not common?
- During OO Design of an application
  - We identify the real world objects we wish to represent in the application
  - Then we Abstract the common attributes and behaviours (relevant to the application) of similar objects to create classes

sample answers

Objects - Examples

- a Clock:
  - Attributes: hours, minutes, seconds
  - Behaviours: setTime, displayTime, …
- a Student:
  - Attributes: name, address, ID, degree
  - Behaviours: changeAddress, enrollInUnit, …
- a Teacher:
  - Attributes: name, address, ID, position
  - Behaviours: changeAddress, assignUnitToTeach, …
- a Coin:
  - Attributes: currency type, value, yearMinted
  - Behaviours: flip, getUppermostFace, …

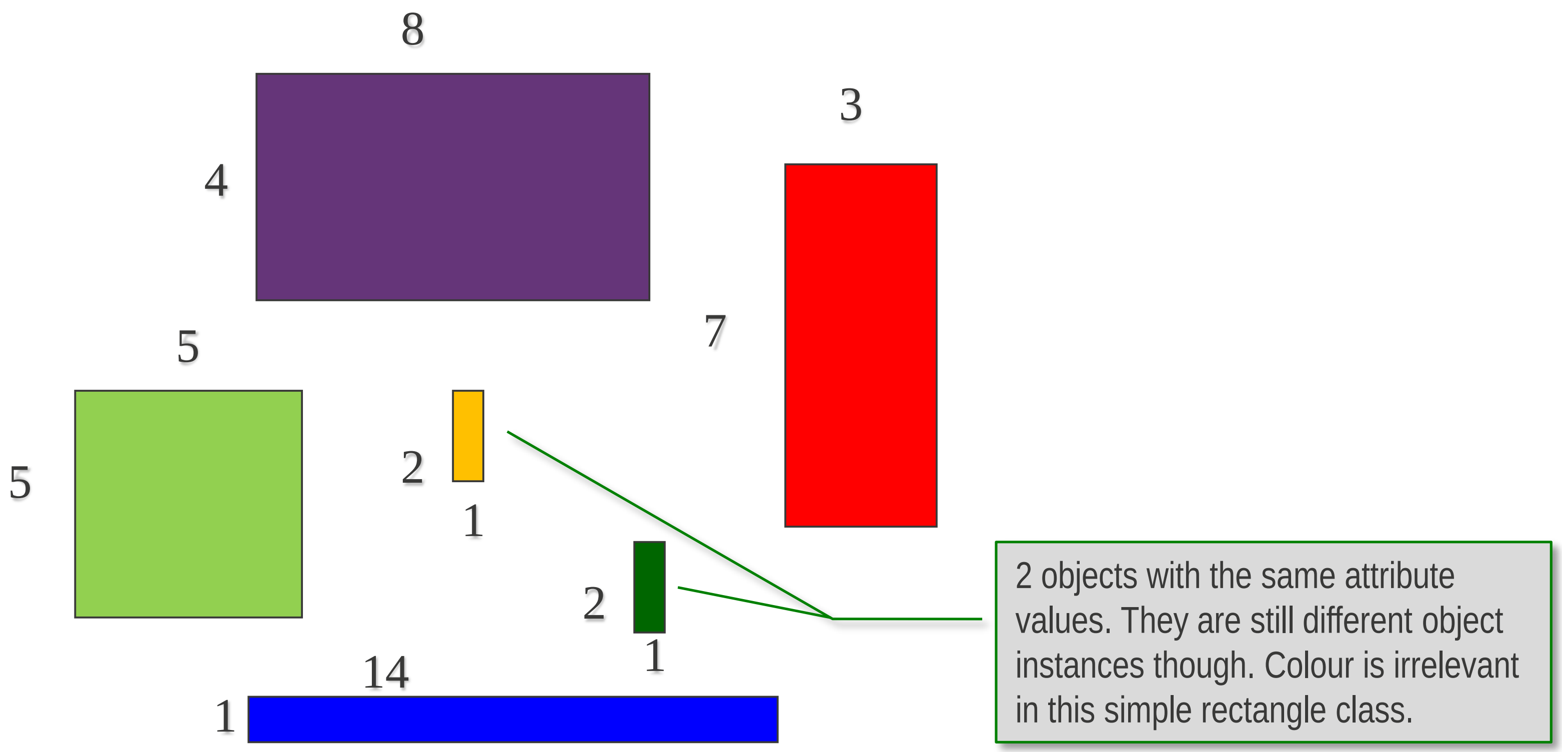
In addition to the example behaviours listed here there are usually 2 additional behaviours for each attribute: setAttributeValue and getAttributeValue.
The attributes values should not be changed directly but only through these behaviours which protect the attribute values as required.
Much more on this later.

Classes vs. Objects

- Classes are a recipe for creating (instantiating) objects (instances) of the class
  - The recipe is a list (template) of attributes (data not data values) and behaviours (methods) each instance of the class must have
- Objects instantiated from the same class
  - Have the same list of attributes
    - But independent attribute values
  - Have the same list of behaviours
    - But perform these behaviours independently as required
- Classes are designed by *abstracting* common, relevant features of similar, real life objects
  - They are then used, in Java, as a recipe to instantiate program versions of these objects

Classes vs. Objects

- e.g. instances of the class Rectangle:
  - Many instances, but all instantiated from one class
  - We choose to make this a very simple class with just 2 attributes, height and width (not position or colour)



Writing Classes in Java

- To write a Java class you must:
  - Choose a name for the class
    - It should be a <u>singular</u> noun that describes what objects of the class are
  - Declare attribute variables to store the data objects of the class must have
    - There may also be other private variables that support the workings of the class
  - Write methods to implement the behaviours of objects of the class
    - These are the called the methods of the “public interface”
    - There may also be other (private) methods that support the public methods (i.e. are called from them)
      - (We explain private methods next week)

Classes in Java - Syntax

- Syntax

We do not know the meaning of 'public'. So for now it's treated as plumbing.

Class definition begins here

```
public class className{
   constant declarations

   variable declarations

   methodDefinitions
}
```

Class definition ends here

Classes in Java – Example complete class

It's a common style rule to begin class names with an uppercase letter to distinguish then from variable and methods names which commonly begin with a lowercase letter.

```
public class Rectangle {

   private int height;
   private int width;

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

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

This is how attribute variables are declared.
Outside of any method with the private visibility modifier.
These "Class-level" variables are called Instance Variables because each instance (object) of the rectangle class will have its own independent set of these to hold its attribute values.

These two methods are both public and are therefore part of the class's interface.
These particular methods are quite passive and do not change any Instance variable values

Rectangle Class - Notes

- Note the following:
  - No main() method
  - No `static` after `public` in method headers
  - Variables height and width are declared just once, and not in any method, but are used in multiple methods (scope of instance variables?)
  - height and width use the Java visibility modifier keyword `private` before their data type in their declaration
- Most of the above will be explained in this module
  - Others explained in next module

Declaring Attributes

- In the Rectangle class two attribute variables are declared
  - height
  - width
- These are declared inside the class but outside of any method
  - The scope of such variables is the entire class
- The code of all methods in the class will be able to set and get the values of these attribute variables (but this doesn't mean they should!)
- We call such variables **Instance Variables**
  - Instance variables and any other variables declared outside any methods are called class-level variables for obvious reasons

Scope and Lifetime

- For Instance Variables
  - Scope (i.e. code that can set and get their values)
    - The entire class they are declared in
  - Lifetime (i.e. time period they exist in memory)
    - From the moment their object is created until it is removed from memory
    - Remember there is a set of instance variables for each instantiated object
    - Methods of the class can be invoked on an object one or more times but the lifetime of instance variables of the this object span (extend over) these executions
- For Method (Local) Variables
  - Scope
    - From their declaration to the end of the method they were declared in
  - Lifetime
    - Their method’s execution time
    - This cannot span method calls (even multiple calls to the same method)

Declaring Object Reference Variable

- Lets declare some variables that can refer to (point at) Rectangle objects
  - Note we are creating Rectangle reference variables NOT Rectangle objects
- Why?
  - Because, to get any rectangle-related work done, we need to invoke Rectangle methods on specific Rectangle objects
  - Syntax is:
- Declaring Rectangle reference variables

```
rectangleReferenceVariable.rectangleMethod(…);
```

```
Rectangle fatRectangle;
Rectangle skinnyRectangle;
Rectangle squareRectangle;
```

Where does this code appear in the program?
NOT in the Rectangle class!
It belongs in the code of some other class that needs to create and use Rectangle objects.

Instantiating Objects

- Defining a class does NOT make an object
- Declaring a reference variable of the class also does NOT make an object
- To make an object it must be instantiated

```
//declare Rectangle reference variable
Rectangle aRectangle;
//now instantiate Rectangle object
//and point Rectangle reference variable at it
aRectangle = new Rectangle(5, 10);


//or do it all in one statement
Rectangle bRectangle = new Rectangle(2, 3);
```

Familiar?
Scanner class and objects!

Constructor method.
Same name as class name.
Parameter values (input data) used to initialise instance variable.

The instantiation operator

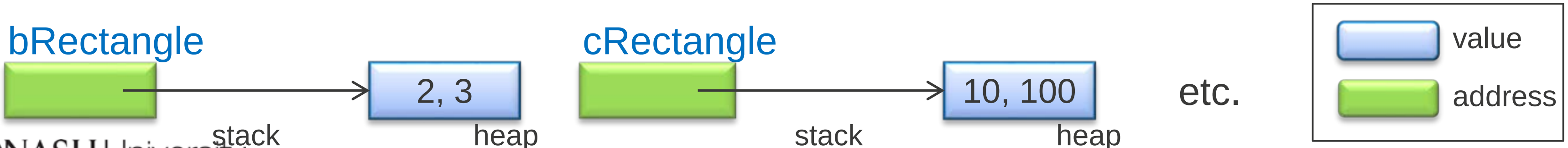
Reference type.
Same name as
class name.

Reference variable

Instantiation - Semantics

```
Rectangle bRectangle = new Rectangle(2, 3);
```

- The instantiation operator (new)
  - Returns an address in memory big enough to hold all of a Rectangle object's attribute values (instance variables)
- The Rectangle Constructor method (Rectangle (2, 3))
  - Initialises the instance variable values
  - In this case with its input data
- The address returned by the new operator is assigned to the Rectangle reference variable on the LHS of the assignment operator (bRectangle)
- The entire process repeats each time a Rectangle object is instantiated

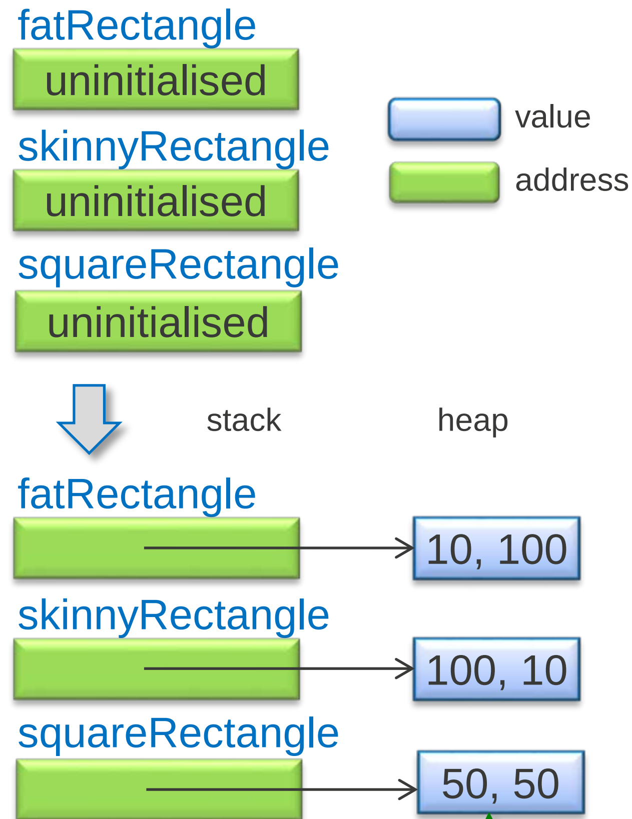


Instantiating Objects

```
//declare local Rectangle reference variables
Rectangle fatRectangle;
Rectangle skinnyRectangle;
Rectangle squareRectangle;

//instantiate Rectangle objects
//and point previously declared
//Rectangle reference variables at them
fatRectangle = new Rectangle(10, 100);
skinnyRectangle = new Rectangle(100, 10);
squareRectangle = new Rectangle(50, 50);
```

Constructor method initialises values of instance variables created by new operator



Data of 3 Rectangle instances in memory

- After the Rectangle variable declarations
  - Each variable is capable of pointing at any existing Rectangle object
  - They are all uninitialised and will cause a compile error if used
- After the Rectangle object instantiations
  - 3 Rectangle objects exist
  - Each is referenced by one and only one Rectangle object reference

i.e. there are no aliases

Constructor

- A Constructor is a special method
  - It has the same name as its class
  - It has no return type NOT EVEN void
- Its code executes immediately after the instantiation operator (new) has allocated memory space for an object's data and before any other code executes
  - Usually a Constructor's code should just initialise the new object's instance variables

```
Person aPerson = new Person();
```

Constructor

Default Constructor

- If you don’t code a Constructor (user-defined Constructor) for a class
  - Java provides a Default Constructor
    - You can’t see its code, it’s hidden
  - This Default Constructor initialises all instance variables to their default values
    - This is the value they are initialised to in their declaration OR
    - If not initialised in their declaration their Java default values
      - int: 0
      - float/double: 0.0
      - boolean: false
      - Object references (including Strings): null

This Default Constructor has no parameters (input data)

Java keyword, means referencing no object (see later)

Default Constructor - Example

```
public class Person {

   private int age = 99;
   private String name = "nobody";

}
```

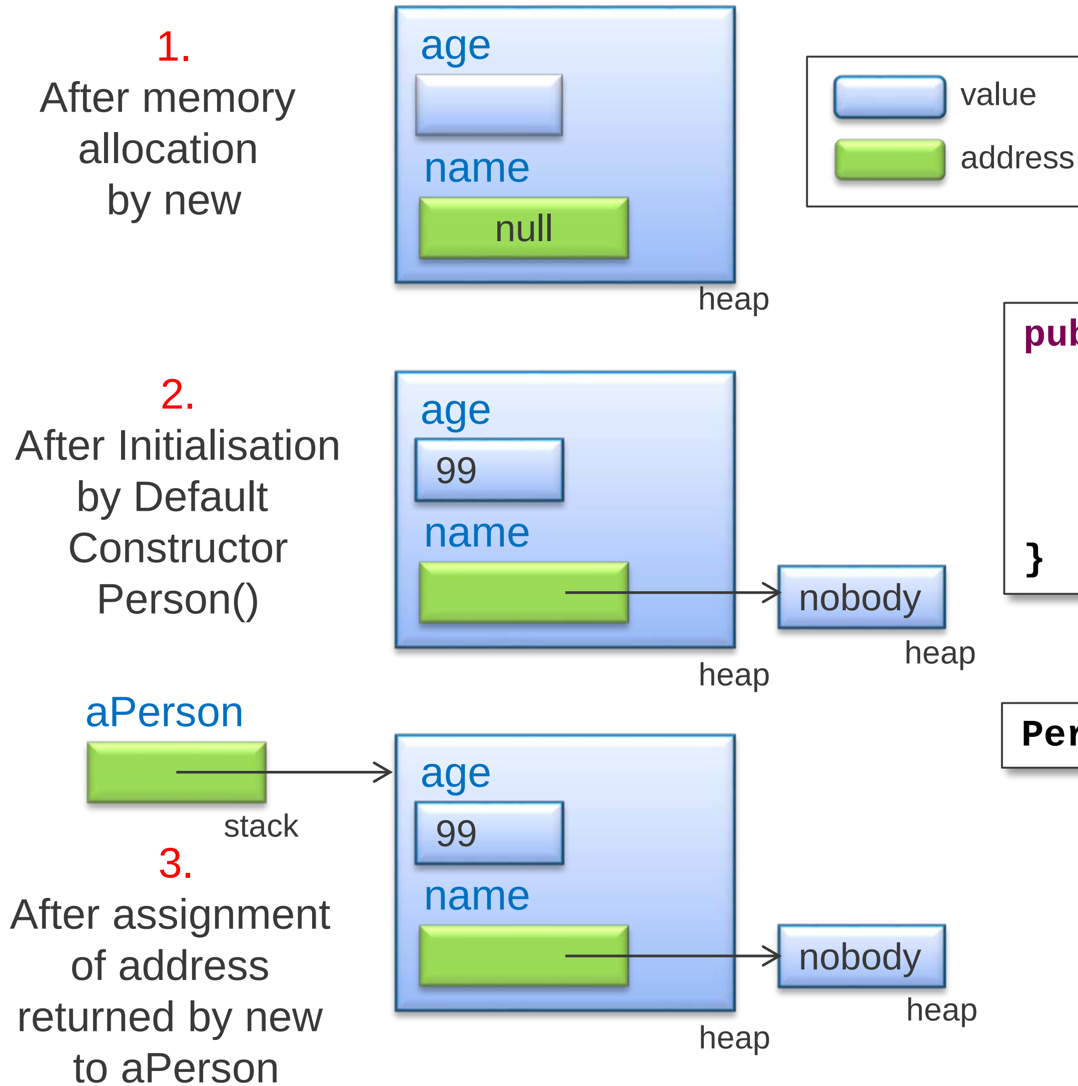
A very simple class with no Constructor coded.
Java provides a hidden Default Constructor

This code appears in another class which need to create and use a Person object

```
Person aPerson = new Person();
```

Because the Person class does not contain a coded Constructor this refers to the Default Constructor

Default Constructor - Example



A very simple class with no Constructor coded.
Java provides a hidden Default Constructor

```java
public class Person {

    private int age = 99;
    private String name = "nobody";

}
```

```java
Person aPerson = new Person();
```

This code appears in another class which needs to create and use a Person object

Programmer-Defined Constructor

With or without parameters (input data)

- If any Programmer-Defined Constructor is coded for a Class, Java does NOT provide a Default Constructor

```
public class Person {

   private int age;
   private String name;

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

}
```

No return type (not even void)

Same name as class

A Programmer-Defined Constructor So no Default Constructor provided for the Person class

To keep things simple this constructor has no parameters (input data), but constructors may be declared with parameters

Let Constructors Initialise

- It's poor style to initialise instance variables in their declaration
  - See a) on previous slide
  - It's too inflexible
    - e.g. should every person object be initialised with the same age and name
- Let Constructors do initialisation of instance variables
  - See b) on previous slide
  - When Constructors have parameters, initial values for instance variables can be specified, at time of instantiation, through these parameters and assigned to instance variables in the Constructor's code
  - e.g.

```
Person aPerson = new Person(21, "Chris");
```

Invoking Methods on Objects

- Nothing happens to an object instance after its instantiation until one of its class's methods is invoked on it
- Typically
  - Code outside of a class
    - Instantiates an object of the class
      - You know how to do this
    - Points a reference variable at it
      - Know how to do this
    - Invokes one of the class's methods on the reference variable to get some work done (possibly including getting a return value)
      - Did this with Scanner objects

Familiar?
Scanner class and objects!

This is sometimes called sending a message to the object

Invoking Methods on Objects

Like any method call this may or may not return a value depending on the method

- Syntax

```
objectReferenceVariable.methodName(parameters)
```

- objectReferenceVariable
  - An object reference variable
- methodName
  - Must be a public method of the class of the object referenced by objectReferenceVariable
- Parameters
  - Additional data required by the method to perform its task (not all methods require such data)
- Examples
  - Coming very soon

Method Overloading

A method's return type is not part of its signature and so a method name cannot be overloaded based only on different return types

- Methods each have a signature
  - It is the method name and the formal parameter list
    - Specifically, for the formal parameter list: the number of formal parameters, their type and the order of these types

- Method Overloading allows methods with the same name but unique signatures to coexist in a class
  - When one of these methods is called the Compiler knows which one to execute, despite the name ambiguity, by matching the actual parameter list types in the call with one of the various unique formal parameter lists in the overloaded method definitions

Method Overloading

- When an overloaded method is called the compiler selects which overload is to be called
  - By matching the actual parameter list types in the call with one of the various unique formal parameter lists in the overloaded method definitions

```
public static void main(String[] args){

    :
    a = 10 + tryMe(5, 3);
    :
}
```

```
private static float tryMe(int x){
    return x * 2;
}

private static float tryMe(int x, int y){
    return x * y;
}
```

- We have actually already seen overloading in action many times when using System.out.println(…)
  - e.g.

```
int myInt = 5;

System.out.println("Hello");
System.out.println(myInt);
```

These two calls invoke two different methods. One displays a String the other displays integers.

Constructor Overloading

- Since constructors are special methods they too can be overloaded
  - This allows a great range of initialisation options during instantiation

Part of person class

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

   public Person(int initAge, String initName){
      age = initAge;
      name = initName;
   }

   public Person(int initAge){
      age = initAge;
      name = "noName";
   }
   :
   :
```

Two constructors:
Same name (same as their class)
Different signatures

In another class that needs to create and use people objects

```
int hireAge = 20;

Person headProgrammer = new Person(34, "Mary");
Person anonymousTester = new Person(23);
```

It's clear to the compiler which constructor is being called

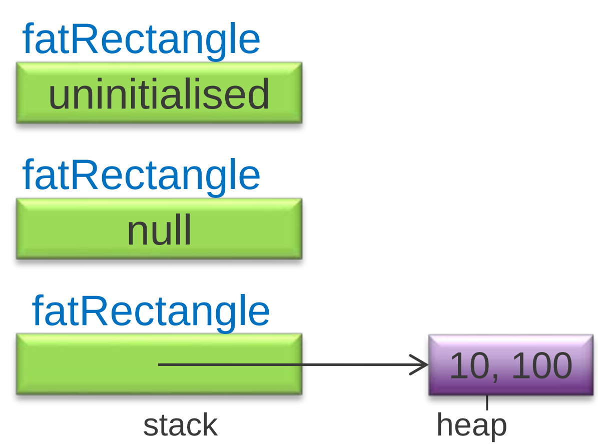
null

- It's a Java keyword
- It's a memory address
  - Actually the address composed of all 0's
  - This is not a real address
- If you set a reference variable to <u>null</u> Java interprets this to mean this reference variable is currently pointing at no object of its reference type
  - The reference variable still keeps it class type
- In Java you can use null explicitly
  - e.g. `Rectangle rectangle1 = null;`
  - e.g. `if (rectangle1 == null)…`
- e.g.

```
Rectangle fatRectangle;


fatRectangle = null;


fatRectangle = new Rectangle(10, 100);
```



null

- Instance variables that are reference variables are automatically initialised to null
  - local variables that are reference variables are NOT
- NullPointerException
  - If a reference variable has the value null and an attempt is made to invoke one of its methods
    - A run-time NullPointerException error will occur
  - This is a very common error
    - It often crops up in beginner's code because declaring a reference variable is incorrectly thought to instantiate an object of the variable's type
  - If a reference variable is uninitialised (this is different from having the value null) and an attempt is made to invoke one of its methods a compile time error will occur (i.e. before execution)

Class Constants

- We know that a constant is a named value which cannot change
  - E.g. `final int DAYS_IN_WEEK = 7;`
- Instead of being limited to placing constants inside a method as a local constant, we can also declare class-level constants
  - Like class-level variables, all methods will share the value of the constant
  - Like normal constants, the value cannot be changed
  - Declared at start of class, as:

```
static final data-type NAME = value;
```

Class Constants – Example

```
public class Rectangle {
   private static final MIN_LENGTH = 1;
   private int height;
   private int width;

   public Rectangle(int H, int W) {
      if (H < MIN_LENGTH)
        height = MIN_LENGTH;
      else
        height = H;
      if (W < MIN_LENGTH)
        width = MIN_LENGTH;
      else
        width = W;
   }
   public int computeArea() {
      return height * width ;
   }
   public int computePerimeter() {
      return  2 * (height + width);
   }
}
```

We can easily change the minimum ONCE

We don’t need to alter this code if we change the minimum side lengths.