# TABLE OF CONTENTS

**INTRODUCTION** ............................................................................................................... 1
  Module aim ......................................................................................................................... 1
  Module abstract ................................................................................................................ 1

**Unit 2: System Development** .......................................................................................... 1
  Learning outcomes and assessment criteria .................................................................... 2
    Summary of learning outcomes and assessment criteria ............................................. 3
  Module content ............................................................................................................... 4
  Lectures .......................................................................................................................... 5
  Class exercises and activities ......................................................................................... 6
  Information resources ..................................................................................................... 6

**Prescribed textbook** ....................................................................................................... 6
  Prescribed textbook for Object Oriented Programming .................................................. 6

**Recommended information sources** ............................................................................. 6
  Recommended books ......................................................................................................... 6
  Recommended websites .................................................................................................... 7

**Using this Study Guide** .................................................................................................. 7

**Concluding remarks** .................................................................................................... 11

**Unit 1: Object Oriented Programming** ......................................................................... 13
  Learning objectives ......................................................................................................... 13
  Prescribed reading ........................................................................................................... 14

**Introduction** ................................................................................................................... 14

1.1 **Object oriented system characteristics** ..................................................................... 14
  1.1.1 Delivery environments ......................................................................................... 14

1.2 **Object oriented programming vs procedural programming** .................................. 14

1.3 **Programming languages** .......................................................................................... 15
  1.3.1 Control structures .................................................................................................. 15
  1.3.2 Libraries ............................................................................................................... 18
  1.3.3 IDEs .................................................................................................................... 18

1.4 **Software engineering concepts** .................................................................................. 18
  1.4.1 Modularity ............................................................................................................ 19
  1.4.2 Objects and classes ............................................................................................... 19
  1.4.3 Inheritance ........................................................................................................... 49
  1.4.4 Polymorphism ....................................................................................................... 58
  1.4.5 Encapsulation ....................................................................................................... 61
  1.4.6 Relationships ....................................................................................................... 61

**Concluding remarks** ..................................................................................................... 63

**Self-assessment** ............................................................................................................. 63

**Unit 2: System Development** ....................................................................................... 65
  Learning objectives ......................................................................................................... 65
  Prescribed reading ......................................................................................................... 66

**Introduction** ................................................................................................................... 66

2.1 **CRC** .......................................................................................................................... 66
  2.1.1 Identifying classes ............................................................................................... 67
  2.1.2 Identifying class responsibilities ......................................................................... 67

2.2 **UML class diagrams** ................................................................................................ 68
  2.2.1 Class diagrams ..................................................................................................... 68
  2.2.2 Attributes ............................................................................................................. 68
  2.2.3 Methods ................................................................................................................ 69
Introduction

Welcome to Object Oriented Programming! This introductory unit will provide you with an overview of the module.

The main sources of information for Object Oriented Programming are a prescribed textbook and Study Guide. The Study Guide should not be seen as a replacement for the prescribed textbook – you have to use it in conjunction with the textbook, which contains the actual learning content of the module. You are expected to work through the textbook independently. The Study Guide facilitates this process by means of references to relevant chapters/sections in the prescribed textbook.

In this introductory unit, we provide you with the following information on Object Oriented Programming:

- A brief description of the aim of the module
- An abstract of the module
- The learning outcomes and assessment criteria pertaining to the module
- An outline of the module content
- An outline of the module structure
- An explanation of the purpose, design and proper use of the Study Guide and prescribed textbook

Module aim

This module will provide you with an understanding of the principles of object oriented programming as an underpinning technological concept in the fields of programming, data management and system development.

Module abstract

Object oriented programming is an industry-proven method used to develop reliable modular programs; it is particularly popular in the fields of software engineering and system development.

The consistent use of object-oriented techniques can lead to shorter development life cycles, increased productivity, adaptable code, the re-use of different technologies as well as the interaction between different systems via common platforms that lower the cost of producing and maintaining systems.

System development with objects simplifies creating and maintaining complex applications. Object oriented programming is a way of modelling software that maps programming code to the real world.
In terms of its influence, object-oriented technologies can be found in many systems, from commercial operating systems to mobile telephones as well as in various multimedia applications. The majority of programming languages are object oriented in focus, with exceptions offering specialised programming resources: such is dominant in Visual Studio, C++, Java, the Microsoft .Net environment, ActionScript, etc.

Students undertaking this module will develop their understanding of the object-oriented paradigm; they will develop code suited to a range of platforms using object-oriented methodologies.

**Learning outcomes and assessment criteria**

Upon successful completion of this module, you will:

1. Explain the principles of object oriented programming
2. Design object oriented programming solutions
3. Implement object oriented programming solutions
4. Test and document object oriented programming solutions

The following table outlines the assessment criteria that are aligned to the learning outcomes.
## Summary of learning outcomes and assessment criteria

<table>
<thead>
<tr>
<th>Learning outcomes</th>
<th>Assessment criteria to pass</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Upon successful completion of this module, you will:</strong></td>
<td><strong>You can:</strong></td>
</tr>
<tr>
<td>1. Explain the principles of object oriented programming</td>
<td>1.1 Discuss the principles, characteristics and features of object oriented programming</td>
</tr>
<tr>
<td>2. Design object oriented programming solutions</td>
<td>2.1 Identify the objects, data and file structures required to implement a given design  &lt;br&gt; 2.2 Design an object oriented programming solution for a given problem</td>
</tr>
<tr>
<td>3. Implement object oriented programming solutions</td>
<td>3.1 Implement an object oriented programming solution based on a prepared design  &lt;br&gt; 3.2 Define the relationships between objects in order to implement design requirements  &lt;br&gt; 3.3 Implement object behaviours using control structures in order to meet design algorithms  &lt;br&gt; 3.4 Make effective use of integrated development environments (IDEs), including code and screen templates</td>
</tr>
<tr>
<td>4. Test and document object oriented programming solutions</td>
<td>4.1 Critically review and test object oriented programming solutions  &lt;br&gt; 4.2 Analyse actual test results against expected results to identify discrepancies  &lt;br&gt; 4.3 Evaluate independent feedback on a developed object oriented programming solution and make recommendations for improvement  &lt;br&gt; 4.4 Create onscreen help to assist computer program users  &lt;br&gt; 4.5 Create documentation for the support and maintenance of computer programs</td>
</tr>
</tbody>
</table>

These outcomes are covered in the module content and they are assessed in the form of written assignments and semester tests. If you comply with and achieve all the pass criteria related to the outcomes, you will pass the module.

Learning and assessment may be performed across modules, at module level or at outcome level. Evidence may be required at outcome level, although opportunities exist for covering more than one outcome in an assignment.
Module content

1. **Explain the principles of object oriented programming**

   Object characteristics: this entails the types of object, e.g. constructors, destructors, etc.; object classification; object features, e.g. inheritance, polymorphism, encapsulation, public/private classes, public/private methods and message passing; and libraries, e.g. interpreted, open source and common.

   Variables: this entails public/private instance variables and static references.

   Software engineering: this entails features, e.g. modularity, encapsulation, re-use, method overload, instance variables, classes, abstract classes and interfaces.

   Classes: this entails characteristics, e.g. identifying attributes, controlling attribute and method scope, inheritance, aggregation, association and polymorphism.

2. **Design object oriented programming solutions**

   Development: this entails selection, e.g. identifying programming languages and libraries; and selecting development environments.

   Design methodologies: this entails design options, e.g. re-using existing systems, adapting code and using open sources.

   Design methods: this entails tools, e.g. class-responsibility-collaboration (CRC) cards; class diagrams; identifying dependencies and inheritance; and data and file structures.

   System delivery: this entails style, e.g. scripted, interpreted and compiled.

   Programming platforms: this entails types, e.g. graphics user interfaces (GUIs) and script and command lines.

   Delivery environments: this entails the types of device, e.g. mobile, hand-held, web-based, desktop and dedicated.

   Interactions: this entails exchanging data with other systems, compliance, compatibility and standards recognition.

   Design refinements: this entails clarifying designs via the principles of maximum coherence and minimum coupling between classes.
3. **Implement object oriented programming solutions**

Coding: this entails conventional and pre-defined language commands, e.g. class libraries and downloaded, imported and reversion code.

Control structures: this entails types, e.g. sub-routines, branching, iteration, interrupts and signals.

Complexity: this entails including multiple classes; applying inheritance when created code and re-using objects.

IDEs: this entails typical elements, e.g. source code editors, compilers, interpreters, build automation tools and debuggers.

4. **Test and document object oriented programming solutions**

Testing: this entails mechanisms, e.g. valid declarations, debugging and comment code, naming conventions and checking functionality against requirements and documentation.

Errors: this entails handling, e.g. managing extremes and using system-imposed statements.

Impact testing: this entails types, e.g. range, input and load testing, and system compatibility.

Onscreen help: this entails methods, e.g. pop-ups, help menus and hotspots.

Documentation: this entails technical documentation, e.g. designs, delivery systems, platforms, environments, file structures, coding, constraints and maintenance requirements.

**Lectures**

Each week has four compulsory lecture hours for all students. It is recommended that the lecture hours be divided into two sessions of two hours each, but this may vary depending on the campus.

Each week has a lecture schedule, which indicates the approximate time that should be allocated to each activity. The week’s work schedule has also been divided into two lessons.
**Class exercises and activities**

You will be required to complete a number of exercises and activities in class. These activities and exercises may also contribute to obtaining a pass, therefore, it is important that you are present in class so that you do **not** forfeit the opportunity to be exposed to such exercises and activities.

Activity sheets that are submitted should be kept by the lecturer so that they can be used as proof of criteria that were met, if necessary.

**Information resources**

You should have access to a resource centre or library with a wide range of relevant resources. Resources can include textbooks, e-books, newspaper articles, journal articles, organisational publications, databases, etc. You can access a range of academic journals in electronic format via EBSCOhost. You may have to ask a campus librarian to assist you with accessing EBSCOhost.

**Prescribed textbook**

**Prescribed textbook for Object Oriented Programming**

ISBN: 9780321861276

**Recommended information sources**

**Recommended books**


**Recommended websites**

http://java.sun.com/docs/books/tutorial/


www.dmoz.org/Computers/Computer_Science/Publications/

www.github.com

www.junit.org

www.tutorialsonpoint.com

**NOTE**

- Web pages provide access to a further range of Internet information sources.
- You must use these resources with care, justifying the use of information gathered.

**Using this Study Guide**

As indicated earlier, the prescribed textbook is your main source of information for this module and the Study Guide serves as a guide to such.

The purpose of the Study Guide is to facilitate your learning and to help you to master the content of the prescribed textbook and other material. It, further, helps you to structure your learning and manage your time; provides outcomes and activities to help you master said outcomes; and directs you to the appropriate chapters/sections in the prescribed textbook. It is, therefore, important that you start with the Study Guide.

The Study Guide has been carefully designed to optimise your study time and maximise your learning so that your learning experience is as meaningful and successful as possible. To deepen your learning and enhance your chances of success, it is important that you read the Study Guide attentively and follow all instructions carefully. Pay special attention to the module outcomes at the beginning of the Study Guide as well as at the beginning of each unit.

It is essential that you complete the exercises and other learning activities in the Study Guide as your module assessments (examinations, tests and assignments) will be based on the assumption that you have completed such.
The Study Guide accompanies the prescribed textbook, therefore, it should be read in conjunction with such: it should **not** be deemed as a replacement for the prescribed textbook.

**Purpose**

The purpose of the Study Guide is to facilitate your learning process and to help you to structure your learning and to master the content of the module. The prescribed textbook covers certain themes in detail; where applicable, more simplified explanations are provided in the Study Guide.

It is important for you to attentively work through the prescribed textbook and Study Guide and to follow all the instructions as set out in the latter. In this way, you should be able to deepen your learning and enhance your chances of success.

**Structure**

The Study Guide is structured as follows:

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Unit 1</th>
<th>Object oriented programming</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unit 2</td>
<td>System development</td>
</tr>
<tr>
<td></td>
<td>Unit 3</td>
<td>System testing and documentation</td>
</tr>
<tr>
<td></td>
<td><strong>Glossary</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Individual units**

The individual units in the Study Guide are structured in the same way. Each unit contains the following features, which should enhance your learning process:

<table>
<thead>
<tr>
<th>Unit title</th>
<th>Each unit title is based on the title and content of a specific learning outcome or assessment criterion (criteria) as discussed in the unit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning outcomes and assessment criteria</td>
<td>The unit title is followed by an outline of the learning outcomes and assessment criteria, which will guide your learning process. It is important for you to become familiar with the learning outcomes and assessment criteria as they represent the overall purpose of the module as well as the end product of what you should have learnt in the unit.</td>
</tr>
<tr>
<td>Learning objectives</td>
<td>Learning objectives, which follow the learning outcomes and assessment criteria, are statements that define the expected goal(s) of the unit in terms of the specific knowledge and skills that you should acquire as a result of mastering the unit content. Learning objectives clarify, organise and prioritise learning and they help you to evaluate your own progress, thereby taking responsibility for your learning.</td>
</tr>
</tbody>
</table>
Introduction

The prescribed reading section is followed by an introduction that identifies the key concepts of the unit.

Content

The content of each unit contains the theoretical foundation of the module and is based on the work of experts in the field of the module. Theory is illustrated by means of relevant examples.

Concluding remarks

The concluding remarks at the end of each unit provide a brief summary of the unit as well as an indication of what you can expect in the following unit.

Self-assessment

The unit ends off with a number of theoretical self-assessment questions that test your knowledge of the content of the unit.

Glossary

As you can see, we include a glossary at the end of the Study Guide. Please refer to such as often as necessary to familiarise yourself with the exact meaning of terms and concepts involved in object oriented programming.

The use of icons

Icons are used to highlight (emphasise) particular sections/points in the Study Guide, to draw your attention to important aspects of the work or to highlight activities. The following icons are used in the Study Guide:

**Activity**

This icon indicates a learning activity/exercise that has to be completed, whether individually or in a group, in order to assess (evaluate) your understanding of the content of a particular section.

**Learning outcome alignment**

This icon indicates how the individual units in the Study Guide are aligned to a specific learning outcome and its assessment criteria.

**Prescribed reading**

This icon indicates the relevant chapters/sections in the textbook that you are expected to study.

**Test your knowledge**

This icon appears at the end of each unit; it indicates that you are required to answer self-assessment questions to test your knowledge of the content of the foregoing unit.
## Alignment to prescribed textbook

The following table reflects the alignment between the learning outcomes, assessment criteria, units in the Study Guide and chapters in the prescribed textbook.

### Study Guide/prescribed textbook alignment

<table>
<thead>
<tr>
<th>Learning outcomes</th>
<th>Assessment criteria</th>
<th>Study Guide units</th>
<th>Textbook chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Explain the principles of object oriented programming</td>
<td>1.1 Discuss the principles, characteristics and features of object oriented programming</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td>2. Design object oriented programming solutions</td>
<td>2.1 Identify the objects, data and file structures required to implement a given design</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td>3. Implement object oriented programming solutions</td>
<td>2.2 Design an object oriented programming solution for a given problem</td>
<td>2</td>
<td>2, 5, 6, 10</td>
</tr>
<tr>
<td></td>
<td>3.1 Implement an object oriented programming solution based on a prepared design</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>3.2 Define the relationships between objects in order to implement design requirements</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>3.3 Implement object behaviours using control structures in order to meet design algorithms</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>3.4 Make effective use of IDEs, including code and screen templates</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td>4. Test and document object oriented programming solutions</td>
<td>4.1 Critically review and test object oriented programming solutions</td>
<td>3</td>
<td>4, 5</td>
</tr>
<tr>
<td></td>
<td>4.2 Analyse actual test results against expected results to identify discrepancies</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>4.3 Evaluate independent feedback on a developed object oriented programming solution and make recommendations for improvement</td>
<td>1</td>
<td>1, 3, 4, 7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>4.4 Create onscreen help to assist computer program users</td>
<td>3</td>
<td>4, 5</td>
</tr>
<tr>
<td></td>
<td>4.5 Create documentation for the support and maintenance of computer programs</td>
<td>2</td>
<td>2, 5, 6, 10</td>
</tr>
</tbody>
</table>
Concluding remarks

At this point, you should be familiar with the module’s design and structure as well as with the use of the prescribed textbook in conjunction with the Study Guide.

In Unit 1, we start with the actual module content by looking at an overview of object oriented programming.
Unit 1: Object Oriented Programming

Unit 1 is aligned with the following learning outcomes and assessment criteria:

**Learning outcomes**

- LO1 Explain the principles of object oriented programming
- LO2 Design object oriented programming solutions
- LO3 Implement object oriented programming solutions
- LO4 Test and document object oriented programming solutions

**Assessment criteria**

- AC1.1 Discuss the principles, characteristics and features of object oriented programming
- AC2.1 Identify the objects, data and file structures required to implement a given design
- AC3.1 Implement an object oriented programming solution based on a prepared design
- AC3.2 Define the relationships between objects in order to implement design requirements
- AC3.3 Implement object behaviours using control structures in order to meet design algorithms
- AC3.4 Make effective use of integrated development environments (IDEs), including code and screen templates
- AC4.2 Analyse actual test results against expected results to identify discrepancies
- AC4.3 Evaluate independent feedback on a developed object oriented programming solution and make recommendations for improvement

**Learning objectives**

After studying this unit, you should be able to:

- Understand object oriented programming
- Understand software engineering concepts
- Understand the importance of software engineering concepts
- Understand why object oriented programming is better than procedural programming
- Understand inheritance, polymorphism and encapsulation
- Understand aggregation and association
- Understand programming languages
- Identify different programming languages and libraries
- Understand packages
- Understand IDEs
- Understand class structure attributes (data) and methods (behaviour)
- Understand and implement unified modelling language (UML) class diagrams

**Prescribed reading**


**Introduction**

Classes, which are adopted in many programming language structures, combine data and methods. Objects are instances of classes that are created. Practices that develop around ideas in system development are called ‘object oriented programming’.

1.1 **Object oriented system characteristics**

Software systems are becoming central to almost all business operations. Object-oriented systems are preferred owing to the ever-changing business environment, which requires robust systems. Object-oriented systems are based on the behaviour and structure of information systems in the form of little modules that contain data and processes.

1.1.1 **Delivery environments**

Object oriented programming is used to create different types of application. These range from basic systems to large-scale enterprise applications. Applications for consumer devices, such as cellphones and personal digital assistants (PDAs), are also developed via objects (Deitel & Deitel, 2010:9).

1.2 **Object oriented programming vs procedural programming**

There are different programming methodologies apart from object oriented programming, one of which is procedural programming. Procedural programming is based on the development of procedures, which are separate from data. As a result, access to data is unpredictable and uncontrolled.
In object oriented programming, data and the operations that use such are both contained in objects; this is particularly important when hiding data. Data hiding plays a big role in object oriented system security.

# 1.3 Programming languages

The following describes a programming language:

- How the language describes the data
- The statements that work on said data
- The grammar that describes how expressions are formed

Most programming languages, including Java, have a specification document, which identifies expressions and statements as well as what various groupings mean: this is known as the ‘semantics’ of a programming language. It is crucial for developers to familiarise themselves with the semantics of programming languages in order to speed up system development.

## 1.3.1 Control structures

A control structure is a syntactic structure that controls other statements. Such can be viewed in terms of sequence, selection and repetition. Indentation is used for controlled statements.

### 1.3.1.1 Sequence

This is when program statements are executed one after the other, in the order in which they were written.

```java
num1 = x + b;
num2 = num1 + v;
```

**Figure 1 – Sequence**  
Source: Deitel and Deitel (2010:78)

### 1.3.1.2 Selection

This is code that you can execute some of the time, **not** all of the time. Test conditions are in parenthesis; these determine whether the statements in a selection block will be executed.

Selection can be implemented using the `if` statement and `switch`. 
Conditions in `if` statements are expressions that can either be true or false. A program will make a decision based on the value of a condition. Conditions can be formed using equality operators (== and !=) and relational operators (>, <, >= and <=).

Figures 2 and 3 illustrate `if` statements:

```java
if (<test>) {
    <statement>
    <statement>
    <statement>
}
```

**Figure 2 – If statement**
Source: Reges and Stepp (2014:267)

```java
if (number1 == 12) {
    sum = 1 + 4;
    total = total + sum;
}
```

**Figure 3 – If statement**
Source: Ndai Mapaso (2015)

Switch performs different actions based on the possible values of a constant integral expression or type byte, short, int or char (Reges & Stepp, 2014:1144).

Figure 4 illustrates a `switch`:

```java
switch (grade) {
    case 9:
        count = count * 9;
        break;
    case 8:
        count = count * 8;
        break;
    default:
        count = count * 7
        break;
}
```

**Figure 4 – Switch statement**
Source: Ndai Mapaso (2015)
1.3.1.3 Repetition

Repetition is also known as ‘looping statements’.

A for loop avoids redundancy by repeatedly executing a sequence of statements over a given range of values.

Figure 5 illustrates the structure of a for loop:

```java
for (<initialisation>;<continuation test>;<update>) {
    statement;
    statement;
    statement;
}
```

**Figure 5 – For loop**  
Source: Reges and Stepp (2014:119)

If you were to develop a program to calculate the square root of every number from one to five, you could use the following (Figure 6):

```java
for (j = 1, j <= 5; j++) {
    System.out.println (j + “squared = ” + (j * j));
}
```

**Figure 6 – For loop**  
Source: Reges and Stepp (2014:119)

The above program initialises variable j to 1. The statement with the println method is executed whilst the value of j is less than or equal to 5. After each println has been executed, the value of j is increased by 1 from the j++ statement.

A while statement is a repetition statement that allows programmers to repeat an action while some condition remains true.

Figure 7 illustrates a while statement:

```java
int product = 1;
while (product <= 100) {
    product = 4 * product;
}
```

**Figure 7 – While statement**  
Source: Reges and Stepp (2014:340)

The value of the product above will be 4, 16, 64, 256.
1.3.2 Libraries

Programming language developers have come up with a way to avoid re-inventing the wheel. They have achieved this goal by moving frequently re-used code into separate files, known as ‘libraries’. A program stored in such can only be executed when developed code makes a call to the right library.

Java has a huge amount of pre-written software that developers can use to enhance their programs. For example, if a programmer wants to develop a program that connects to a specific Internet site, Java has a library that can simplify such a connection. Java has different libraries to perform different functions (Table 1):

<table>
<thead>
<tr>
<th>Library</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java.io</td>
<td>Retrieves data from a database</td>
</tr>
<tr>
<td>Java.awt, javax.swing</td>
<td>Draws a graphics user interface (GUI)</td>
</tr>
<tr>
<td>Java.math</td>
<td>Performs mathematical computations</td>
</tr>
</tbody>
</table>

Table 1 – Java libraries

All libraries are collectively known as ‘Java class libraries’. Developers can refer to the Java class library called ‘Application Programming Interface’ (API).

1.3.3 IDEs

An IDE is a software application that provides full facilities to programmers when developing computer systems. An IDE, normally, consists of built-in functionality to help programmers edit and debug software programs and, as a result, maximise the productivity of software developers. An IDE can be viewed as a single program in which all development is done; such provides authoring, compiling, modifying, deploying and debugging software features.

1.4 Software engineering concepts

Software engineering comprises all associated documentation and configuration data that are required to produce a finished and functional program. Software engineering is used in the development of different kinds of software that are in need of maintenance, new development and that run different operations. It can be thought of as a group of objects that consists of activities that are enclosed in one program.

Software engineering is important in contemporary business as it helps organisations to overcome problems with technological solutions.
Table 2 summarises some frequently asked questions (FAQs) along with their answers; such will help to explain software engineering concepts:

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is software engineering?</td>
<td>Software engineering is an engineering discipline that encompasses all aspects of software development</td>
</tr>
<tr>
<td>What is software?</td>
<td>Software is computer programs and their associated documentation</td>
</tr>
<tr>
<td>What is the difference between system engineering and software engineering?</td>
<td>System engineering deals with all aspects of computer-based system development, including software, hardware and process engineering. Software engineering is part of this general process</td>
</tr>
<tr>
<td>What are the essential software engineering activities?</td>
<td>The essential software engineering activities are software specification, development, validation and evolution</td>
</tr>
<tr>
<td>What are the key challenges faced by software engineering?</td>
<td>Software engineering needs to cope with the demands of reduced delivery time, increased diversity and developing secure and trustworthy software</td>
</tr>
</tbody>
</table>

Source: Sommerville (2011:6)

1.4.1 Modularity

Modular design involves programs that consist of modules, which are processing components. Modules are easy to maintain, understand and implement; they can also be re-used. Each module can be developed, tested and corrected separately, by different programmers, and, thereafter, integrated into the final system.

1.4.2 Objects and classes

Objects are the building blocks of an object oriented programming program. Programs that use object oriented programming technology are, basically, collections of objects. A program consists of objects that have properties and operations that said objects can perform.

In object oriented programming, key characteristics of objects include:

- The object’s behaviour: the methods that can be applied to the object.
- The object’s state: how the object reacts when methods are applied to it.
- The object’s identity: how the object can be distinguished from other objects that may have the same behaviour and/or state.
A simple rule for identifying classes is to look for nouns in a given problem. Methods correspond to verbs. When taking an order processing system as an example, some nouns may be:

- Item
- Order
- Payment
- Account

The above nouns may lead to classes.

The next step is to look for verbs. Item is added to Order. Order can be shipped or cancelled. Payment is applied to Order. Consider each of the verbs and identify the one object that has the major responsibility of performing it.

The ‘noun and verb rule’ is only a guideline and only with experience can you decide which nouns and verbs are important when building classes.

As stated earlier, each object consists of data and behaviour:

- Object data: the data stored within an object represent the state of such. In object oriented programming terminology, this data is called an ‘attribute’. The object Employee can have the following attributes: name, surname, date of birth, gender and telephone number.
- Object behaviour: this is what an object can do. In object oriented programming, behaviour is contained in methods and you invoke such by sending messages.
- Class: this can be viewed as a template or blueprint from which objects are actually made.

### 1.4.2.1 Controlling access to members of a class

There are different access level modifiers that can be used to define a class. Access level modifiers determine whether other classes can use a particular field or invoke a particular method. Packages are named groups of classes. There are two levels of access control, namely:

1. At the top level: public or package private (there is no explicit modifier)
2. At the member level: public, private, protected or package private (there is no explicit modifier)
Table 3 summarises the different access level modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>The class is visible to all classes everywhere</td>
</tr>
<tr>
<td>Protected</td>
<td>The member can only be accessed with its own package and, in addition, by a subclass of its class in another package</td>
</tr>
<tr>
<td>No modifier/package private</td>
<td>The class is visible only within its package</td>
</tr>
<tr>
<td>Private</td>
<td>The member can only be accessed within its own class</td>
</tr>
</tbody>
</table>

Source: [http://docs.oracle.com/javase/tutorial](http://docs.oracle.com/javase/tutorial)

Access levels to members as allowed by each modifier are shown in Table 4:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Protected</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>No modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Source: [http://docs.oracle.com/javase/tutorial](http://docs.oracle.com/javase/tutorial)

The Class column indicates whether the class itself has access to the members as defined by that access level. A class has access to its members.

The Package column indicates whether classes in the same package as the class (despite their parentage) have access to the member.

The Subclass column indicates whether subclasses of the class declared outside of this package have access to the member.

The World column indicates whether all classes have access to the member.

Access levels affect programs in **two** ways, namely:

1. When you use classes from other sources, e.g. classes in Java, access levels will determine which members of said classes your own classes can use.
2. When you create a class you need to decide which access level every member variable and method in your class should have.
Figure 8 illustrates two collections of classes: package one and package two:

![Figure 8 - Collections of classes]

Table 5 illustrates where the members of the Alpha class are visible to each of the access modifiers as applied to them:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Alpha</th>
<th>Beta</th>
<th>AlphaSub</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>No modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Source: [http://docs.oracle.com/javase/tutorial](http://docs.oracle.com/javase/tutorial)

### 1.4.2.2 Points to note when choosing access levels

In project teams, programmers share programs. To ensure that errors do not occur, access levels can be used:

- Use the most restrictive access levels, such as private
- Avoid public fields except for constants
- A class may be declared with the public modifier
1.4.2.3 Java classes

A class consists of a class header and body. The class header line is always the first line of a class. A basic class has the following:

- The word `public`
- The word `class`
- The name of the class

Important basic elements to identify are:

- The class header line
- The beginning of the class’s body, which is represented by a `{`
- The end of the class’s body, which is represented by a `}`

The body of a class comes after the header line and is enclosed in `{}`. This set of braces determines the scope of the class, i.e. the beginning and end of the class. There are **two** sections of code in the body of a class, namely:

1. The first section declares the type of data to be used in the program.
2. The second section defines the methods/procedures that will be applied to the data.

A Java class will include both or even just **one** of the above sections.

Always remember the basic concept that a class can be viewed as a plan where items required are listed and procedures defined (Figure 9):

```
Class header line → public class MyFirstClass
                  → {
Class’s body → class data and methods go here
                  → }
```

**Figure 9 – Structure of a basic Java class (MyFirstClass)**

Source: Bronson (2004:19)
• Naming classes

Class names start with a capital letter. Consecutive words in a class name also begin with a capital letter. In Figure 9, the class name was written as MyFirstClass. Also take a look at Figure 10:

```
public class MyFirstClass {
    private String name;

    public void displayName() {
        System.out.println("Your name is");
    }
}
```

Figure 10 – Sample Java class
Source: Bronson (2004:19)

1.4.2.4 Java programs

The Java language provides a set of classes and programs that may be constructed using pre-defined classes. Every Java program must consist of a class and a method called ‘main()’. The execution of a program begins in the main() method.

• main() method

Figure 11 illustrates the structure of the main() method:

```
public static void main (String[] args) {
    //place your method statements here;
}
```

Figure 11 – main() method
Source: Bronson (2004:25)
A description of the `main()` method, as shown in Figure 11, would be:

1. The **visibility modifier** is `public`.
2. The **scope designation** refers to `static`, which specifies how the method is created and stored in memory. General specific objects that must be used independently of any specific object must be declared as `static`. The `main()` method can be described as a general-purpose method.
3. The **return type** is the type of data that is returned from the method. The reserved word `void` shows that the method will **not** return any value when it has completed executing.
4. The **method name** is `main()`.
5. The **argument list** is the type of data, if there is any, that is sent to the method when it is run. The data is referred to as the ‘arguments’ of a method. These are provided as command line arguments.
6. The braces, `{}`, signify the beginning and end of the method. The statements within these determine what the method does.
7. The method name, together with the type of data sent to the method (which is defined within parenthesis), is referred to as a method’s ‘signature’.

### 1.4.2.5 Java methods

Java has **three** types of permitted word, namely:

1. Reserved words
2. Pre-defined words (referred to as ‘standard identifiers’)
3. Programmer supplied words (referred to as ‘identifiers’)

Identifiers **cannot** be reserved words. Java methods require parenthesis following its name. Such indicates that the identifier refers to a method name. Therefore, the main method is referred to as `main()`.

### 1.4.2.6 Building your own classes

When constructing an object from a class, you will have to create an **instance** of the class. Each object stores information about what it currently looks like (its state). The state of an object may change over time. The change in the state of an object must be as a result of method calls. If an object’s state changes and this is **not** as a result of a method call, encapsulation is broken.
In Figure 12 we create a class to represent an employee. The **Employee** class has **name**, **surname** and **gross** attributes:

```java
class Employee {
    // data declaration section
    private String surname;
    private double gross;

    // method declaration section
    // constructor method for the Employee()
    public Employee (String s, String n, double grossPay) {
        surname = s;
        name = n;
        gross = grossPay;
    }

    private String name;
    public String getSurname() {
        return surname;
    }

    public double calculateNet (double grossPay) {
        double netSalary = 0;
        netSalary = grossPay - (grossPay * 0.20);
        return netSalary;
    }
}
```

![Figure 12 – Employee class](source: Ndai Mapaso (2015))

There are **three** instance fields that will hold the data to be manipulated inside an instance of the **Employee** class:

1. `private String surname;`
2. `private String name;`
3. `private double gross;`

The keyword **private** ensures that the only methods that can access these instance fields are the methods of the **Employee** class only. No outside method can read or write to these fields.

Please note that **name** and **surname** are references to **String** objects. Classes can contain instance fields of any defined class type, e.g. if you have defined an **Employee** class then any class may use **Employee** for an instance field.
1.4.2.7 Constructing data declaration sections

There are three classifications of variables, namely:

1. Instance
2. Class
3. Local

There is an additional type that is treated like a variable but formally referred to as a ‘parameter’. Classifications of variables depend on placement within a class and the inclusion or exclusion of the word `static`.

Declaration statements can be placed either outside or within a method. Any variable declared outside a method, such as `private String name` in the `Employee` class, automatically becomes part of the class’s declaration section even if the declaration is made between methods.

Instance variables are those variables whose declaration:

1. Is made within a class’s body and outside of any method
2. Does **not** contain the `static` reserved word

When constructing the data declaration section, stick to the convention of placing all instance declaration statements at the top of a class’s body. This is mainly done to create an easily identifiable data declaration section for anyone who is going to read the code.

The reserved word (keyword) `private` at the beginning of each declaration statement restricts access to the declared variable so that only methods defined in the method definition section can retrieve or modify such.

If we were to decide to add a calculation for tax to the `Employee` class, we may need to include an appropriate variable in the declaration section. It is acceptable to include a variable as follows:

```java
private double taxAmount;
```

This really is the individual programmer’s choice as the net amount earned is readily calculated without providing the storage area for the tax amount.

From our `Employee` class declaration, the variables `name`, `surname` and `gross` are instance variables because they were placed outside of the methods and do **not** include the reserved word `static`; `static` creates a variable that cannot be included within an object.
1.4.2.8 Creating objects

Objects are created from the instance variables declared in data declaration sections or from other object types. The methods that are defined in method declaration sections are used to either:

- Provide operations that can be applied to the created objects
- Create general-purpose functions that are independent of any one object

The mechanics of creating objects are the same for all classes. Let us track how an object is created by using the instance variables in our Employee class. Figure 13 illustrates how to create a single Employee object called ‘worker’:

```java
public class EmployeeTest
{
    public static void main (String[] args)
    {
        Employee worker = new Employee (name, surname, grossPay);
    }
}
```

**Figure 13 – Worker object**

The statement `Employee worker` is a declaration statement that declares a variable called ‘worker’ to be of type Employee. The variable `worker` is user defined. The data type for this variable is Employee, which is the name of the class from which we want to create the object.

In the `main()` declaration statement, declaring an object to be of a class type is the same as declaring a variable to be of a primitive data type, such as `double` or `int`. Note that there is no reserved word `private` in the declaration: this is not because we are declaring an object but rather because of the placement of the declaration within a method. In a method we do not necessarily have to use the reserved word `private`.

Declaration statements within a method are referred to as ‘local variables’. For example, `netSalary` is a local variable in the `calculateNet()` method. Local variables can only be used within the methods that declared them (Figure 14):

```java
public double calculateNet (double grossPay)
{
    double netSalary = 0;
    netSalary = grossPay - (grossPay * 0.20);
    return netSalary;
}
```

**Figure 14 – Local variable**

Source: Ndai Mapaso (2015)
Local variables do **not** include access specifications as these variables are private by definition and thus do **not** require an explicit *private* designation.

The statement `Employee worker` can be explained as follows:

Working with the assumption that `EmployeeTest` has been compiled, the `main()` declaration memory storage area is reserved for a variable called `worker`; the value placed in this variable is an address designated as `null`. This means that the variable does **not** currently refer to a valid object.

The next step would be to create an actual object through the use of the `new` operator in Java:

```java
Employee worker = new Employee (name, surname, grossPay);
```

The `new` operator creates an object of the specified type by getting enough memory to store the required values. In our example, sufficient memory to store **two** references to `String` and a `double` is obtained. Memory is dynamically allocated while the program is executing. The `new` operator is referred to as the ‘dynamic memory allocation operator’. The process of creating a `new` object, using a dynamic memory allocator, is referred to as ‘creating an instance’ and ‘instantiating an object’. This should explain why the variables `name`, `surname` and `gross` are called ‘instance variables’. An instance of these variables exists when an object is created; each object will have **one** instance of each instance variable.

Each object created from a class gets its own set of instance variables. If another object is created use:

```java
Employee emp = new Employee (name, surname, grossPay);
```
Each of the objects, `worker` and `emp`, will have its own set of the variables `name`, `surname` and `gross` as shown in Figure 15:

![Diagram showing object variables](Image)

**Figure 15 – Each object gets its own set of instance variables**

Source: Bronson (2004:86)

You can also first declare the reference variable and then instantiate the object in **two** separate statements:

```java
Employee worker;
```

and

```java
Worker = new Employee (name, surname, grossPay);
```

The `Employee` class has **three** methods as mentioned:

- `public Employee`
- `public String getSurname`
- `public double calculateNet`

All of the methods are tagged as `public`. This means that any method in any class can call the method.

The format of a method header is as follows:

```
public returnType methodName (parameter list) \(\leftarrow\) required parenthesis
```
Table 6 summarises the different class method types:

<table>
<thead>
<tr>
<th>Type of method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Initialises instance variables when an object is created</td>
</tr>
<tr>
<td>Accessor</td>
<td>Gives the value stored in an object’s variable</td>
</tr>
<tr>
<td>Mutator</td>
<td>Modifies the value stored in an object’s variable</td>
</tr>
</tbody>
</table>

Source: Bronson (2004:91)

1.4.2.9 Constructors

A constructor is any method that has the same name as its class. A class can have no user-defined constructors or one or more user-defined constructors, as long as each constructor has a unique signature. Compile errors, usually, occur when the unique identification of a constructor is not possible.

In the Employee class, we will now introduce a new type of method called a ‘constructor’. In object oriented programming, the name of a constructor is the same as the name of a class. The constructor runs when you construct the objects of the Employee class - giving the instance fields the initial values that you want them to have. The compiler will recognise that the method name is identical to the class name and will thus consider the method to be a constructor.

When you create an instance of the Employee class as in the code:

```java
new Employee ("Makhura", 4000);
```

you have to set the instance fields as:

```java
surname = "Makhura"; gross = 4000
```

There is a difference between a constructor and other methods. A constructor can only be called in conjunction with the new operator.

A few things to note about constructors are:

- A constructor has the same name as its class.
- A class can have more than one constructor.
- A constructor may take zero or more parameters.
- A constructor has no return values, not even void.
- A constructor is always called with the new operator.
Avoid the following with constructors:

- A constructor **cannot** be applied to an existing object to re-set an instance of a field, e.g. `John. Employee ("Olivia Pope", 150);`.
- If there is a return value, the compiler will **not** treat the method as a constructor.
- Keep constructors simple and use such for initialisation purposes only.
- Introducing local variables with the same name as the instance fields (e.g. the following constructor) (Figure 16) will **not** set the `gross`.

```java
public Employee (String name, double grossPay)
{
    String surname = name;
    double gross = grossPay;
}
```

*Figure 16 – Local variable* Source: Ndai Mapaso (2015)

The above variables are only accessible inside the constructor.

- **When is a constructor called?**

A constructor is called when a new object is created. The `new` keyword creates an instance of the `Employee` class, which allocates the required memory:

```
Employee emp = new Employee ("Makh", 5000);
```

The above code will instantiate an `Employee` object and call the `Employee` method, which is the constructor.

- **What is inside a constructor?**

The function of a constructor is to initialise the memory allocated when the `new` keyword is encountered. The code inside a constructor is expected to set the object to its initial state. Attributes are initialised in constructors.

- **Default constructors**

If a constructor is **not** included in a class, the class will still compile. The compiler or system provides a constructor if the class has no explicit constructor. Such a default constructor creates the object and calls the constructor of its superclass. In most cases, the superclass will be part of the language framework, such as the `Object` class in Java.
If a constructor was **not** created for the `Employee` class, the following default constructor would have been inserted by the compiler (Figure 17):

```java
public Employee()
{
    super();
}
```

*Figure 17 – Default constructor*

Source: Ndai Mapaso (2015)

If you have added a constructor, the compiler will **not** add the default constructor to a class.

### Using multiple constructors

An object can be created in many different ways. In order for you to cater for such a scenario, you will need to provide more than **one** constructor (Figure 18):

```java
public class Employee
{
    int gross;
}
```

```java
public Employee()
{
    gross = 0;
}
```

*Figure 18 – Multiple constructors*

Source: Ndai Mapaso (2015)

You may want to initialise the attribute `gross` to 0, as was done by the constructor. Then, in another constructor, you may want to pass an initialisation parameter that allows `gross` to be set to a certain amount. The other constructor is illustrated in Figure 19:

```java
public Employee (int g)
{
    gross = g;
}
```

*Figure 19 – Multiple constructors*

Source: Ndai Mapaso (2015)

Both of the constructors will be in **one** class.
1.4.2.10 Destructors

A destructor is a method in object oriented programming that is called when an object’s life cycle is over; such includes proper clean-up functions. Said clean-up involves releasing system memory that the object would have acquired. Java and .NET reclaim their system memory via a Garbage collection mechanism. There are other languages where manual memory management is required. In languages such as C++, developers are expected to include code in destructors to free up memory. If memory is not freed, there will be a memory leak. As long as memory has not been released, such will not be accessible to the entire operating system. A memory leak could end up using available system memory.

Every class that is created in Java has the methods of the Object class. One of these methods is the finalise method. This method is rarely used in most system development projects as it can cause system performance problems.

The Java Virtual Machine (JVM) uses the automatic Garbage collection to reclaim memory taken by objects that are no longer in use. This is done when there is no longer any reference to an object. Although Java has a Garbage collection mechanism, other problems may still arise.

The Garbage collector calls the finalise method to perform termination housekeeping just before Garbage reclaims the object’s memory. The method is not a destructor but rather a method that provides additional safety to ensure that external resources, such as opening and closing files, will be done.

To invoke the method, you can use:

Object.finalize()

or

System.runFinalizersOnExit(true)

The finalise method is not guaranteed to perform at a specific time. The Garbage collector could execute before a program terminates, therefore, it is not clear when finalise will be called.

Developers can force the Garbage collector to release memory, however, Garbage will do so at its own time. Programmers can use:

System.gc()

or

Runtime.getRuntime().gc()
Figure 20 illustrates the structure of the `finalise` method:

```java
protected void finalise()
{
    //insert finalisation code here
}
```

**Figure 20 – finalise() method**

The `finalise` method is defined with the visibility modifier `protected` to prevent access to `finalise()` by code defined outside of its class (Figure 21):

```java
package testgc;
/**
 * @author makh
 */
public class TestGC
{
    /**Example shows Garbage collector in action
     * Note that the finalize() method of object GC1 runs without being
     * specifically called and that the ID’s of Garbage collected objects are not
     * always sequential
     */
    public static void main (String[] args)
    {
        Runtime rt = Runtime.getRuntime();
        System.out.println ("Available Free Memory: " + rt.freeMemory());
        for (int i = 0; i < 10000; i++)
        {
            GC1 x = new GC1(i);
        }
        System.out.println ("Free Memory before call to gc(): " + rt.freeMemory());
        System.runFinalization();
        System.gc();
        System.out.println ("Free Memory after call to gc(): " + rt.freeMemory());
    }
}

class GC1
{
    String str;
    int id;
    GC1 (int i)
    {
        this.str = new String ("abcdefghijklmnopqrstuvwxyz");
        this.id = i;
    }
}

protected void finalize()
{
    System.out.println ("GC1 object " + id + " has been finalized.");
}
```

**Figure 21 – finalise() method**
### 1.4.2.11 Accessors and mutators (getters and setters)

The concepts of ‘getters’ and ‘setters’ support data hiding. Objects should **not** directly manipulate data within other objects; getters and setters provide controlled access to object data. Getters and setters are sometimes called ‘accessor methods’ and ‘mutator methods’ respectively. The **set** and **add** methods modify the state of objects.

- **Accessor methods**

Accessor methods provide a means of reading the values as stored in object variables. They are member methods that can access a class’s private data members for the purpose of returning individual values. The **get()** method only looks up the state of a method and then reports back. An object with multiple instance variables will have multiple accessor methods, **one** for each variable.

A method that returns either a combined value or displays **one** or more values is also frequently referred to as either ‘accessor’ or ‘modified accessor’ methods.

For example, if we had a **Date** class that has **month**, **day** and **year** instance variables, a modified accessor might convert the **three** values into a string, such as 01/01/2015, and then return this string’s value. An additional modified accessor could simply display all **three** values. True accessor methods would return the individual instance variable values individually. Each accessor method will be given individual names, such as **getSurname()** in our **Employee** class example (Figure 22):

```java
public String getSurname()
{
    return surname;
}
```

*Figure 22 – Accessor method*

Source: Ndai Mapaso (2015)

When constructing classes, remember to include a complete set of accessor methods. Each accessor method does **not** necessarily have to return the data member’s exact value; it can also return a useful representation of said value. For example, a **Date** class could have a **get()** method that returns 01/01/2015.
• Mutator methods

Mutator methods provide a way of changing one or more of an object’s values after such has been created (Figure 23):

```java
public setGross()
{
    gross = 250;
}
```

**Figure 23 – Mutator method**  
Source: Ndai Mapaso (2015)

A class can have many mutator methods as long as each method has a unique name. In our `Employee` class, we could have a mutator that changes the object’s `surname`, a mutator that changes the object’s `name` and a mutator that changes the object’s `gross`.

In summary, the basic methods that each class should contain are constructors, accessors and mutators.

**Activity**

Complete the example used in this activity (Figure 24).

Your main task is to create constructor, accessor and mutator methods as well as to complete the `main()` method.

Tax is pegged at 20% of `grossPay`.

Create two programs: one with the given data and the other where you allow the user to capture values.
package employee;
import java.util.Scanner;
/**
 * @Author Makh
 * Date created: 26-05-2014
 * Name of program: Employee.java
 */

public class Employee {
    //enter the name of the employee
    private String surname;
    private double gross;

    //initialising the constructor for the Employee() using the default constructor
    public Employee() {
        surname = "Mapaso"
        gross = 4000;
        name = "Ndai"
    }

    //set the surname of the employee
    public void setSurname(String sname) {
        surname = name;
    }

    //get the surname of the employee
    public String getSurname() {
        return surname;
    }

    //set the gross of the employee
    public void setNewValues() {
        gross = gross + 0.55;
        name = "KB";
        surname = "Makh";
    }

    //get the gross of the employee

    //insert the missing methods
    public static void main(String[] args) {
        //instantiating Employee class
        Employee worker = new Employee();
        System.out.println ("Employee Surname" + worker.getSurname());

        //calculate net salary
        System.out.println ("Net Salary" + worker.calculateNet(grossPay));
        System.out.println ("Changed Gross" + worker.setSalary());
    }
}
1.4.2.12 Developing class methods

Previously, we created methods to initialise, modify (setters) and display (getters) an object’s state.

Below is a slightly different program with an **Employee** class constructor method that uses literals that are coded within the method to initialise each newly create **Employee** object to the same values. This restriction also applies to the mutator method `setNewValues()`, which re-sets gross to `gross + 0.55`, name to “KB” and surname to “Makh”.

In a program, the only values that we can give to an object are literal values, supplied as part of the method.

We now want to modify the program to make use of data that is given when the method is invoked. The method accepts external data, that is, data that is **not** hard coded into the method. To do this, define or create a method that is able to correctly receive, store and process said data. The passed data is called the ‘argument’ of the method. If no data is required by the method, no argument is allowed to be passed to the method when it is invoked.

In our previous example this is shown via `worker.setNewValues()`.

Assume that the mutator method has been created to accept a **double** value and that the object `worker` has been created. The statement would look like this:

```
worker.setNewValues(KB, Makh, 0.55)
```

or

```
worker.setNewValues(String s, String n, double amountAdded)
```
The afore-mentioned is represented in Figure 25:

Figure 25 – A method that accepts external data independent of an object
Source: Ndai Mapaso (2015)

Instead of using literal data, arguments can also constitute variable names. Therefore, values stored in variables will be passed on to the method. It is the parenthesis that allows you to pass on additional data, besides what is contained in the named object, when a method is invoked.

Irrespective of how a method is constructed, when it is invoked, it is referred to as a ‘called method’. The act of invoking a method is called ‘calling a method’ and the method doing the calling is called a ‘calling method’.

- **Writing methods**

To pass data on to a called method, said method should have been written to accept data. A called method will produce results after it has successfully received data.

Figure 26 illustrates the structure of a method:

```
Method’s header line: public methodName()
→ {
Method’s body: Java statements go in here
→ }
```

Figure 26 – Structure of a method
Source: Bronson (2004:130)
Remember that a method consists of a method header and body, as shown in Figure 26. Method headers are there to:

- Specify access privileges, i.e. where methods can be called
- Identify the data type of returned values
- Provide method names
- Specify the number, order and types of argument expected by methods

The structure of a method that can receive arguments is:

```java
public void setValues (String s, String n, string grossPay) { no semicolon
```

All values passed on to the method `setValues` are passed on to the variables `s`, `n` and `grossPay` (in that order). When the method is called, the number, sequence and data type of the arguments passed to the method must be the same in number, sequence and data type with the parameters declared in the method’s header line. Each parameter should have its data type declared individually. Method variables are placed in the body whereas method parameters are placed in parenthesis.

Figure 27 illustrates a mutator method using parameters:

```java
public void setValues (String s, String n, string grossPay) {
    surname = s;
    name = n;
    gross = grossPay;
}
```

**Figure 27 – Mutator method using parameters**

Source: Ndai Mapaso (2015)
Now include the method `setNewValues()` in a fully functional program. If you provide a constructor with no arguments it will be regarded as a default constructor (Figure 28):

```java
public class Employee {
    //data declaration section
    private surname;
    private name;
    private gross;

    //method declaration section
    public Employee() //constructor
    {
        surname = "Mapaso";
        name = "Ndai";
        gross = 3555.60;
    }

    public void showValues() //modified accessor method
    {
        System.out.println ("Surname, Name and Gross pay" + surname + " " + name + " " + gross);
    }

    public void setValues (String s, String n, string grossPay) //mutator
    {
        surname = s;
        name = n;
        gross = grossPay;
    }

    public double calculateNet()
    {
    }

    public static void main (Strings[] args)
    {
        Employee worker = new Employee(); //create and initialise an object of type Employee
        System.out.println ("The object’s initial values are");
        worker.showValues;
        worker.setValues ("KB", "Makh", 10000);
        System.out.println ("The values of this object have been changed to");
    }
}
```

**Figure 28 – setNewValues() method**

Source: Ndai Mapaso (2015)

### Method overloading

Method overloading is when the same method name is used for more than one method. A compiler must be able to determine which method to use, depending on the data type of the parameters, not the data type of the return values (if any). Overloaded methods are mostly useful in the development of constructor methods as they allow programmers to initialise objects in more than one way.
Figure 29 illustrates two constructors in Java:

```java
public Employee() //constructor
{
    surname = “Mapaso”;  
    name = “Ndai”;  
    gross = 3555.60;
}

public Employee(String s, String n, string grossPay) //another constructor
{
    surname = s;  
    name = n;  
    gross = grossPay;
}
```

**Figure 29 – Java constructors**

The constructor that is used depends on the supplied argument types when the method is called. For example:

```java
Employee worker = new Employee();
```

will call the default constructor, i.e. the constructor with no arguments.

```java
Employee worker = new Employee(“Mapaso”, “Ndai”, 4000)
```

will cause the compiler to use the constructor with two Strings and a double.

In method overloading, the same method name is used; each method that uses the same name must be written and exist as a separate entity. The code within a method does not have to be similar but should perform the same operations.

A compiler will determine which method to use, depending on the method’s name and its parameter list. In the header line, the part that has the method name and the parameter list is called the method’s ‘parameter signature’. Each method in a class should have a unique parameter signature.

- Returning a single value

A called method must provide the data type of the returned value as well as the actual value being returned. Consider the method in Figure 30:

```java
public void calculateNetSalary()
{
    System.out.println (”Net Pay is “ + gross - (gross * 0.20));
}
```

**Figure 30 – Single value**
The word `void` specifies that a method is **not** going to return a value. If a method is supposed to return a value, the word `void` must be replaced with the data type that represents the value that is going to be returned by said method.

If a method is to return an integer value, the method header line would be:

```java
public int calculateNetSalary()
```

If a method is to return a `double`, the method header line would be:

```java
public double calculateNetSalary()
```

In the method body, there should be a statement that will force the return of the required value. This is done through the use of the `return` keyword:

```java
returnExpression;
```

Upon encountering a `return` statement, the expression is evaluated and the value sent back to the calling method. Control is reverted back to the calling method (Figure 31):

```
public double calculateNetSalary()
{
    return gross - (gross * 0.20);
}
```

**Figure 31 – return statement**

Source: Ndai Mapaso (2015)

The data type in the `return` statement should match the data type declared in the method’s header line. In Figure 31, the data type in the header line is `double`, therefore, the `return` expression is expected to return a double value.
Figure 32 illustrates a complete program used to calculate the **Employee** net salary using a tax rate of 20%. The program is using external data:

```java
package employee;
import java.util.Scanner;
/**
 * @Author Makh
 * Date created: 26-05-2014
 * Name of program: Employee.java
 */

public class Employee {
    //enter the name of the employee
    private String surname;
    private double gross;
    private String name;

    //initialising the constructor for Employee() using the default constructor
    public Employee(String s, String n, double grossPay) {
        surname = s;
        gross = grossPay;
        name = n;
    }

    //set the surname of the employee
    public void setSurname(String s) {
        surname = s;
    }

    //get the surname of the employee
    public String getSurname() {
        return surname;
    }

    //set the gross of the employee
    public void setNewValues() {
        gross = gross + 0.55;
        name = "KB";
        surname = "Makh";
    }

    //get the gross of the employee
    //insert the missing methods

    //method to calculate net salary
    public double calculateNet() {
        double netSalary = 0;
        netSalary = gross - (gross * 0.20);
        return netSalary;
    }

    public static void main(String[] args) {
        double grossPay;
    }
}
```
String s, n;

//instantiating Employee class
Scanner input = new Scanner(System.in);
System.out.println(“Enter surname”);
s = input.nextLine();
System.out.println(“Enter name”);
n = input.nextLine();
System.out.println(“Gross Pay”);
grossPay = input.nextDouble();

Employee worker = new Employee(s, n, grossPay);
System.out.println(“Employee Surname” + worker.getSurname());
System.out.println(“Changed Gross” + worker.setSalary());
System.out.println(“Net Pay” + worker.calculateNet());
}
}

**Figure 32 – Employee class**
Source: Ndai Mapaso (2015)

### Returning multiple values

The procedure for returning multiple values would be to return said values directly via a named object. In such a case, you would expand the class’s instance variables to include both or all of the variables; these could automatically be calculated and stored within each constructor and mutator method.

For example, in a program that calculates the tax and net salary of an employee, there would be one method to calculate both the tax and net salary. Another way is to calculate both the tax and gross, then concatenate these values into a single string and then to return such (Figure 33):

```java
public String calcTaxAndNetSalary()
{
    return (“Tax” + 0.20 * gross + “Net Salary” + gross - (gross * 0.20));
}
```

**Figure 33 – Multiple values**
Source: Ndai Mapaso (2015)

The expression in the `return` string begins with a string value. This makes us interpret the `+` signs as concatenation operations so that the calculated tax and net salary values are converted to their equivalent string representation. The value returned from this can be expressed as:

```java
System.out.println (worker.calcTaxAndNetSalary());
```
1.4.2.13 Class attributes

It is possible for two or more objects to share attributes. In some languages, such as Java, C# and C++, this is done by declaring an attribute as static.

If an attribute is declared as static, it is allocated a single piece of memory for all objects instantiated from a class. It can still exist even if no object is instantiated. All objects of a class use the same memory location.

In Java, a static variable belongs to a class as a whole. Every object thus shares the same static variable. static variables exist outside of any specific object; Java provides a way of accessing such independently of any object. static methods can also be invoked independently of any object and such methods can perform any tasks not associated with any object. One of these tasks is to start program processing in the main() method before an object is created.

When a static variable is accessed as a data member of a specific object, it is prefixed by the object name. In the statement private static int count; count is defined as private and is used within its class (Figure 34):

```java
package staticnumber;
import java.util.Scanner;
/**
 * @author Karabo
 */
public class StaticNumber {
    private static int count;
    private int number1;
    private int number2;
    public StaticNumber(int n1, int n2) {
        count = 0;
        number2 = n2;
        number1 = n1;
    }
    public int addNumbers() {
        return count = number1 + number2;
    }
    public int valueOfCount() {
        return count;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
```
If a static variable is a public variable, it can also be accessed by prefixing it with a class name when it is used outside of its defining class (as shown in the program). For example, outside of its class, in the `OutsideClass`, the variable `count` is accessed by prefixing it with its class name, `StaticNumber`:

```java
return number + StaticNumber.count;
```

Figure 35 illustrates this point:

```java
package staticnumber;
import java.util.Scanner;
/**
 * @author Karabo
 */
public class StaticNumber{
  public static int count;
  private int number1;
  private int number2;
  public StaticNumber(int n1, int n2){
      count = 0;
      number2 = n2;
      number1 = n1;
  }
  public int addNumbers()
  {
      return count = number1 + number2;
  }
  public int valueOfCount()
  {
      return count;
  }
  /**
   * @param args the command line arguments
   */
```
```java
public static void main (String[] args)
{
    int num1;
    int num2;
    Scanner input = new Scanner (System.in);
    System.out.println (“Enter the first number”);
    num1 = input.nextInt();
    System.out.println (“Enter the second number”);
    num2 = input.nextInt();
    StaticNumber ob1 = new StaticNumber (num1, num2);
    StaticNumber ob2 = new StaticNumber (num1, num2);
    OutsideClass outClass = new OutsideClass();
    System.out.println (“Count value for static Count in object 1 is” + ob1.addNumbers());
    System.out.println (“Count value for static Count in object 2 is” + ob2.valueOfCount());
    System.out.println (“Count value for static Count in object number 2 is” + outClass.addNumbers());
}
}
```

**Figure 35 – static variable**

Source: Weisfeld (2013:67)

To understand the scope of this static variable, run the two programs and note the value of count.

### 1.4.3 Inheritance

Inheritance is a core principle of object oriented programming; such allows a class to inherit all the attributes and behaviours of another class. This allows you to ‘specialise’ a class by inheriting from it and provides additional functionality (attributes and behaviours).

Inheritance is a fundamental concept of object oriented programming. Under inheritance new classes can be created based on existing classes. When inheriting from an existing class, you re-use (inherit) methods and attributes as well as add new methods and attributes to adapt to the new class situation.

An existing class is referred to as a ‘superclass’ whereas a new class is a ‘subclass’ or ‘derived class’. A derived class is a totally new class; it incorporates all the variables and methods of the superclass or ‘base class’.
We can distinguish between **simple inheritance** and **multiple inheritance**.

In simple inheritance, each derived class has only a **single immediate base type**, for example, a `Circle` can be the base type and `Sphere` and `Cylinder` could be the child classes. In this case, `Sphere` is inheriting from `Circle`.

In multiple inheritance, each derived class has **two** or more base types, for example, `Minivan` can inherit from both `Car` and `Truck`.

Figure 36 illustrates simple and multiple inheritance:

![Figure 36](image)

Class derivations of both single and multiple inheritance are referred to as ‘class hierarchies’ as they show hierarchy or the order in which **one** class is derived from another.

Inheritance is implemented differently by the different object oriented programming languages. In Java, the syntax is:

```java
public class deriveClassName extends baseClassName
```

If `Circle` was the base class, the syntax would be (Figure 37):

```java
public class Cylinder extends Circle
{
    //add variables
    //add methods
}
```

![Figure 37](image)
1.4.3.1 Extending classes

We will again use the Employee class to explain inheritance.

Suppose you work for a civil engineering firm as a programmer. The core business of the company is water works. Engineers are given an extra bonus after each successful project. The rest of the employees only receive a salary. The engineers are regarded as employees and thus also receive a salary. Therefore, both engineers and all other employees (including you as a programmer) receive a salary from the company.

In the afore-mentioned scenario, define a new class Engineer. Retain all that has been programmed in the Employee class as well as all the attributes that are in said class. There is a ‘is-a relationship’ as an Engineer is an Employee.

In Java, define the Engineer class that inherits from Employee as follows (Figure 38):

```java
class Engineer extends Employee {
    //add its attributes and methods here
}
```

![Figure 38 – Inheritance](source: Ndai Mapaso (2015))

All inheritance in Java is considered public. The keyword extends shows that you are creating a new class that derives from an existing class. In the example, add a new variable to cater for a bonus (Figure 39):

```java
class Engineer extends Employee {
    private double bonus;
    public Engineer (double b) {
        bonus = b;
    }
}
```

![Figure 39 – Bonus](source: Ndai Mapaso (2015))

In the method declaration section, provide a constructor (Figure 40):

```java
public Engineer (String s, String n, double g, double b) {
    super (s, n, g);
    bonus = b;
}
```

![Figure 40 – Method declaration section](source: Ndai Mapaso (2015))
Use the keyword `super (super (s, n, g);)` to call the constructor of the `Employee` superclass with `s, n, g` for `surname, name` and `gross pay`. The `Engineer` constructor **cannot** access the private attributes of the `Employee` class; it must initialise through the constructor. The `super` syntax invokes the constructor in `Employee`. In the constructor, the call using `super` must be the first statement in the subclass constructor.

For the engineers to get a bonus, create a method to add bonus to the net salary as calculated in the `Employee` class (Figure 41):

```java
public double addBonus()
{
    double newpay = 0;
    newpay = bonus + super.calculateNet();
    return newpay;
}
```

**Figure 41 – Bonus**

Source: Ndai Mapaso (2015)

The method `addBonus()` adds the bonus to the net salary that is calculated by the method `calculateNet()`. To access this public method in the subclass `Engineer`, use the keyword `super`:

```java
super.calculateNet();
```

The compiler will know that the method is in the superclass.

Now suppose that a tax rate of 45% is charged on the gross if the bonus is more than R7500.

`addBonus()` will call the method `calculateNet()` from the superclass `Employee` if the bonus is less than R7500 and use the normal tax rate for each employee (20%). If the bonus is above R7500, the method will call the accessor method `getSalary()` in order to effect the new tax rate of 45% on the given gross. In this example, we calculate the tax directly using the following formula (Figure 42):

```java
newpay = bonus + (super.getSalary() * 0.55); //depending on preference formula
it is similar to
newpay = bonus + (super.getSalary() - (super.getSalary() * 0.45));
```

**Figure 42 – Bonus**

Source: Ndai Mapaso (2015)
Now create another method to show the values (Figure 43):

```java
public void showValues()
{
    System.out.println ("Employee Surname" + super.getSurname());
    System.out.println ("Employee Gross" + super.getSalary());
    System.out.println ("Employee Bonus" + bonus);
    System.out.println ("Initial pay before bonus" + super.calculateNet());
    System.out.println ("Employee Netpay" + addBonus());
}
```

Figure 43 – Method
Source: Ndai Mapaso (2015)

To get access to the private attributes `surname` and `gross pay` in the `Employee` class, whilst in the `Engineer` class, use the accessor methods `getSurname()` and `getSalary()`.

To display the bonus, call `addBonus()`:

System.out.println ("Employee Netpay" + addBonus());

To instantiate the method in `main()` (Figure 44):

```java
public static void main (String[] args)
{
    //declare variables
    //input statements
    Engineer worker = new Engineer (s, n, grossPay, bonusAmount);
    worker.showValues();
}
```

Figure 44 – main() method
Source: Ndai Mapaso (2015)

Figure 45 illustrates a class diagram for the inheritance program:

![Class diagram](image)

Figure 45 – Class diagram
Source: Ndai Mapaso (2015)
1.4.3.2 Abstract classes

An abstract class is a type of class that can only be used as a base class for another class; such thus cannot be instantiated. To make a class abstract, the keyword `abstract` is used. Abstract classes may have one or more abstract methods that only have a header line (no method body). The method header line ends with a semicolon (;). Any class that is derived from the base class can define the method body in a way that is consistent with the header line using all the designated parameters and returning the correct data type (if the return type is not `void`). An abstract method acts as a place holder; all derived classes are expected to override and complete the method.

If a class contains an abstract method, it should be defined as an abstract class. An abstract class may or may not have abstract methods. Non-abstract classes that are derived from abstract classes should override all abstract class methods. Because all derived classes from abstract classes have to implement methods, none of the abstract methods can be defined as `static` or `final`.

Figure 46 illustrates a class diagram for an abstract class:

![Abstract class diagram](image)

**Figure 46 – Abstract class**

Source: Weisfeld (2013:145)

The above class diagram has the same notation as inheritance, however, the methods and class name in the abstract classes are in italics.

Figure 47 illustrates an example in Java:

```java
abstract public class Shape
{
    double area;
    public abstract double getArea();
}
```

**Figure 47 – Java example**

Source: Ndai Mapaso (2015)
getArea() is defined as an abstract method, therefore, any derived classes will define and implement such. If Triangle or Rectangle were the derived class, it would be defined as follows (Figure 48):

```java
class Rectangle extends Shape
{
    int height;
    int width;
    public Rectangle (int h, int w)
    {
        height = h;
        width = w;
    }
    public double getArea()
    {
        area = height * width;
        return area;
    }
}
```

**Figure 48 – getArea() method**  
Source: Ndai Mapaso (2015)

To instantiate the object in `main()`, do the following (Figure 49):

```java
public static void main (String[] args)
{
    //declare input variables
    //input data
    //instantiate classes
    Rectangle rect = new Rectangle (h, w);
    System.out.println (“Area of Rectangle” + rect.getArea());
}
```

**Figure 49 – main() method**  
Source: Ndai Mapaso (2015)

The Shape class is not instantiated. Instead it is only Rectangle that is instantiated to create the object rect.

An abstract class can also provide a concrete method. An example could be a method used to choose a consistent darker colour for all shapes (Figure 50):

```java
abstract public class Shape
{
    double area;
    public void decideColor()
    {
        System.out.println (“The shade of every shape should be a darker one”);
    }
    public abstract double getArea();
}
```

**Figure 50 – Concrete method**  
Source: Ndai Mapaso (2015)

The method decideColor() is a concrete method.
1.4.3.3 Interfaces

An interface is a special type of inheritance; such provides abstract methods only. UML uses italics for methods in interfaces. Notations use dotted lines that connect with arrow heads. In Figure 51, the Shape class implements the Shape interface:

![Interface relationship](image)

**Figure 51 – Interface relationship**

Source: Reges and Stepp (2014:647)

An interface consists of constant and abstract methods. The keyword `interface` is used. Figure 52 illustrates the structure of an interface:

```java
interface interfaceName {
    constant declaration
    abstract method declaration
}
```

**Figure 52 – Interface**

Source: Weisfeld (2013:149)

A constant can be defined as `public`, `static` or `final`. Abstract methods in an interface must be `public` and `abstract`.

Code for the interface `Shape` would be (Figure 53):

```java
public interface Shape {
    public String showLabel = "Shape";
    void draw();
    double getArea();
}
```

**Figure 53 – Shape interface**

The keyword `implements` is used to show that a class is derived from an interface. The general syntax here is (Figure 54):

```java
class ClassName implements InterfaceName {
    //override all abstract method definitions
}
```

**Figure 54 – Interface syntax**  

Code for the `Circle` class would be (Figure 55):

```java
public class Circle implements Shape {
    private double radius;
    public Circle (double radius) {
        this.radius = radius;
    }
    public void draw() {
        System.out.println (“Drawing a circle”);
    }
    @override
    public double getArea() {
        return Math.PI * this.radius * this.radius;
    }
}
```

**Figure 55 – Circle class**  

The keyword `this` directs a compiler to access an object variable’s radius, not the variable that is in the compiler argument list.

### 1.4.3.4 Object: the superclass

Every class in Java extends `Object`, although we never get to express it as `Class Employee extends Object`. There are different services that `Object` performs, for example, the `equals` method in the `Object` class is used to test whether or not two objects point to the same memory.

An important method in `Object` is `toString`; this returns a string that represents the value of an object. A superclass creates a string as follows:

```java
String answer = new String();
```
A newly created string is called ‘answer’. Each subclass should define its own `toString` method and add its variables. To combine the subclass string and the superclass string, the subclass will call `super.toString()`.

Consider the class diagram in Figure 56:

![Class diagram](image)

Figure 56 – toString method  
Source: Ndai Mapaso (2015)

In the class `College` the `toString` method would be (Figure 57):

```java
public String toString()
{
    String answer = new String();
    answer = “College name” + collegeName;
    return answer;
}
```

Figure 57 – College class  
Source: Ndai Mapaso (2015)

In the class `Lecturer` the `toString` method would be (Figure 58):

```java
public String toString()
{
    return “College name” + super.toString() + “Department” + department;
}
```

Figure 58 – Lecturer class  
Source: Ndai Mapaso (2015)

### 1.4.4 Polymorphism

Polymorphism is an advantage of object oriented programming. It is a Greek word that means ‘many shapes’. Each object should have a method defined to respond to a message that is sent to it. Each subclass is a separate entity and thus might require different responses to the same message. A base member method can be overridden using overloaded derived member methods. Overriding constitutes replacing an implementation of a parent with a child. The overloaded method to call is determined during run time; this is called ‘dynamic binding’.
1.4.4.1 **Operator overloading**

There are some object oriented programming languages that allow you to overload operators. An example is C++. The + operator can be overloaded to perform string concatenation.

1.4.4.2 **Method overloading**

When a class has multiple methods with the same name but different parameters, it is known as ‘method overloading’. There are different ways of overloading a method, namely:

- Changing the number of arguments
- Changing the data type
- The order of parameters

After an overloaded method has been loaded onto Java, a compiler will select an appropriate method by analysing the type, order or number of arguments in the parameter list.

Method overloading **cannot** be made possible by changing the return type of a method. It will **not** be possible for a compiler to determine which version of a method to call.

1.4.4.3 **Method overriding**

Method overriding is when a subclass implements a new version of a method thus replacing code that could have been inherited from a superclass. A child class may need to replace some of the standard base class behaviour with its own.

If you want to override a method, write said method from the base class that you want to replace in the subclass. There is no special syntax required, except that the method’s name and signature must match those in the base class. There must be a is-a (inheritance) type of relationship between the **two** classes (Reges & Stepp, 2014:609).
Figure 59 illustrates that the `Reptile` class has a `move()` method, however, `Crocodile` wants to specify the actual speed:

```java
public class Reptile {
    public void move() {
        System.out.println("Moves very fast");
    }
    public static void main(String[] args) {
        Crocodile croco = new Crocodile();
        Lizards lizard = new Lizards();
        croco.move();
        lizard.move();
    }
}

class Crocodile extends Reptile {
    public void move() {
        System.out.println("Moves at 30-50 km/hr");
    }
}
class Lizards extends Reptile {
    public void move() {
        System.out.println("Moves at 34.6 km/hr");
    }
}
```

**Figure 59 – Method overriding**

Source: [http://stackoverflow.com/questions/15995540/arraylist-containing-different-objects-of-the-same-superclass-how-to-access-me](http://stackoverflow.com/questions/15995540/arraylist-containing-different-objects-of-the-same-superclass-how-to-access-me)

Lizards can invoke the overridden method `move()` from the base class `Reptile` by using the keyword `super`. If you want a child class to include the inherited behaviour into its behaviour, in the `Lizards` class `move()` method, include the following (Figure 60):

```java
    super.move();
```

**Figure 60 – move() method**

Source: [http://stackoverflow.com/questions/15995540/arraylist-containing-different-objects-of-the-same-superclass-how-to-access-me](http://stackoverflow.com/questions/15995540/arraylist-containing-different-objects-of-the-same-superclass-how-to-access-me)
You cannot override a static method because such is bound to a class whereas instance methods are bound to objects. A method can be re-declared.

Table 7 summarises the method overloading and overriding concepts:

<table>
<thead>
<tr>
<th>Method overloading</th>
<th>Method overriding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to improve program readability</td>
<td>Used when an object needs to implement its behaviour instead of behaviour from an inherited class</td>
</tr>
<tr>
<td>Performed in a class</td>
<td>Performed within two classes that have a is-a type of relationship</td>
</tr>
<tr>
<td>Parameters must be different</td>
<td>Parameters and return types must be exactly the same</td>
</tr>
</tbody>
</table>

Source: [http://docs.oracle.com/javase/tutorial/java/IandI/override.html](http://docs.oracle.com/javase/tutorial/java/IandI/override.html)

### 1.4.5 Encapsulation

Generally, classes are designed in such a way that data is hidden from other objects so that the latter only have access to functions. Encapsulation allows an interface to be visible to other objects (i.e. the methods and functions that are needed from outside the object can be seen), however, all data and any methods that are internal to the object can be hidden.

If the value of a hidden object’s field is required externally, you can add accessor methods to return the value. Accessor methods do not necessarily violate the encapsulation of an object. Accessor methods only return a copy of a field’s value to the calling object. The calling object will thus not be able to change the state of the object using accessor methods.

### 1.4.6 Relationships

Table 8 summarises UML association notations:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>One and only one</td>
</tr>
<tr>
<td>n</td>
<td>Exactly the specified number (where n is the integer)</td>
</tr>
<tr>
<td>0..1</td>
<td>Zero to one</td>
</tr>
<tr>
<td>m..n</td>
<td>m to n (where m and n are integers)</td>
</tr>
<tr>
<td>*</td>
<td>Zero to any positive integer</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero to any positive integer</td>
</tr>
<tr>
<td>1..*</td>
<td>One to any positive integer</td>
</tr>
</tbody>
</table>

Source: Bronson (2004:269)
Cardinality refers to the number of objects that correspond to a class. For example, consider a **Computer** class diagram that consists of **Motherboard** and **RAM chip**. A computer only has 1 motherboard. This cardinality is thus represented as 1. A computer can have more than 1 RAM chip but should have at least 1 RAM chip. Therefore, the cardinality can be represented as 1..* (Figure 61):

![Figure 61 – Cardinality in a UML diagram](Image)

Source: Weisfeld (2013:194)

Composition is, normally, used when inheritance cannot be implemented. There are **two** types of composition, namely:

1. Aggregation
2. Association

Aggregation can be implemented when a class represents a particular component of another class. This is a type of relationship where a class or object ‘consists of’ or is ‘composed of’ other classes or objects. We could use the example of a sentence as explanation: a sentence consists of words, which consists of characters. Therefore, characters are part of words, which themselves are part of sentences (Figure 62) (Bronson, 2004:269).

![Figure 62 – Cardinality in a UML diagram](Image)

Source: Bronson (2004:269)

With association, an object can offer its services to other objects: an object in an application can establish an association with another object and can then send a message requesting services from said object. An association between two objects thus has to be established before objects can communicate with each other.

Association is represented by a straight line connecting **two** classes or **two** objects. Association is often signified by phrases such as ‘is related to’, ‘is associated with’, ‘has a’, ‘is employed by’ and ‘works for’.
1.4.6.1 Dependence relationships

A dependence relationship is the most general relationship. An example would be the Order class that uses the Account class as Order objects need to access Account objects to check credit status. The Item class does not depend on the Account class as Item objects do not process customer accounts. Truck uses TruckSchedule. In other words, a class depends on another class if its methods use objects in the latter (Figure 63):

![Figure 63 - Dependence](Source: Horstmann and Cornell (2002))

Inheritance or an ‘is-a’ relationship shows a relationship between a base class and subclass. Aggregation has a ‘has-a’ relationship.

Concluding remarks

The design of software systems is fundamental to the production of quality software products. Therefore, designing a system properly is the key to the development of successful software systems. It is apparent that we cannot talk about designing software systems without object oriented programming concepts. To aid you in your understanding of object oriented programming concepts, we have applied such to real-world systems.

Self-assessment

Test your knowledge

1. Describe the different types of access that a class member can have.
2. What is a constructor?
3. What is a destructor?
4. What are constructors used for?
5. What is a class?
6. Can a class have many constructors?
7. What types of member can a class have?
8. What are the characteristics and methods of a class?
9. Explain abstraction and generalisation.
10. Is there any relationship between polymorphism and abstraction? What about polymorphism and inheritance?
11. What is operator overloading? Is such possible in Java?
12. Explain, in detail, why private members of a base class are not accessed directly by a derived class.
13. What is multiple inheritance? What are the advantages and disadvantages of such a relationship?
Unit 2: System Development

Unit 2 is aligned with the following learning outcomes and assessment criteria:

**Learning outcomes**

<table>
<thead>
<tr>
<th>LO2</th>
<th>Design object oriented programming solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO3</td>
<td>Implement object oriented programming solutions</td>
</tr>
<tr>
<td>LO4</td>
<td>Test and document object oriented programming solutions</td>
</tr>
</tbody>
</table>

**Assessment criteria**

<table>
<thead>
<tr>
<th>AC2.1</th>
<th>Identify the objects, data and file structures required to implement a given design</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC2.2</td>
<td>Design an object oriented programming solution for a given problem</td>
</tr>
<tr>
<td>AC3.1</td>
<td>Implement an object oriented programming solution based on a prepared design</td>
</tr>
<tr>
<td>AC3.2</td>
<td>Define the relationships between objects in order to implement design requirements</td>
</tr>
<tr>
<td>AC3.3</td>
<td>Implement object behaviours using control structures in order to meet design algorithms</td>
</tr>
<tr>
<td>AC3.4</td>
<td>Make effective use of IDEs, including code and screen templates</td>
</tr>
<tr>
<td>AC4.2</td>
<td>Analyse actual test results against expected results to identify discrepancies</td>
</tr>
<tr>
<td>AC4.3</td>
<td>Evaluate independent feedback on a developed object oriented programming solution and make recommendations for improvement</td>
</tr>
<tr>
<td>AC4.5</td>
<td>Create documentation for the support and maintenance of computer programs</td>
</tr>
</tbody>
</table>

**Learning objectives**

After studying this unit, you should be able to:

- Understand how classes collaborate with each other
- Understand inheritance in **Swing** applications
- Understand persistent objects
- Understand the object life cycle with persistence
- Understand the Java API
Prescribed reading


Introduction

There are two main ways in which you can re-use classes in Java object-oriented programs, namely:

1. Composition
2. Inheritance

The conceptual design of the above are explained via class-responsibility-collaboration (CRC) cards and UML class diagrams.

2.1 CRC

CRC is one of the most popular methods used to identify and categorise classes in a system. This method, essentially, keeps track of classes in an application. At the heart of CRC modelling lies use cases and scenarios from which objects and classes are derived to form the basis of object-oriented design.

CRC diagrams are used to show the allocation of responsibilities to classes as well as their necessary collaborations with other classes. There must thus be ways of keeping track of classes as well as their interactions.

Each CRC card represents a single class’s data attributes, responsibilities and collaborations. These cards can be non-electronic but there are some computer applications that can be used to model said cards. CRC cards can be described as a collection of standard index cards.

Three sections are created on each card, namely:

1. The name of the class
2. The responsibilities of the class
3. The collaborations of the class
2.1.1 Identifying classes

In general, nouns indicate possible classes. However, not all nouns that are identified can be made into classes. Some classes that are not identified as nouns in the initial check may be identified as you iterate the process of creating classes.

2.1.2 Identifying class responsibilities

Responsibilities can be identified by selecting verbs that are associated with classes or in a given scenario of a system. These responsibilities are actions. Verbs are not the only elements that can be used to construct responsibilities for a class.

When identifying class responsibilities, note the following:

- **Not** all verbs in the requirements summary will end up as responsibilities of a class.
- Several verbs may need to be combined to create an actual responsibility.
- Some responsibilities will only come later (not mentioned in the given summary).
- This is an iterative process thus you need to keep going back to the summary to update both the summary and responsibilities.
- If there are two or more classes that share a responsibility, each of those will have the responsibility.

A responsibility shows the knowledge or information that a class possesses. Said knowledge or information can either be stored within attributes or requested via collaboration with other classes. It is possible for a responsibility to correspond to one or more operations. In some cases it will be difficult to determine the actual responsibilities as it will appear as if there are other alternatives and they, in turn, appear correct. At the beginning of system design, CRC diagrams can be used to produce initial diagrams.

Table 9 indicates the structure of a CRC diagram:

<table>
<thead>
<tr>
<th>Class name:</th>
<th>Responsibilities:</th>
<th>Collaborations:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class responsibilities are listed in this section</td>
<td>Collaborations with other classes are listed in this section (include a brief description of the purpose of each collaboration)</td>
</tr>
</tbody>
</table>

Source: Weisfeld (2013:112)
2.2 UML class diagrams

UML class diagrams are used to represent objects in a standard way. UML is used in object oriented system development (virtualising, specifying, constructing and documenting system elements). There are different UML constructs.

Program modelling is the design of applications. UML is a program modelling language that consists of its own set of rules and notations.

To describe classes and their relationships, we use class diagrams. Object diagrams describe specific objects as well as their relationships. Remember that a class is referred to as a type of object wherein many specific objects can be created. An object thus refers to a specific single item as created from a class.

2.2.1 Class diagrams

A class diagram is represented by a box that is divided into three parts, namely:

1. Class name
2. Attributes
3. Methods

The top part of a class diagram contains the class name. The name of a class begins with a capital letter.

2.2.2 Attributes

Attributes fall in the middle section of class diagrams. Constructors are put in the same section as methods as they are considered to be methods. In Java, attributes that are listed in class diagrams will become the variables that are declared in the class data declaration section.

Each attribute has two qualities, namely:

1. Type
2. Visibility

Attribute type can be either a primitive data type, such as integer, double, real, Boolean or character, or a class type, such as string. Type should be included in class diagrams; such follows after a mandatory colon (:).
To the left of an attribute or method name there is a visibility indicator that determines where such can be seen, i.e. whether it can be used by other classes or whether it is restricted to the class defining it.

Public visibility is shown by (+), private by (-) and protected by (#).

2.2.3 Methods

Methods fall in the last section of class diagrams. The same logic as used on attributes applies to methods. Methods in a class diagram express return type rather than type. A method will have an access modifier, method name and return type. Every method name is followed by a colon (:) that separates it from the return type.

Figure 64 illustrates a Person class diagram:

![Person class diagram](image)

**Figure 64 – Person class diagram**  
Source: Ndai Mapaso (2015)

In the Person class diagram, there are name, address and age private attributes. name and address are of type String and age is of type int. The constructor Person() has no return type. In said class diagram, all other methods should have a return type except for constructors.
The program in Figure 65 directly translates the class diagram in Figure 64 into code:

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

    public Person (String nam, String add, int ag) {
        name = nam;
        address = add;
        age = ag;
    }

    public String getName() {
        return name;
    }

    public void setName (String s) {
        name = s;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress (String a) {
        address = a;
    }
}
```

**Figure 65 – Person class diagram**

Source: Ndai Mapaso (2015)

### 2.2.4 Object diagrams

In an object diagram, the class name is optional. If a class name has been included it is underlined. When a class name is given, an optional object name can be provided to precede the class name. These are both separated by a colon ( : ).

An object diagram shows an instance of a class. When translating a diagram into code, use the class and object names given in the class and object diagrams.
Figure 66 illustrates an object diagram for the Person class:

![Person class object diagram](image)

**Figure 66 – Person class object diagram**

Source: Ndai Mapaso (2015)

The object name is `per`.

The code in Figure 67 is an implementation of the class diagram and the instance of the class/object diagram, as discussed:

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

    public Person(String nam, String add, int ag) {
        name = nam;
        address = add;
        age = ag;
    }

    public String getName() {
        return name;
    }

    public void setName(String s) {
        name = s;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String a) {
        address = a;
    }

    public void main(String[] args) {
        Person per = new Person(“Elton”, “Long Street”, 19);
    }
}
```

**Figure 67 – Person class**

Source: Ndai Mapaso (2015)
Figure 68 summarises the UML notations for class relationships:

![UML Notations Diagram](image)

2.3 Re-using objects

2.3.1 Interactive dialogue input

In Java, there is a GUI method used to enter user data: `showInputDialog()`, which is in the `JOptionPane` class. When this method is called, a dialogue box is created; said box allows the user to enter a string. The string is then converted to a primitive data type value:

```java
JOptionPane.showInputDialog (string)
```

The string argument is a prompt that is displayed inside the dialogue box, for example:

```java
String num = JOptionPane.showInputDialog ("enter the number");
```

If the user presses **Cancel**, a null value is returned. The statement:

```java
sum = Double.parseDouble (num2) + Double.parseDouble (num);
```

converts the strings to data type `Double`.

2.3.2 GUIs

Java has a set of objects that can be used to create GUI applications. These objects, which are referred to as ‘components’, were originally in package `java.awt` (abstract window toolkit). A package – `javax.swing` – with much simpler tools was subsequently released. It can be said that the latter package is based on the former.
There are generic types of object used to create GUIs. They are:

- **Top-level container**: this is a basic window structure that includes borders and can be resized. It can hold intermediate containers and atomic components. Each GUI must have one of these to link the GUI with the operating system.

- **Intermediate container**: this is a graphical object that is displayed in top-level containers. It can hold other intermediate containers and atomic components. It is used to simplify the placement and exact positioning of atomic components.

- **Atomic components**: these are graphical objects used to display individual items of information. They must be placed in containers with the add() method.

Each GUI component created in Java has a placement level. Atomic components and containers are logically placed inside other appropriate containers. `JFrame` is a top-level container wherein a standalone Swing-base GUI forms the starting point of the hierarchy. A primary function of top-level containers is to interface with operating systems directly.

Table 10 summarises **awt** and **Swing** components:

<table>
<thead>
<tr>
<th>Component type</th>
<th>awt name</th>
<th>Swing name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level container</td>
<td>Frame</td>
<td>JFrame</td>
</tr>
<tr>
<td>Top-level container</td>
<td>Applet</td>
<td>JApplet</td>
</tr>
<tr>
<td>Top-level container</td>
<td>Dialog</td>
<td>JDialog</td>
</tr>
<tr>
<td>Top-level container</td>
<td>Window</td>
<td>JWindow</td>
</tr>
<tr>
<td>Intermediate container</td>
<td>Panel</td>
<td>JPanel</td>
</tr>
<tr>
<td>Intermediate container</td>
<td>InternalFrame</td>
<td>JInternalFrame</td>
</tr>
<tr>
<td>Intermediate container</td>
<td>RootPane</td>
<td>JRootPane</td>
</tr>
<tr>
<td>Intermediate container</td>
<td>LayeredPane</td>
<td>JLayeredPane</td>
</tr>
<tr>
<td>Atomic component</td>
<td>Button</td>
<td>JButton</td>
</tr>
<tr>
<td>Atomic component</td>
<td>Menu</td>
<td>JMenu</td>
</tr>
<tr>
<td>Atomic component</td>
<td>Label</td>
<td>JLabel</td>
</tr>
<tr>
<td>Atomic component</td>
<td>Checkbox</td>
<td>JCheckbox</td>
</tr>
<tr>
<td>Atomic component</td>
<td>Radio Button</td>
<td>JRadioButton</td>
</tr>
<tr>
<td>Atomic component</td>
<td>TextField</td>
<td>JTextField</td>
</tr>
<tr>
<td>Atomic component</td>
<td>TextArea</td>
<td>JTextArea</td>
</tr>
</tbody>
</table>

Source: Bronson (2004:469)

Intermediate containers and atomic components are referred to as ‘lightweight components’. They are called ‘lightweight’ because they are fully implemented in Java and do not need to invoke the operating system to draw GUI components.
There are many ways of constructing GUI windows via the Swing package. There are some predominant approaches to such; in this module, we will construct the GUI Separate class using Swing components. The main() method is issued to instantiate objects of this class.
Figure 69 illustrates how to combine some components to create a GUI screen. The code has comments to explain how to use some of the components:

```java
package myframe2;
import javax.swing.*;
/**
 * @author Makh
 */
public class MyFrame2 extends JFrame {
    public MyFrame2()
    {
        //if you are adding components to this frame, add them in the constructor
        //from JFrame the title bar is going to be inherited via super() using
        //JFrame (String)
        super ("My First Frame");
        //setBounds - the window is going to be 450 x 700 in size and
displayed at the x, y position which is 600, 400
        setBounds (600, 400, 450, 700);

        //exit the program when the frame is closed
        setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        //create components, e.g. buttons, labels
        JLabel label = new JLabel ("Name");
        JButton button = new JButton ("HELP");
        JPanel panel = new JPanel();

        //add components to a panel
        pane.add (label);
        pane.add (button);

        //with Swing use the content pane for frames
        getContentPane().add(pane);
        pack();

        //to display the frame setVisible to true
        setVisible (true);
    }
    /**
     * @param args the command line arguments
     */
    public static void main (String[] args)
    {
        MyFrame2 mf = new MyFrame2();
        //if you have not setVisible to true in the constructor, setVisible to true
        //in the main() method
        //mf.setVisible (true);
    }
}
```

Figure 69 – Components  
Source: Ndai Mapaso (2015)
2.3.3 Applets

An applet window is a container within the Swing package. It is a Java class that extends JApplet (the superclass of Swing applets). JApplet is an intermediate subclass of the ordinary Applet class. An applet is an awt component.

The life cycle of an applet is:

- **init()**: this method is required for any initialisation in an applet. It is called automatically the first time Java launches an applet.
- **start()**: this method is automatically called after a browser calls init(). It is also called whenever a user returns to the page containing the applet (if they had left such). It can be called many times unlike the init() method that is called only once.
- **stop()**: this method can be called repeatedly in the same applet. It is automatically called when a user moves off of the page containing the applet.
- **destroy()**: this method is automatically called when a browser shuts down normally.
- **paint()**: this method is called immediately after start() as well as any time an applet needs to repaint itself in a browser. paint() is inherited from java.awt.

Applets reside inside web pages thus there is no need to include a method for exiting such.

In Figure 70, paint() draws a rectangle and polygon:

```java
import java.awt.*;
import java.applet.Applet;

public class Apple extends Applet {
    private String comment;
    public void init() {
        comment = "draw a polygon by clicking";
    }

    public void paint (Graphics g) {
        g.drawString (comment, 300, 300);
        int[] x = {15, 5, 20, 25, 35};
        int[] y = {5, 30, 30, 35, 45};
        g.drawPolygon (x, y, 5);
        g.drawRect (5, 5, 4, 4);
    }
}
```

**Figure 70 – paint()**  
2.3.4 Adding event handlers

The Swing package handles events by using a set of interfaces called ‘event listeners’. When a class is required to respond to a user event, it should implement the appropriate event handler that deals with such.

There are different registration configurations, including:

- **One** listener can be registered to **one** component
- **Three** listeners can be registered to **one** component
- **One** listener can be registered to **three** components

The java.awt.event package has basic event listeners and objects that represent specific events. Event listeners in Java include ActionListener, WindowListener, MouseMotionListener, etc. Each class associated with the afore-mentioned must handle all the methods associated with such.

**ActionListener** actions events that are generated by users. This interface has **one** method: actionPerformed(). This method is always invoked by ActionListener. An event is received as a parameter (Figure 71):

```java
public void actionPerformed(ActionEvent evt) {
    //handle the event here
}
```

*Figure 71 – ActionListener*  
Source: Bronson (2004:497)

The originating object notifies the registered listener by sending an **actionEvent** object.

Think back to how we declared a class that inherits from an interface. Use the **implements** keyword to define a class that implements a listener interface:

```java
public class name extends JFrame implements ActionListener
```

In our example, we want to create a frame that has **two** buttons (b1 and b2). If you click on either button, ActionListener through actionPerformed() should change the text on the frame title bar. Register the listener with the buttons using the following syntax (Figure 72):

```java
b1.addActionListener (this);
b2.addActionListener (this);
```

*Figure 72 – Syntax*  
Source: Ndai Mapaso (2015)
The event handling method receives the object; the object’s `getSource()` method is the most appropriate element to use to determine the component that sent the event. This is executed as follows (Figure 73):

```java
public void actionPerformed(ActionEvent evt)
{
  Object source = evt.getSource();
  if (source.equals(b1))
    setTitle("Town Club Road");
}
```

**Figure 73 – `getSource()`**  
Source: Ndai Mapaso (2015)

Since it is an object, we use the method `equals()` instead of `==` in the `if` statement.

Figure 74’s event handling approach is based on the observer design pattern category under ‘behavioural patterns’:

```java
package changeTitle;
import java.awt.event.*;
import javax.swing.*;
/**
 * @author Makh
 */
public class ChangeTitle extends JFrame implements ActionListener
{
  JButton b1 = new JButton("Long Street");
  JButton b2 = new JButton("Cavendish");
  
  public ChangeTitle()
  {
    Super ("Title Bar");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //registering the listener with the buttons
    b1.addActionListener (this);
    b2.addActionListener (this);
    JPanel panel = new JPanel();
    pane.add(b1);
    pane.add(b2);
    getContentPane().add(pane);
    pack();

    //to display the frame setVisible to true
    setVisible (true);
  }

  public void actionPerformed (ActionEvent evt)
  {
    Object source = evt.getSource();
    if (source.equals(b1))
      setTitle ("Fun, Fun");
    else if (source.equals(b2))
      setTitle ("Claremont");
  repaint();
```
public static void main (String[] args) {
    ChangeTitle frame = new ChangeTitle();
}

Figure 74 – Event handling approach
Source: Ndai Mapaso (2015)

Concluding remarks

At this point, you should be able to implement inheritance and design basic windows. Remember to test programs during development and before deploying systems to users.

Self-assessment

Test your knowledge

1. Draw a UML class diagram for a college:
   - The college has classes of students
   - Each class has a set of lecturers
   - Each lecturer lectures a set of modules
   - Students have a name and unique number in class
   - Classes have an unique text identifier
   - Lecturers also have names
   - Modules have a name, amount of classes and amount of exercises
   - Lecturers and students are people

2. Create CRC cards for a scenario in which a bank holds different types of account for its customers (who can be individuals or companies), namely, deposit accounts and loan accounts. All accounts have a customer, balance and interest rate. Deposit accounts allow deposits and withdrawals. Loan accounts only allow deposits.
Unit 3: System Testing and Documentation

Unit 3 is aligned with the following learning outcomes and assessment criteria:

**Learning outcomes**

LO2  Design object oriented programming solutions  
LO3  Implement object oriented programming solutions  
LO4  Test and document object oriented programming solutions

**Assessment criteria**

AC2.1  Identify the objects, data and file structures required to implement a given design  
AC3.1  Implement an object oriented programming solution based on a prepared design  
AC3.4  Make effective use of IDEs, including code and screen templates  
AC4.1  Critically review and test object oriented programming solutions  
AC4.2  Analyse actual test results against expected results to identify discrepancies  
AC4.4  Create onscreen help to assist computer program users  
AC4.5  Create documentation for the support and maintenance of computer programs

**Learning objectives**

After studying this unit, you should be able to:

- Implement different types of system testing  
- Understand and produce different types of system documentation  
- Understand how to create help pop-ups and menus in JavaScript  
- Understand how to use debugging tools in IDEs  
- Understand how to use the JavaDoc tool

**Prescribed reading**

**Introduction**

System testing ensures that systems meet the functional requirements of clients. The primary aim of a system test is to find errors and correct them before said system is used. There is no way to guarantee a system completely free of errors but following a specific approach will assist in removing system errors. System documentation describes the specific implementation of systems.

### 3.1 Version control

Version control is a system that allows you to:

- Keep track of your files over time
- Share source code and documents
- Save different versions

A version control system (VCS) allows you to compare changes made to a file later on; it provides a database that can be used to store all revisions made to a program by all developers working on such.

Large software development projects, with many developers/authors, require a VCS so that team members can easily track changes and avoid erroneous file updates.

VCSs should be used for the following reasons:

- It makes it easy for users to revert back to the most correct copy of a document, if need be. You can thus easily recover from a recent erroneous document or source code.
- It can be used to control source code and files.
- If you are working as a team, you will be able to track who made changes as well as the time of such.
- Team members can have consistent file names and directory structures.
- Deploying different versions of code to servers is made easy.
- Sandboxing: insurance against accidental permanent changes to files by making temporary changes in a separate area. You can then test and correct errors before ‘checking-in’ your changes.
- Developers can collaborate with ease.
- Synchronisation: the simultaneous development of different functions on a system by allowing team members to share files and stay up-to-date with the latest versions of available files.

Users, usually, create different folders on their respective machines to save files that they will be working with to. Errors could occur if a user forgets which folder is the most recent as they can then accidentally overwrite files that contain their most accurate work.
It is advised that you create an account on www.github.com or any other VCS that you have access to on campus. You can also download the GitHub app onto your tablet.

### 3.1.1 Basic VCS concepts

There are many different VCSs. They all use the same concepts but may differ slightly when labelling processes/concepts.

Table 11 summarises the main VCS concepts:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>A separate copy of a file or folder created for separate use. A branch will allow a user to create a copy or snapshot of a repository that can be modified in parallel without making any changes to the Main.</td>
</tr>
<tr>
<td>Change-set</td>
<td>Changes that are made after a commit; these changes are recorded as a change-set.</td>
</tr>
<tr>
<td>Check-in</td>
<td>Uploading a file to a repository (if there were any changes made to said file)</td>
</tr>
<tr>
<td>Check-out</td>
<td>Downloading a file from a repository</td>
</tr>
<tr>
<td>Client</td>
<td>The computer that a team member is connecting from.</td>
</tr>
<tr>
<td>Commit</td>
<td>In VCSs, all changes can be tracked. Individual changes made to a file can be submitted as a single collection. This collection of changes is what is referred to as a ‘commit’. Changes are made once a commit is made.</td>
</tr>
<tr>
<td>Conflict</td>
<td>There are some situations when a VCS may, between two revisions, be able to work out which changes are to be applied.</td>
</tr>
<tr>
<td>Diffing/Diff/Change/Delta</td>
<td>Being able to locate changes/differences between two files. You can use this option to see which lines of code changed and who changed them</td>
</tr>
<tr>
<td>Merging</td>
<td>When a user is comfortable with the changes made to a branch, you will make such part of the Main or Trunk. The new file versions will seamlessly merge with the Trunk.</td>
</tr>
<tr>
<td>Repository (repo)</td>
<td>A persistent data store for files. This is where files and the history of changes to such are stored.</td>
</tr>
<tr>
<td>Revert</td>
<td>Discarding your local changes and reloading the latest version from a repository</td>
</tr>
<tr>
<td>Server</td>
<td>The computer that stores a repository</td>
</tr>
<tr>
<td>Trunk/Main</td>
<td>The primary location of files in a repository</td>
</tr>
<tr>
<td>Update/Sync</td>
<td>When you want to synchronise your files with those in a repository</td>
</tr>
</tbody>
</table>

Source: [http://git-scm.com/doc](http://git-scm.com/doc)

### 3.1.2 Types of VCS

There are centralised VCSs and distributed (decentralised) VCSs. A centralised VCS has a single server with all versioned files. There are a number of clients that can check-out files from such a central point.
3.2 System testing

Testing and debugging have different goals, namely:

- Testing intends to find the presence of flaws in software systems
- Debugging aims to locate and correct such flaws

Finding program faults is not a simple process as a fault may not always be located at the point where the program produces an error or ‘failed’ message. To find a program fault, additional tests may have to be developed to reproduce the original fault. Going through a program manually, line by line, may help to trace a fault. Debugging tools can also help programmers to locate the source of a problem in pieces of code. These debugging tools are, in most cases, part of a set of language support tools that are integrated within an IDE; they can check if valid declarations and naming conventions have been met.

Unit testing can be used to produce quality software products. This type of testing can either be done manually or automatically (Table 12):

<table>
<thead>
<tr>
<th>Manual testing</th>
<th>Automatic testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performed without any testing automation tools</td>
<td>Uses tool support; executes test cases with automation tools</td>
</tr>
<tr>
<td>Tedious and time consuming</td>
<td>Test cases are run faster than when using human resources</td>
</tr>
<tr>
<td>A high number of human resources is required to run tests, hence, it requires huge investment in this regard</td>
<td>Less human resources are invested as most processes are automated</td>
</tr>
<tr>
<td>May not be as reliable as expected as some tests may not be performed with high precision owing to human error</td>
<td>Processes and results are highly reliable owing to the high levels of precision in automated systems</td>
</tr>
<tr>
<td>Non-programmable: no programming is executed to trace hidden faults or information</td>
<td>Programmable: sophisticated tests can be developed to trace hidden faults or information</td>
</tr>
</tbody>
</table>

Adapted from: [http://www.tutorialspoint.com/junit/](http://www.tutorialspoint.com/junit/)

3.2.1 JUnit testing

JUnit is a Java unit testing framework; it is used for automated testing. JUnit belongs to the unit testing frameworks collectively known as ‘xUnit’. It is an open-source framework used to write and run tests.
JUnit provides:

- Annotations to identify test methods (for example, @Test)
- Assertions for testing expected results (for example, `assertEquals(expResult, result);`)
- Code which executes functions/methods

JUnit tests can be performed automatically; it can check results and give immediate feedback. JUnit also shows test progress. If a test has failed, it will show the result in red; if passed, the result will be highlighted in green.

Use the following program to create a simple test (Figure 75):

```java
package square;
import java.util.Scanner;

public class Square
{
    public int Area (int side)
    {
        int area = 0;
        area = side * side;
        return area;
    }

    public static void main (String[] args)
    {
        Scanner input = new Scanner (System.in);
        Square shape = new Square();
        System.out.println (shape.Area(4));
    }
}
```

![Figure 75 – System test](source:Ndai Mapaso (2015))

To install JUnit, do the following:

- Right-click on the program that you want to test.
- Install JUnit if it has **not** been installed.
- Complete the installation process by following the simple instructions.
If JUnit is already installed, do the following:

- Right-click on the program that you want to test.
- Go to **Tools > Create Tests**.
- Select the options that you want from the following window (Figure 76):

![Figure 76 – JUnit options: NetBeans version 7.1](image)

(The above window will allow you to select frameworks that are available to you.)
Click **OK** and then select which JUnit version you want to use (Figure 77):

![JUnit Version Selection](image)

**Figure 77 – JUnit NetBeans version 7.1**

Source: Ndai Mapaso (2015)

JUnit will automatically generate the following code for you (Figure 78):

```java
package square;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class SquareTest
{
    public SquareTest()
    {
    }

    @BeforeClass
    public static void setUpClass()
    {
    }

    @AfterClass
    public static void tearDownClass()
    {
    }

    @Before
    public void setUp()
    {
    }

    @After
    public void tearDown()
    {
    }

    /**
     * Test of area method, of class Square
     */
    @Test
    public void testArea() {
```

---

© CTI Education Group
public void testArea()
{
    System.out.println ("area");
    int side = 0;
    Square instance = new Square();
    int expResult = 0;
    int result = instance.Area(side);
    assertEquals (expResult, result);
    //TO DO review the generated test code and remove the default call to fail
    fail ("The test case is a prototype.");
}

/**
 * Test of main method, of class Square
 */
@Test
public void testMain()
{
    System.out.println ("main");
    String[] args = null;
    Square.main (args);
    //TO DO review the generated test code and remove the default call to fail
    fail ("The test case is a prototype.");
}

/*
 * Test of area method, of class Square
 */
@Test
public void testArea()
{
    System.out.println ("area");
    int side = 0;
    Square instance = new Square();
    int expResult = 0;
    int result = instance.Area(side);
    assertEquals (expResult, result);
    //TO DO review the generated test code and remove the default call to fail
    fail ("The test case is a prototype.");
}

// Comment out all the calls to fail:
// fail ("The test case is a prototype.");
// Go to the test methods annotated as @Test.
// Note the following method (Figure 79):

// You can have a few lines edited to:
    int side = 4; int expResult = 16;

Figure 78 – JUnit code
Source: Ndai Mapaso (2015)
JUnit created `expResult` for your expected result. This is where you key in your expected result. For example, you want the `side` to be 4. The answer would be 16 from the formula `side * side`.

The statement `int result = instance.Area();` invokes the method `area()` from your source code and returns the value via `result`.

`assertEquals(expResult, result);` compares the expected result that you keyed in with the result from `area()`.

To run the test file, do the following:

- Right-click on the test file.
- Click on **Test File** or press `<Control>` + `<F6>`.
- If the expected result and the result from the method are similar, you will see the following window (Figure 80):

![Figure 80 – Expected result: NetBeans version 7.1](source.png)

- If the expected answer was, say 36, and the input `side` was set to 4, the output would be the following (Figure 81):

![Figure 81 – Result output: NetBeans version 7.1](source.png)

The above highlights that you have to check your formula in the method and correct such if it was **not** correct.
3.3 Onscreen help

Onscreen help is developed for software products and complex professional products, such as control equipment. There are different types of onscreen help, such as:

- The classic ‘Help’ function
- Onscreen wizards
- Onscreen book-type help (e.g. Acrobat)
- Onscreen videos
- Hypertext Markup Language (HTML) help
- Interactive tutorials (e.g. www.codeacademy.com)

Onscreen help is a field that is still being developed. Recent computer applications provide context-sensitive pop-up menus. Pop-up boxes can be created via JPopupMenu in JComponent. A pop-up triggers when a user clicks on a component that supports pop-up menus.

3.4 Error handling

When classes are produced for the first time, they will contain errors. Error detection and trapping can be included in code to help deal with such. There are different ways of handling errors, namely:

- Ignoring the errors
- Checking for potential errors
- Throwing exceptions

Most object oriented programming languages have a function called ‘exceptions’. Exceptions are unexpected events that occur in systems. If a variable is declared as an integer and we use console Scanner by calling `nextInt`, the program will throw an exception if the user keys in any value that is not an int (Reges & Stepp, 2014:298).

Figure 82 illustrates the structure of a try/catch block:

```
try
{
   //possible error in code
}

catch (Exception e)
{
   //code to handle the exception
}
```

**Figure 82 – try/catch block**

3.5 System documentation

When developing software systems, a lot of information has to be recorded and documented as such relates to the different aspects of system development.

System documentation includes all documents that describe a system, ranging from the requirements specification to the final acceptance test plan. Documents describing the design, implementation and testing of a system are essential if a program is to be understood and maintained. Similar to user documentation, it is important that system documentation is structured, i.e. with overviews that lead the reader into more formal and detailed descriptions.

According to Sommerville (2011:246), for large systems developed to customer specification, system documentation should include the following:

- A requirements document with an associated rationale.
- A document describing the system’s architecture.
- For each program in the system, a description of the architecture of that program.
- For each component in the system, a description of the functionality and interface of that component.
- Program source code listings, which should be commented; said comments should explain complex sections of code as well as provide a rationale for the coding method. If meaningful names and a structured programming style are used, much of the code should be self-documenting without the need for additional comments. This information is nowadays maintained electronically, rather than on paper, with selected information printed on demand.
- Validation documents, which may be required for quality assurance processes in an organisation.
- A system maintenance guide, which ensures that all representations are kept in step when the system changes.

For smaller systems, documentation is, usually, less comprehensive. Most developers will try to maintain a specification of the system, an architectural design document and the program’s source code.

Sommerville (2011:246) further states that the pressure of work results in modifications being set aside until it becomes very difficult to find what should be changed. The best solutions to such a problem is to:

- Support document maintenance with software tools, which record document relationships
- Remind software engineers when changes to a document will affect another
- Record possible document inconsistencies
Van der Linden (2004: 48) mentions that many Java software suites are documented via JavaDoc.

**Concluding remarks**

When a bug is detected early on in the software development life cycle, it can be addressed faster and at a lower cost. Documentation helps you to track designs and communicate such to other developers before implementing a system. This allows for valuable feedback on design early on, while it is still easy to make proposed changes.

**Self-assessment**

1. Describe the process of throwing exceptions in Java and how this mitigates errors.

2. When a **try** block consists of two or more associated **catch** blocks, which **one** of them will be triggered by an exception?

3. What are the advantages of using **catch** blocks?

4. What is an exception?

5. Explain how Java documents are used as part of system documentation.

6. List five methods that can be employed in system testing.
<table>
<thead>
<tr>
<th>Glossary</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>A class that is <strong>not</strong> intended for instances; its purpose is to serve as a superclass for other classes. Abstract classes may contain <code>abstract</code> methods</td>
</tr>
<tr>
<td>Abstract data type</td>
<td>A data type that is defined by its operation rather than by its representation; representations are <code>private</code> and may only be accessed by defined operations</td>
</tr>
<tr>
<td>Abstract method</td>
<td>A method definition that consists of a method signature without a method body; such is marked with the keyword <code>abstract</code></td>
</tr>
<tr>
<td>Abstract subclass</td>
<td>A subclass that becomes concrete when it provides implementation for all its inherited <code>abstract</code> methods; if this is <strong>not</strong> the case, the subclass itself will be abstract</td>
</tr>
<tr>
<td>Application Program Interface (API)</td>
<td>An interface specified as a set of operations that allows access to an application program’s functionality</td>
</tr>
<tr>
<td>Autoboxing</td>
<td>This is performed automatically when a primitive type is used in a context that requires a wrapper type</td>
</tr>
<tr>
<td>Behaviour</td>
<td>The methods of a class implement its behaviour. A particular object’s behaviour is a combination of the method definition of its class and the current state of the object</td>
</tr>
<tr>
<td>Binary operator</td>
<td>An operator takes two operands. Java has many binary operators, such as the arithmetic operators <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code> and <code>%</code> as well as Boolean operators <code>&amp;&amp;</code> and `</td>
</tr>
<tr>
<td>Catching an exception</td>
<td>An exception is caught within the <code>catch</code> clause of a <code>try</code> statement. Catching an exception provides a program the opportunity to recover from a problem or to attempt to repair whatever caused such exception</td>
</tr>
<tr>
<td>Class constant</td>
<td>A variable defined as both <code>final</code> and <code>static</code></td>
</tr>
<tr>
<td>Coupling</td>
<td>This arises when classes are aware of each other because their instances must interact; linkage between two classes may be strong or weak</td>
</tr>
<tr>
<td>Debugging</td>
<td>An attempt to pinpoint and fix the source of an error</td>
</tr>
<tr>
<td>Design pattern</td>
<td>A description of a common computing problem as well as a small set of classes and their interaction structure that helps to solve the problem</td>
</tr>
<tr>
<td>Expression</td>
<td>A combination of operands and operators that produces a resulting value. Expressions have a resulting type, which affects the context in which they may be used</td>
</tr>
<tr>
<td>Fault detection</td>
<td>The use of processes and runtime checking to detect and remove faults in a program before such results in system failure</td>
</tr>
<tr>
<td>Inheritance</td>
<td>This allows the definition of one class as an extension of another</td>
</tr>
<tr>
<td>Inheritance hierarchy</td>
<td>Classes that are linked via inheritance relationships form an inheritance hierarchy</td>
</tr>
<tr>
<td>Interface</td>
<td>A specification of the attributes and operations associated with software components; an interface is used as a means of accessing component functionality</td>
</tr>
<tr>
<td>Message passing</td>
<td>Object interaction is characterised by message passing</td>
</tr>
<tr>
<td>Multiple inheritance</td>
<td>A situation wherein a class inherits from more than one superclass</td>
</tr>
<tr>
<td>Namespace</td>
<td>The area of a program in which particular identifiers are visible. Java uses packages to provide namespaces and their visibility rules, i.e. <code>private</code>, <code>package</code>, <code>protected</code> and <code>public</code></td>
</tr>
<tr>
<td><strong>Object construction</strong></td>
<td>The creation of an object, usually, via the <em>new</em> operator. When an object is created, an appropriate constructor from its class is invoked</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Object serialisation</strong></td>
<td>The writing of an object’s contents in such a way that its state can be restored, either at a later time or within a different process</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>A named grouping of classes and interfaces that provides a <em>package</em> namespace</td>
</tr>
<tr>
<td><strong>Pattern</strong></td>
<td>A recurring theme in class design/usage</td>
</tr>
<tr>
<td><strong>Re-use</strong></td>
<td>Inheritance allows the re-use of previously written classes in a new context</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>A language’s scope rules determine how widely variables, methods and classes are visible within a class or program</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td>The activity of finding out whether a piece of code (method, class or program) produces the intended behaviour</td>
</tr>
<tr>
<td><strong>Walkthrough</strong></td>
<td>The activity of working through a segment of code, line by line, while observing changes in state and other behaviour in an application</td>
</tr>
</tbody>
</table>
Bibliography


CTI is part of Pearson, the world’s leading learning company. Pearson is the corporate owner, not a registered provider nor conferrer of qualifications in South Africa. CTI Education Group (Pty) Ltd. is registered with the Department of Higher Education and Training as a private higher education institution under the Higher Education Act, 101, of 1997. Registration Certificate number: 2004/HE07/004. www.cti.ac.za.