Pilot Study – AUT University

Introduction

Information from AUT University was provided by Dr James Skene. This study includes:

- An analysis of the learning outcomes from the Course Descriptor. How does the current language, Java, meet these outcomes?
- Discussion with Dr Skene about the thinking of AUT University concerning a possible move to C.

Analysis of the Course Content and learning outcomes

The AUT University learning outcomes include the following:

LEARNING OUTCOMES:

On successful completion of this paper students will be able to:

- Write syntactically correct program statements.
- Assemble a program from statements that control the order in which tasks are performed.
- Assemble a program from statements that control the representation and processing of data.
- Read programs and predict what they do.
- Design and write programs to solve simple problems.
- Find and fix errors in programs.
- Write programs that interact with the user and the execution environment.
- Use tests to control program quality.
- Apply programming and documentation standards.

CONTENT

- Structured programming.
- Procedural programming.
- Types.
- Expressions.

- Program correctness.
- Testing.
- Debugging.
- Modularity.

Course material

Programming 1 at AUT University is taught using a combination of lectures, laboratory tutorials and assignments. The titles for the sections below are from the lecture material and are introduced in the following order.

Lecture 1 - What is programming?

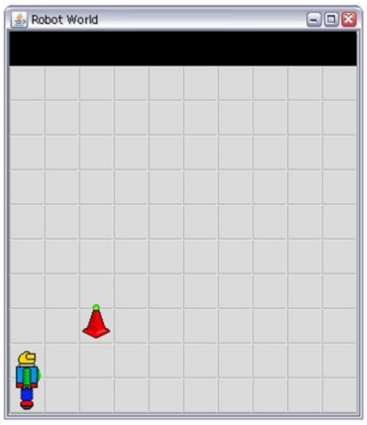
Introduces the different areas of computer science as different parts of a zoo; Computer Science is like the Natural History section while Software Development is like the breeding program. IT Service Science is analogous to selling tickets while Networks and Security are about designing appropriate cages.

The assumption behind these introductory sessions is that the students have had little or no exposure to software development. The first lecture provides some recommendations for tackling new or difficult concepts, the role of the Teaching Assistants and the value of working it out yourself. The concept of a *computer* is introduced, what parts they contain and some historical background.

Procedural concepts, stored programs and instructions are introduced by comparing the program instructions to making a cup of tea. *Processes* are presented as a way of understanding ways of solving problems using procedural steps. Java is briefly introduced as the language for the course.

Lecture 2 - How to write any computer program

Karel the Robot is introduced as a way of illustrating what procedural statements are used for. The BlueJ Java environment is discussed as a development environment.



Procedural steps are discussed to instruct the robot to move in different directions and pick up items. Example procedural *methods* include:

moveRobotForwards(); turnRobotLeft(); pickUpItemWithRobot();

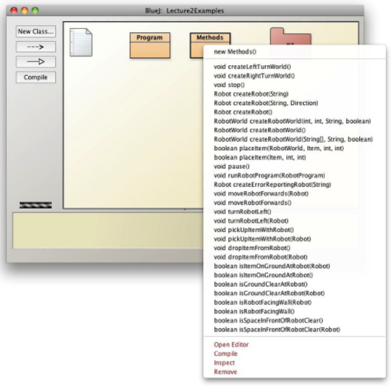
Methods are introduced as being *instructions* and the *call* concept is presented as a way of invoking an instruction at the appropriate step in the process.

The BlueJ environment and the program source contains structural elements which are explained as being necessary but which are not explored in detail at this time.

During this stage, students issue individual method calls by right-clicking on the list of available methods rather than building the program immediately line-by-line. They are then shown how to link method calls in a chain to create a set of procedural steps.

Issuing instructions directly

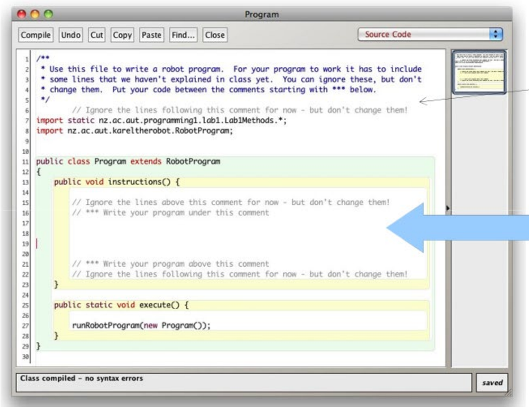
- Right-click on 'Methods'



```
new Methods()
void createLeftTurnWorld()
void createRightTurnWorld()
void stop()
Robot createRobot(String)
Robot createRobot(String, Direction)
Robot createRobot()
RobotWorld createRobotWorld(int, int, String, boolean)
RobotWorld createRobotWorld()
RobotWorld createRobotWorld(String[], String, boolean)
boolean placeItem(RobotWorld, Item, int, int)
boolean placeItem(Item, int, int)
void pause()
void runRobotProgram(RobotProgram)
Robot createErrorReportingRobot(String)
void moveRobotForwards(Robot)
void moveRobotForwards()
void turnRobotLeft()
void turnRobotLeft(Robot)
void pickUpItemWithRobot()
void pickUpItemWithRobot(Robot)
void dropItemFromRobot()
void dropItemFromRobot(Robot)
boolean isItemOnGroundAtRobot(Robot)
boolean isItemOnGroundAtRobot()
boolean isGroundClearAtRobot()
boolean isGroundClearAtRobot(Robot)
boolean isRobotFacingWall(Robot)
boolean isRobotFacingWall()
boolean isSpaceInFrontOfRobotClear()
boolean isSpaceInFrontOfRobotClear(Robot)
Open Editor
Compile
Inspect
Remove
```

This concept of accepting elements which are important but cannot be explored at this early stage is necessary to focus the students on the foundational concepts that are important at this time. While the commands such as *moveRobotForwards()* are actually invoking methods in objects, they are not presented as such at this stage.

Writing a program



```
/**
 * Use this file to write a robot program.  For your program to work it has to include
 * some lines that we haven't explained in class yet.  You can ignore these, but don't
 * change them.  Put your code between the comments starting with *** below.
 */
        // Ignore the lines following this comment for now - but don't change them!
import static nz.ac.aut.programming1.lab1.Lab1Methods.*;
import nz.ac.aut.kareltherobot.RobotProgram;


public class Program extends RobotProgram
{
    public void instructions() {

        // Ignore the lines above this comment for now - but don't change them!
        // *** Write your program under this comment




        // *** Write your program above this comment
        // Ignore the lines following this comment for now - but don't change them!
    }

    public static void execute() {

        runRobotProgram(new Program());
    }
}

Class compiled - no syntax errors                                          saved
```

The need for accurate syntax is explained. While *moveRobotForwards();* works, *move Robot Forwards()*; will not. A complete program is shown to accomplish the task of sending the robot to a place, retrieving an object and returning. The program is shown to work to specification since the steps specified and then executed, in the correct order, achieve the stated aims of the program.

Conditionals are introduced and the concept of boolean true/false. This is done without variables, using new methods that return boolean values.

```
isItemOnGroundAtRobot()
```

E.g.

```
if(isItemOnGroundAtRobot()) {
    pickUpItemWithRobot();
}
```

Also: `isGroundClearAtRobot()`

Loops are introduced as ways of extending the capabilities of the robot to solve more complex problems.

Structure theory is presented as a way of ordering and arranging instructions to solve a particular problem.

Lecture 3 - How computer programming works

Introduces switches, binary numbers, processors, memory and hard drives as elements that are needed to support programs. Decimal and binary numbering systems are introduced.

Machine code is discussed as a way of enabling a processor to manipulate bits and perform tasks. Arguments and jumps are explored. This leads to the concept of a *high-level language* and its advantages. Java is described as a high-level language and contrasted with assembler/machine code. The difference between natural language and programming languages is discussed; ambiguity is shown to be a problem that computers do not handle well.

Lecture 4 – Text

Numbers and *instructions* were introduced previously. *Text, characters and strings* are now introduced as well as ASCII encoding. This reinforces the concepts of representation of information and the ability of procedural instructions to manipulate data. Files are introduced.

The flow of concepts at this stage includes:

Character strings to represent instructions
→ storing instructions in text files
→ compilation and interpretation.
→ detecting programming instruction *errors*
→ Java as a *semi-interpreted* language.

Lecture 5 - Syntax

Language S*yntax* is introduced as part of the set of concepts *Syntax* + *Semantics* + *Construction*. How program instructions are assembled using correct punctuation and statement terminators is explained. Syntax trees are used to explain the purpose of individual program statements, including conditionals and loops.

Lecture 6 - Nesting program statements

Introduces formatting as a way of making program structures clearer. Syntax diagrams and comments are used to explain techniques. Recursion, block statements and balanced brackets to demonstrate regions of code that are either entered into or skipped due to conditionals. *Nesting* of code blocks. Indenting as a way of making nested regions easier to see.

Understanding the rules of syntax diagrams helps students to understand the function of individual Java methods and constructs from the documentation.

Lecture 7 – State and semantics

A program statement can cause the current *state* of the program to change. Concept of *semantics* (what a statement does). Understanding the state of the program at a particular time can be used to help determine if it is operating correctly. Concept of *requirements*. Not all states are important. The concept of running to the correct final state is introduced.

Lecture 8 – Tracing code

Techniques for understanding what a program will do without running it. Understanding the current state of the program (i.e. where it is up to) and what has changed as a result of executing a statement.

Lecture 9 – Variables and Data Types

Purpose of *variables* and their use in remembering values that are useful. Variables have *types* and are used for different purposes. Concept that variables can be used in conditional statements to determine appropriate actions. Java *type declarations* are introduced and *initialisation.* Program *literals* and *constants.* Testing values of variables and comparison.

Lecture 10 – Operators

Operators as a way of calculating a new value from a current value. *Assignment* and the concept that a variable can be modified as a result of applying an operator. *Arithmetic operators*, *Unary* operators. Operator *precedence.*

Expressions are not mathematics. In the real world, there is no value that is simultaneously three more than itself. However, in programming, *value* = *value* + *3* is useful since it is an *instruction*. *Prefix* and *Postfix* operators

Lecture 11 – Output and debugging

Definition of the String data type and Escape codes (\n, \r). How to print to the terminal using strings. Introducing *print()* and *prinln()* methods and string literals. Printing the values of variables.

Program diagnostics and debugging. Program traces after code has run (e.g. "*the loop did not run*")

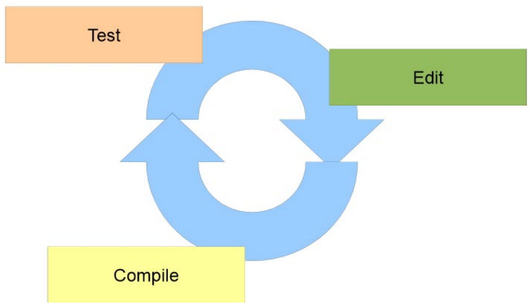
Lecture 12 – How to program

Program *construction* = "How to behave when programming". Understanding the problem as a first step. Requirements and the importance of reading the original question. Introduction to Junit tests. Concept of re-using existing code vs writing from scratch ("toolkits").

Program *plans* in relation to telling the robot the technique to solve the problem. There is a difference between *knowing what needs to be built* and *knowing how to build it*.

Top-down versus Bottom-up approaches. Attacking the biggest task first and breaking it down into smaller steps. Invalid approaches such as "Big-Bang all-at-once" and Multi-tasking (working on or trying to concentrate on more than one step at a time).

Programming style/methodology.



- Continually asking "what does this bit do?"
- Carefully changing things in a controlled manner versus random hacking. "*When in doubt, comment out…*"
- Comments in code and "barrier arms" to debug incrementally down through the code.
- Compiling frequently – testing as the program gets incrementally bigger rather than just trying to debug the whole mess at the end. When it fails, ask "*what changed ?*"
- Test with a purpose rather than just random running.
- Allow enough time. We always underestimate. The need to be focussed.
- Guess once then look it up – major time-saving technique.
- The program has no feelings, is not vindictive and does not like or dislike you. It is a machine. It will only work correctly when it is right.
- Celebrate your success.

Lecture 13 – Booleans and Return values

Boolean variables and relational operators (=, ==, < >). True and False concepts. Logical complements. Truth tables. Boolean operators in control structures (*if, while*).

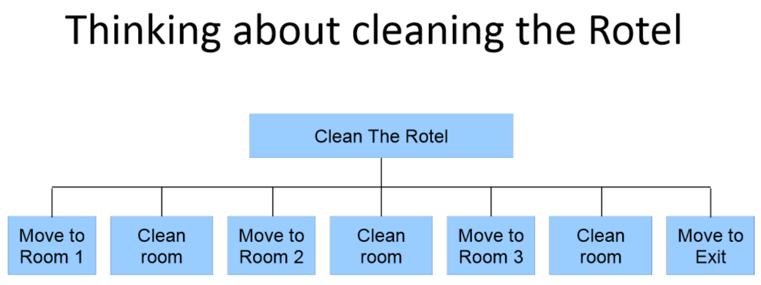
Lecture 14 – Procedures and Procedural Code.

Example of navigating to and cleaning all the rooms in a motel. Examination of completed code samples rather than just concepts. *Procedures* are also called *methods* or *subroutines*

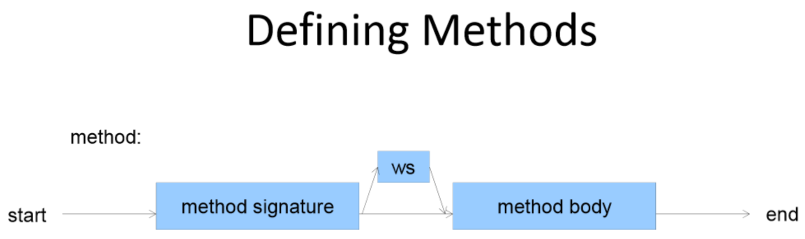
Purpose of procedures and benefits:

1. We have eliminated 'code duplication'
2. The code is easier to read, understand and check for correctness

Techniques for tracing method calls. Diagrams to understand program structure.



Syntax diagram of methods to understand the structure. *Method signatures* and *return types* including *void*.



Similarity of method body to a block statement.

Procedural abstraction = breaking larger block of code down into smaller procedures. Definition*: "Abstraction is whenever we simplify something complex by putting it in a box*"

Lecture 15 – References, Arguments and Return values

Concept of creating a robot. All subsequent methods work on only the last-created robot. Controlling multiple robots requires a *reference* to each robot. Reference is a variable. createRobot() method returns a value (which is a handle to the object). Concept of *complex objects* such as the robot

Question: *First mention of objects?*

Concept of *arguments*. e.g. *moveRobotForwards(robbie); moveRobotForwards(karel);*

Use in control structures and multiple arguments. Argument types.

Syntax of method calls with expressions and arguments. Contrast with methods as *statements*. Method signatures specify what arguments are required/can be required/ for a particular method.

Getting input from the user *String input = readLine();*

Appending strings (concatenation) and the append operator + as used in string concatenation versus mathematical operations.

String comparison versus stringEquals() method. Why if(firstInput == secondInput) fails (comparing addresses not values).

Lecture 16 – Planning an algorithm

Definition of an algorithm and approaches to planning. Algorithm as a series of steps. Programs are not algorithms; programs implement algorithms. Can express as a set of numbered steps and can be *hierarchical* or structure diagrams. Planning includes analysis and design. Consideration of previous implementations (has it been done before?). Steps, top-steps and sub-steps.

Top-down = *plan step-by-step*
versus
Bottom-up = *brainstorm the bits and then try to assemble them.*

Lecture 17 – Developing and algorithm

Backward reasoning, generalisation and exploratory programming. Incremental improvements and iterative planning (refactoring?). Setup of steps and completion of tasks. Alternative approach of planning backwards (i.e. what are we wanting to achieve at the end? Work backwards through the steps). Assumptions.

General versus particular algorithms. Sweet-spot between too general and to abstract, too complex and too simple.

Lecture 18 – Scope of variables.

Local variables and their lifetime (i.e. *scope*). Re-use of identifiers.

Lecture 19 – Return values.

Difficulties of managing variables that go out-of-scope. Evils of global variables (implied). Contrast with returning variables from methods. Using method calls in expressions. Using return values in control structures.

Lecture 20 – Functions

Pass by value. *Pure functions* and *side effects* (extra actions apart from calculating the return value required i.e. non-pure or *impure*). String functions. Scope and lifetime of parameters. What is a method really for? Calculating values or performing actions? Or both?

Lecture 21 – Good programs

What is a good program? Maintenance. Readability versus efficiency. Improving readability. Comments on comments.

Role of stakeholders and requirements. Requirements say what should happen when we interact with the software. Validation. Minimising program costs. What makes a good comment and documentation.

Lecture 22 – References and nulls

Introduction to *objects*. Proposed as being *complex values. E.g. Karel the Robot* has many properties including its name, position, if it is alive. Object is presented as a *package*.

Objects can be *instanced* . Related to *references*. Need to declare references in the same way variables need to be declared. Handling references safely (what NOT to do with them in code). Concept of what a reference names (i.e. not the object – they name the *instance*). Null objects

Like variables, objects have *types* and an object state.

Lecture 23 – Arrays

Example of controlling groups of objects as a unit rather than by repeating lots of code to process each object in turn. Concept of an array storing a list of values and having a data type. All elements are of the same data type. Assignment and initialisation. Index operators.

Lecture 24 – Arrays and FOR loops.

Introduction to FOR loops as control structures. Bounds and exceptions. Scope of variables within loops. Array limitations.

Lecture 24 – Exceptions

How programs cope with bad data/arguments. Implications for run-time checking. Throwing exceptions. Testing for exceptions.

Lecture 26 – Method contracts

Interfaces versus *implementations*.

We only need to know what is written on the box, not what is in the box

The method's *Contract* is explained as being the extra information that is not carried in the method signature (e.g. side-effects, possible errors…). References to Java documents to understand how to use a method without understanding *how* it does its job. Javadocs and how to use them.

Lecture 27 – More about Values

Character expressions, type mismatches, type conversions, *Final* variables as constants

Lecture 28 – Objects and Classes

(week 10)

Right at the end – concept of "lies I told you to make things simpler"

Link to Stanford http://see.stanford.edu/see/materials/icspmcs106a/handouts.aspx

http://see.stanford.edu/see/lecturelist.aspx?coll=824a47e1-135f-4508-a5aa-866adcae1111

28 lectures