COMPUTER SCIENCE
(ITC315113)

Information Booklet
CONTENTS

Java
Introduction ........................................................................................................................................................ 3
Basics............................................................................................................................................................. 4
Fundamentals .................................................................................................................................................. 5
Event driven programming ............................................................................................................................... 10
Creating user interfaces with the AWT ............................................................................................................ 11
Methods ............................................................................................................................................................ 14
Objects and classes ........................................................................................................................................... 16

Computer Fundamentals and Computer Limitations
Number Systems ............................................................................................................................................... 22
Arithmetic in the Binary System ...................................................................................................................... 23
Representation of Unsigned Integers ................................................................................................................ 23
Representation of Signed Integers .................................................................................................................... 24
Representation of Floating Point Numbers (Used For Reals) ........................................................................ 25
Representation of Characters and Strings ......................................................................................................... 26
Representation of Boolean ....................................................................................................................................... 26
ASCII Table ..................................................................................................................................................... 27
Representation of Arrays ...................................................................................................................................... 28
Representation of Images and Sounds .............................................................................................................. 29
Logic Circuits ................................................................................................................................................... 30
List of Logic Laws ........................................................................................................................................... 31
Karnaugh Maps ................................................................................................................................................ 32
Design of a logic Circuit ....................................................................................................................................... 37

Computer Architecture
The Toy Machine .............................................................................................................................................. 38
Structure of the CPU and the Machine Cycle ................................................................................................. 41
Java

Introduction

Java is platform independent. When you compile a Pascal program, it is translated into machine code or processor instructions specific to the processor on the machine that is running it. To use the program on another machine, a compiler must be obtained for that system. The Java compiler generates bytecodes. These instructions are not specific to one processor. The program is then executed by running a bytecode interpreter which reads the bytecodes and executes the program. This interpreter is often called the Java virtual machine.

For Java applets, the bytecode interpreter is built into Java-enabled browsers. For more general applications, an interpreter needs to be installed.

Java is an object-oriented programming language. It includes a set of class libraries that provide basic data types, system input and output capabilities and other utility functions.

Introduction - Hello World Applet

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

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello World", 5,25);  
    }
}
```

The HTML with the applet embedded

```html
<HTML>
<HEAD>
<TITLE>Computer Science Sample 1</TITLE>
</HEAD>

<BODY>
<APPLET CODE="HelloWorldApplet.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Major Applets Activities

Initialization - occurs when the applet is first loaded. Occurs only once.
Starting - A startup behaviour for the applet that may occur many times (eg if applet is stopped.)
Stopping - Occurs when the reader leaves the page containing the running applets.
Destroying - enables the applet to clean up after itself (i.e. free up system resources)
Painting - The way the applet draws on the screen.

- Web page is accessed and <applet> tag read - init( )
- Applet is completely loaded and ready to run - start( )
- User leaves Web Page - stop( )
- User exits browser - destroy( )
**Applet Parameters**

Values can be passed to applets by adding parameters to the HTML code.

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

public class HelloWorldApplet extends Applet {

    public void init () {
        name = getParameter("name");
        name="Hello " + name;
    }

    public void paint(Graphics g) {
        g.drawString(name,5,25);
    }
}
```

The HTML with the applet embedded

```html
<HTML>
<HEAD>
<TITLE> Computer Science Sample 1 </TITLE>
</HEAD>

<BODY>
<APPLET CODE="HelloWorldApplet.class" WIDTH =150 HEIGHT=25>
<Param Name=name VALUE="Babbage">
</APPLET>
</BODY>
</HTML>
```

NB The format of the command `g.drawString` is:

```
g.drawString (string_expression, x_coordinate, y_coordinate)
```

**Java Basics**

**Keywords**

<table>
<thead>
<tr>
<th>abstract</th>
<th>const</th>
<th>finally</th>
<th>int</th>
<th>public</th>
<th>throw</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>continue</td>
<td>float</td>
<td>interface</td>
<td>return</td>
<td>throws</td>
</tr>
<tr>
<td>break</td>
<td>default</td>
<td>for</td>
<td>long</td>
<td>short</td>
<td>transient</td>
</tr>
<tr>
<td>byte</td>
<td>do</td>
<td>goto</td>
<td>native</td>
<td>static</td>
<td>true</td>
</tr>
<tr>
<td>byvalue</td>
<td>double</td>
<td>if</td>
<td>new</td>
<td>super</td>
<td>try</td>
</tr>
<tr>
<td>case</td>
<td>else</td>
<td>implements</td>
<td>null</td>
<td>switch</td>
<td>void</td>
</tr>
<tr>
<td>catch</td>
<td>extends</td>
<td>import</td>
<td>package</td>
<td>synchronized</td>
<td>volatile</td>
</tr>
<tr>
<td>char</td>
<td>false</td>
<td>instanceof</td>
<td>private</td>
<td>this</td>
<td>while</td>
</tr>
<tr>
<td>class</td>
<td>final</td>
<td></td>
<td>protected</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Standard Packages and Classes

java.lang

Contains many standard classes including Integer, Double and math classes. This package is automatically available to programs and hence does not need to be imported.

java.awt


java.applet

This package contains one class Applet. It has all the functionality needed by an applet window.

java.awt.event

is used for implementing MouseListener, KeyListener and other event-driven program features.

java.awt.transfer

is used for transferring data to and from the clipboard.

java.io

is used for file input and output.

java.net

is used for performing network operations, such as sockets.

java.util

includes utilities for GregorianCalendar, Random and Vector.

Java Fundamentals

Variables and Data Types

Variable names start with a letter, underscore or dollar sign. Variable types can be one of the eight primitive data types, the name of a class or interface or an array.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Floating Point</th>
<th>Char</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte (8 bits)</td>
<td>float (32 bits)</td>
<td>char (Unicode)</td>
<td>boolean</td>
</tr>
<tr>
<td>short (16 bits)</td>
<td>double (64 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (32 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long (64 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constants

We could declare pi as a variable but we may also wish to stop it being accidentally changed. We accomplish this by including the word final.

final double pi=3.142;
Comments

The symbols /* and */ are used to surround comments or // can be used for a single line comment.

Expressions and Operators

+    addition    ==    equal to
-    subtraction  !=    not equal
*    multiplication <    less than
/    division     >    greater than
%    modulus      <=    less than or equal to
    (integer remainder)    >=    greater than or equal to
&&   AND
||   OR
^    XOR
!    NOT

(NB Result of / depends on the type of the original data.)

Promoting and casting

When dealing with expressions which contain different data types, Java will promote to the larger type and return an answer with that type.

e.g. 2 * 2.4 = 4.8

If we want the answer to be in the smaller type we cast down the hierarchy explicitly

e.g. (int)(2 * 2.4) = 4

Assignments

x=2+3;
x++; x--; (incrementing by one and decrementing by one)
x+=y; x-=y; x*=y; x/=y; (note: x+=y is the same as x=x+y)

Operator Precedence (Not a complete table)

.  []  ()
++  - -  !  ~  instanceof
new (type) expression
* /  %
+  -
<,  >,  <=,  >=
==,  !=
&&
||
? : (Shorthand for if .. then .. else)
=,  +=,  -=,  /=,  %=,  ^=

Example

import java.applet.*;
import java.awt.*;

public class calculationExampleApplet extends Applet {
    /* Sample Java applet involving calculations */
    public void paint (Graphics g) {
        g.drawString("5+6="+(5+6), 10,10);
        g.drawString("94.0/23.0"+(94.0/23.0),10,10);
    }
}
Mathematical Methods

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Note that methods of the same name may be defined for different types (int, long, float, double).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(int a)</td>
<td>Returns the absolute integer value of a</td>
</tr>
<tr>
<td>abs(long a)</td>
<td>Returns the absolute long value of a</td>
</tr>
<tr>
<td>abs(float a)</td>
<td>Returns the absolute float value of a</td>
</tr>
<tr>
<td>abs(double a)</td>
<td>Returns the absolute double value of a</td>
</tr>
<tr>
<td>cos(double)</td>
<td>Returns the trigonometric cosine of an angle</td>
</tr>
<tr>
<td>exp(double a)</td>
<td>Returns the exponential number e(2.718...) raised to the power of a</td>
</tr>
<tr>
<td>log(double a)</td>
<td>Returns the natural logarithm (base e) of a</td>
</tr>
<tr>
<td>max(int a, int b)</td>
<td>Takes two int values, a and b, and returns the greater of the two</td>
</tr>
<tr>
<td>max(long a, long b)</td>
<td>Takes two long values, a and b, and returns the greater of the two</td>
</tr>
<tr>
<td>max(float a, float b)</td>
<td>Takes two float values, a and b, and returns the greater of the two</td>
</tr>
<tr>
<td>max(double a, double b)</td>
<td>Takes two double values, a and b, and returns the greater of the two</td>
</tr>
<tr>
<td>min(int a, int b)</td>
<td>Takes two integer values, a and b, and returns the smaller of the two</td>
</tr>
<tr>
<td>min(long a, long b)</td>
<td>Takes two long values, a and b, and returns the smaller of the two</td>
</tr>
<tr>
<td>min(float a, float b)</td>
<td>Takes two float values, a and b, and returns the smaller of the two</td>
</tr>
<tr>
<td>min(double a, double b)</td>
<td>Takes two double values, a and b, and returns the smaller of the two</td>
</tr>
<tr>
<td>pow(double a, double b)</td>
<td>Returns the number a raised to the power of b</td>
</tr>
<tr>
<td>random()</td>
<td>Generates a random number between 0.0 and 1.0</td>
</tr>
<tr>
<td></td>
<td>eg. (int)(Math.random()*4+2) will give one of the values 2, 3, 4 or 5.</td>
</tr>
<tr>
<td>round(float)</td>
<td>Rounds off a float value by first adding 0.5 to it and then returning the largest integer that is less than or equal to this new value</td>
</tr>
<tr>
<td>round(double)</td>
<td>Rounds off a double value by first adding 0.5 to it and then returning the largest integer that is less than or equal to this new value</td>
</tr>
<tr>
<td>sin(double)</td>
<td>Returns the trigonometric sine of an angle</td>
</tr>
<tr>
<td>sqrt(double)</td>
<td>Returns the square root of a</td>
</tr>
<tr>
<td>tan(double)</td>
<td>Returns the trigonometric tangent of an angle</td>
</tr>
<tr>
<td>toDegrees(double)</td>
<td>Translates radians to degrees</td>
</tr>
<tr>
<td>toRadians(double)</td>
<td>Translates degrees to radians</td>
</tr>
</tbody>
</table>

Example

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

/* Sample Java applet involving mathematical functions */

public class Math_functions extends Applet {

    public void init() {
    }

    public void paint(Graphics g) {
        g.drawString("The larger number of 10 and 20 is "+Math.max(10,20), 50, 50);
        g.drawString("A random number between 1 and 6 is "+((int)(Math.random()*6)+1), 50, 80);
    }
}
```
Block Statements

Blocks in Java are enclosed between { }.

If … else Statement (Conditional)
if (hours>40) overtime=hours-40;
if (hours>40) 
overtime=hours-40;
else overtime=0;
if (hours>40) 
{overtime=hours-40;
 normal=40;}
else 
{overtime=0;
 normal=hours;}

Switch Conditional

switch (oper) {
case '+':
    g. drawString("5+6="+(5+6), 10, 10);
    break;
case '-':
    g. drawString("5-6="+(5-6),10,10);
    break;
case '*':
    g. drawString("5*6="+(5*6),10,10);
    break;
}

For Loops

for (initialisation; test; increment) {
    statements;
}
Eg. for (int i=1; i<=100; i=i+10)
    g. drawString("Computing is great", i,10);

While Loops

while (condition) {
    statements;
}

Do …. while loops

do {
    statements;
} while (condition);

Note:
- Can only be used with integer and char datatypes
- use of break to exit the switch statement and send control immediately after }
- can have a default case which handles all other values e.g.
  default : g. drawString("wrong operator", 10,10);
    break;

Note: tests before entering the loop each time

Note: test is performed before entering the loop - if the condition is false when the loop is first entered it will not be executed.

Note: test is performed after body of loop is executed so it is always performed at least once.
Arrays and Subscripts

You can create arrays of any type or object e.g. strings, buttons, images

```java
int xcoordinate[] = {0,10,20,30,40,50,60,70,80,90};
```
or

```java
int xcoordinate[] = new int[10];
xcoordinate[0]=0; xcoordinate[1]=10; etc
```

The array can be processed in loops
e.g.
```java
for (int i=0; i<10; i++) {
    g.drawString("*", xcoordinate[i],100);
}
```

Two Dimensional Arrays

```java
int scores[][ ] = new int[3][5];
```

```java
int scores[][] = {
    {93,74,77,55,81},
    {78,77,72,75,80},
    {92,88,82,83,69}
};
```

and processed within nested loops using variables of type scores [i][j]

Strings

(Note: single quotes are used around char type and double quotes around strings)

```java
String subject="Computer Science";
String addon="is great";
```

**Concatenation:** use +  (eg subject="Computer" + " " + "science")

**Comparison:** don’t use ==
```java
string1.equals(addon)
string1.equalsIgnoreCase(subject)
```

```java
string1.compareTo(string2) results in..
- 0 if the strings are equal;
- a negative value if the string object preceded the parameter;
- a positive value if the string object follows the parameter.
```

**Amending:**
```java
String2=string1.replace(‘e’,’a’)
String2=string1.toUpperCase()  string2=string1.toLowerCase()
subject.trim()  //used to move trailing or leading blanks
```

**Examining Strings:**
```java
subject.length() is 16
subject.substring(3,6);   //will return “put” - the characters from starting position 3 to 1 greater than the last character to be extracted. (The first character position is 0.)
subject.charAt(1);        //will give the character at position 1 (resulting in a char datatype)
subject.indexOf("put",2)  //will return 3, i.e. the position of the first letter of the substring "put", starting looking at position 2 – will return -1 if the string is not found.
subject.endsWith("r")    //would result in a true boolean result because the string ends with “r”
subject.beginsWith("X")  //would result in a false boolean result
```

Notice the subscripts begin at 0 i.e. the array goes from xcoordinate[0] to xcoordinate[9].
Converting Strings

to convert an int to a String:

    int n = 123;
    String s = Integer.toString(n);  //s becomes “123”

to convert a float to a String:

    float f = 12.34f;
    String s = Float.toString(f);  //s becomes “12.34”

to convert a String to an int:

    String s = “1234”;
    int n = Integer.parseInt(s);

to convert a String to a float:

    String s = “12.34”;
    Float temp = Float.valueOf(s);
    float f = temp.floatValue();

    //these lines could be replaced with
    float f = Float.valueOf(s).floatValue();

Event Driven Programming

A key component of a GUI is event-driven programming. Therefore the program needs to recognise an event and respond. The program is said to be listening for events. This is done using

`implements specific listener`

The listener interfaces include:

**ActionListener**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>(any component event)</td>
<td>public void actionPerformed(ActionEvent e)</td>
</tr>
</tbody>
</table>

**MouseListener**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>(left button down)</td>
<td>public void mousePressed(MouseEvent e)</td>
</tr>
<tr>
<td>(left button up)</td>
<td>public void mouseReleased(MouseEvent e)</td>
</tr>
<tr>
<td>(left button down then up)</td>
<td>public void mouseClicked(MouseEvent e)</td>
</tr>
<tr>
<td>(mouse enters component area)</td>
<td>public void mouseEntered(MouseEvent e)</td>
</tr>
<tr>
<td>(mouse exits component area)</td>
<td>public void mouseExited(MouseEvent e)</td>
</tr>
</tbody>
</table>

Once the interface has been added using `implements`, stubs are entered for all of the events in the interface.

e.g. if using MouseListener interface we need

```java
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
public void mouseClicked(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
```

Detail is then filled in the event which is of interest.

**ItemListener**

```java
public void itemStateChanged(ItemEvent e) { }
```
Finally, the listener must be registered.
e.g. addMouseListener()

Example: a program to draw an oval where the user clicks on the screen.
import java.applet.*;
import java.awt. *;
import java.awt.event.*;

public class event1 extends Applet implements MouseListener {
    int x=30;
    int y=30;

    public void init() {
        setBackground(Color.blue);
        addMouseListener(this);
    }

    public void mouseReleased(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent c) { }
    public void mouseExited(MouseEvent e) { }

    public void paint(Graphics g) {
        g.fillOval(x,y,50,50);
    }

    public void mousePressed(MouseEvent e) {
        x=e.getX();
        y=e.getY();
        repaint();
    }
}

Creating User Interfaces with the AWT

The major components of the AWT are

Containers - components that contain other components
Canvases - simple drawing surface
UI components - buttons, lists, pop-up menus, check boxes, text fields etc.
Window Construction components - windows, frames, menu bars and dialog boxes.

Applets are automatically AWT containers and hence components can be added to them.

Labels

Labels can be created in the following ways.

Label() Constructs an empty label.
Label(String) Constructs a new label with the specified string of text, left justified.
Label(String, int) Constructs a new label that presents the specified string of text with the specified alignment.
Label prompt;

    public void init()
    {
        prompt= new Label("Please enter number");
        add(prompt);
    }

Label Methods

    getText( ) Returns a string containing label text
    setText(String) Changes text of label
    getAlignment( ) Returns an integer representing alignment of the label
    setAlignment(int) Changes label alignment

Buttons

    public void init( ){
        add(new Button("Award=OA"));  or
        Button award;
        award=new Button("Award=OA");
        add (award);
    }

setLabel(String label) Sets the button's label to be the specific
getLabel() Gets the label of this button.

Making component invisible

    componentName.setVisible(true/false) will be visible if true, not visible if false

Check Boxes

    Used to allow a user to specify one or more choices by clicking on a box.
    Implements ItemListener, handles events with ItemStateChanged method.
    for example:
    Checkbox burger, drink;

    public void init () {
        burger = new Checkbox("Burger", true);  // sets original state of checkbox to true i.e., ticked
drink = new Checkbox("Drink");

        add(burger);
        add(drink);

        burger.addItemListener(this);
        drink.addItemListener(this);
    }

Check Box Methods

    Checkbox( ) Creates an empty checkbox
    Checkbox(String) Creates a checkbox with string as a label
    Checkbox(String, boolean) Creates a checkbox that is either selected or deselected
    getLabel( ) Returns the label as a string
    setLabel( ) Changes text of checkbox
    getState( ) Returns true or false based on whether checkbox is selected.
    setState(boolean) Changes state of check box
Radio Buttons (called Checkbox Groups in AWT library)

To add radio Buttons - used when there are a limited number of mutually exclusive choices e.g. credit card options - one of Visa, MasterCard or Bankcard
Implements ItemListener, handles events with ItemStateChanged method.

for example:
CheckboxGroup creditCardType= new CheckboxGroup( );

public void init() {
    visa=new Checkbox("Visa",creditCardType,false);
    add (visa);
    visa.addItemListener(this);
    ….similarly for other credit card options
}

Choice Menus

These are pop up or pull down menus. An item can be selected by clicking on it. The selected item appears as the only visible part of the menu.
Implements ItemListener, handles events with ItemStateChanged method.

for example:
public void init( ) {
    Choice icecream=new Choice( );
    icecream.add("Chocolate");
    icecream.add("Vanilla");
    icecream.add("Strawberry");
    add(icecream);
    icecream.addItemListener(this);
}

Choice Menu Methods

getItem(int) returns string item at the given position
getItemCount( ) returns the number of items in the menu
getSelectedIndex( ) returns the index position of the selected item
getSelectedItem( ) returns the currently selected item as a string
select(int) selects the item at a given position
select(String) selects the item with the given string

Lists

These are a list of text strings from which one or more can be selected. A scrollbar is provided to scroll up or down the list.
Implements ActionListener, handles events with actionPerformed method.
(This responds to double-clicks)

for example:
List colourList = new List(4,false);
colourList.add("red");
colourList.add("blue");

add(colourList);
colourList.addActionListener(this);

Two parameters: e.g. 4,false
- how many items are visible at once
- false ➔ only one item can be selected
  true ➔ more than one item can be selected
Scrollbar

Implements AdjustmentListener, handles events with adjustmentValueChanged method.

for example:
Scrollbar slider = new Scrollbar(Scrollbar.VERTICAL, initial, visible, min, max)
add(slider);
slider.addAdjustmentListener(this);

Text Fields

Allow you to enter and edit text.

TextField(int) creates text field of int characters wide
Textfield(String, int) creates text field containing string of int characters wide
Textfield(String) creates a text field containing string.

Text Field Methods

getText( ) returns the text that the text field contains
setText(String) puts the given text into the string
getColumns( ) returns the width of the text field
select(int, int) selects the text between the two positions
selectAll( ) selects all text in field
isEditable( ) returns true or false
setEditable(boolean) true enables text field to be edited
getEchoChar(char) returns the character used for masking input
setEchoChar(char) sets the character used for masking input
echoCharIsSet( ) returns true or false

Other features include canvases and cursors.

Methods

Methods have a number of important uses:

- They enable you to break up a class into “modules” which each perform a distinct task. Classes written in this way are much easier to implement and to follow.
- They enable the code for a repetitive task to be written once, and used many times in a class.
- In an object oriented language, they provide the means of using the information that is stored in an object. They also enable you to use the features of one class in another (such as when you call Math.random()).

The standard Applet methods

Three methods are run automatically in order. If we don't further define these we get an empty version by default: in this case, they do not have to be written in your class code – they run automatically anyway.

- init: init is used to set up the GUI (Graphical User Interface) by instantiating and adding components, define the initial values of variables, tell actionPerformed() which events to listen to and set up sounds and images to be used by the Applet.
- start: We will never use this method in this course. However, it can be modified for more advanced purposes.
- paint: This is used to display graphics objects. It runs automatically after the start () method, and you sometimes have to be careful not to write out “junk” this first time around. You can run paint again
(with updated values of variables, text etc) from any other method using `repaint()`. Note that outside of paint we can only use `showStatus` to display a small amount of information.

The general form of a method definition:

```java
public return_value_type method_name (formal parameters) {
  declarations; // These are local variables or objects.
  statements;
  return <expression>; // If required. Must be included if the method is not void.
}
```

**Scope Rules**

Variables can be declared anywhere however they are usually declared in one of three places as follows.

They can be declared just inside the class definition. These variables are known as **instance** or **global** variables and may be used within any method defined within the class. *(class scope)*

```
public class Methods extends Applet
{ TextField t1;
```

Variables can be declared in the parameter list of a method definition. They are known as **parameters** or **local variables** and are used within the method. *(Block scope)*

```
public void stars(int number)
```

Finally variables can be declared within a method definition and are known as local variables. They may only be used within the method.

```
public String stars(int number, char letter)
{ String temp;
```

When a local variable has the same name as an instance variable or a parent method, it is hidden until the child method terminates. The instance variable however can be accessed using this instance variable name.

**Passing parameters**

**Pass-by-value vs pass-by-reference**

When arguments of a **basic data type** are passed to a method they are **pass-by-value**. This means that for each argument passed to a method, a copy is made and it is the copy that is passed and referenced through the parameter made, not the original value. Consequently, whatever the method does to the parameter, it has no effect on the value in the invoking method.

```
e.g.
int i=10;
int x=change(i);
```

This method will return 11 but the original value of `i = 10` is unchanged.

```
public int change (int j)
{  j++;
    return j;
}
```

However, objects are different. When **objects** are passed to a method they are **pass-by-reference**. A reference is not a copy of the object, but a pointer to the object. So if and when a method changes an object parameter, it changes the object in the piece of the program that invoked the method.
Passing strings as parameters

Strings are objects and are therefore passed by reference. But, changing the value of a string is regarded as creating a totally new string, rather than manipulating an existing value via methods. The change therefore has only a local effect and we can think of strings as behaving like a basic data type being passed by value.

Passing arrays as parameters

An array is an object in Java. When an object is used as a parameter it is passed as a reference. **So when an array is passed as a parameter to a method, a reference to the array is passed** (i.e. a pointer to the array and **not a copy** of the array). So when a method changes an array parameter, it changes the array in the piece of program that invoked the method.

```java
int []table = new int[8];
fillZero(table);
```

**with the method:**
```java
public void fillZero(int[] array) {
    for (int s=0; s<array.length; s++)
        array[s]=0;
}
```

Data modelling (Encapsulation) in an object-oriented programming (OOP).

Object-oriented programming is an attempt to model computer programs as closely as possible on objects in the real world. Each object has variables that keep track of what is going on inside the object and methods to allow other objects to communicate with it.

Related data and methods are grouped together in an object in such a way that a programmer can use the object without having to worry about its details, or running the risk of destroying it.

Data and methods which the user does not need to worry about are “hidden” (declared as “private”) while “public” data and methods are available in a way which are simple and safe. In other words, objects should not be able to change (or even look directly at) each other’s instance variables but should use the object’s methods. It allows pre-defined data models such as String to be used without having to know how the string of text is stored or manipulated.

Objects and Classes

Object

- A combination of **data** (variables to keep track of what is happening) and **methods** (a process to let objects communicate) used on that data.
- Objects that we have dealt with include strings, the various applet components such as buttons, colors, fonts and so on.
- As an example if stringVar is an object of the String class, the data stored in stringVar is the text such as “Hello” and the methods are all the methods associated with the String class, such as stringVar.length();

Class

A general computer program which carries a “design” or “blueprint” or “template” for an object, and so enables a programmer to create or “instantiate” any number of objects of this class.
Instantiating an object

When you want to use an object in a class you must:

- **Declare** the object by giving it’s class and identifier name:
  
  `eg Button myButton;`

  By convention the class name begins with a capital while the object name begins lower case.

- **Instantiate** the object by calling one of the constructor methods for the object with the `new` method.
  
  Classes will often have a number of different constructors which take different parameters, and you can look up the constructors for different objects in the Java API help. The constructor method always has the same name as the class, but there may be any number of parameters. Examples are:

  `myButton = new Button("Hello");`

Creating a class

The class must have the following structure:

```
public class <name of Class>
{
    <declarations of variables to be used>
    <constructor method>
    <method declarations>
}
```

- A variable or method in a class can have one of 3 types of access:
  
  `public` accessible from any class.
  
  `private` accessible only from within this class.
  
  `protected` not used in this course.

  Remember that **local** variables are never accessible from outside a particular method.

- The prefix **final** means that a variable or method cannot be changed (variable) or overwritten (method).

- The **instance variables** declared in a class block are generally defined as `private` and can’t be used outside the class. They give a way of keeping track of the state of an object.

  `eg class Car
  {
    private int maxSpeed;
    private int weight;
    private String make;
  }

  Every Car created within the program must have a `maxSpeed`, `weight` and `make`.

- The **constructor method** is a special method used to create (instantiate) and initialise an object and it does not return a value. It has the **same name as the class**. It is recommended to set the instance variables to default values

  `eg public Car()
  {
    maxSpeed = 0;
    weight = 0;
    make = null;
  }

  If you want to specify initial values when an instance of an object is made, you **must** use parameters.

  `eg public Car(int maxSp, int wt, String mk)
  {
    maxSpeed = maxSp;
    weight = wt;
    make = mk;
  }

  So in `init()` this constructor would be used ✧ fredsCar = new Car(500, 1000, “Honda”);

- The **method declarations** are the methods that can be used on Objects from this class. They are blocks of code that can be called from outside the object and can take arguments and optionally return a value.
eg  [methodModifiers]  ResultType  methodName()
        {
          <  method body  >
        }

eg  public  int  getSpeed()
        {
          return  maxSpeed;  // where maxSpeed  is a class or instance variable
        }

The **method modifiers** allow you to say how restrictive the method will be. If it is public, any class can access the method.

**Continuing with the car theme**

```java
Car  fredsCar;    // declare the object
fredsCar = new Car();   // instantiate fredsCar
fredsCar.getSpeed();    // accessing one of the car methods by using the object
// followed by the dot operator
```

// **An Example of a complete Car program**

```java
public class  Car
        {
          private  int  maxSpeed;
          private  int  weight;
          private  String  make;

          public  Car()      // car constructor
          {
            maxSpeed = 0;
            weight = 0;
          }

          public  void  setMaxSpeed(int  speed)
          {
            maxSpeed = speed;
          }
          public  int  getSpeed()
          {
            return  maxSpeed;
          }
          public  void  setWeight(int  heavy)
          {
            weight = heavy;
          }
          public  int  getWeight()
          {
            return  weight;
          }
          public  void  setMake(String  carMake)
          {
            make = carMake;
          }
          public  String  getMake()
          {
            return  make;
          }
        }    // end Car
```
public class CarSample extends Applet
{
    Car fredsCar;
    public void init()
    {
        setBackground(Color.white);
        fredsCar = new Car();
        fredsCar.setMaxSpeed(500);
        fredsCar.setWeight(1000);
        fredsCar.setMake("Honda");
    }
    public void paint(Graphics g)
    {
        int speed;
        int weight;
        String make;
        speed = fredsCar.getSpeed();
        weight = fredsCar.getWeight();
        make = fredsCar.getMake();
        g.drawString("Fred's car Speed " + speed + "weight" + weight + "make" + make, 10, 20);
    }
} // end CarSample

A second example with 2 constructors for the Car

public class Car
{
    private int maxSpeed;
    private int weight;
    private String make;

    public Car() // car constructor 1
    {
        maxSpeed = 0;
        weight = 0;
    }
    public Car(int speed, int carWeight, String carMake) // car constructor 2
    {
        maxSpeed = speed;
        weight = carWeight;
        make = carMake;
    }
    public void setMaxSpeed(int speed)
    {
        maxSpeed = speed;
    }
    public int getSpeed()
    {
        return maxSpeed;
    }
    public void setWeight(int heavy)
    {
        weight = heavy;
    }
    public int getWeight()
    {
        return weight;
    }
public void setMake(String carMake)
{
    make = carMake;
}
public String getMake()
{
    return make;
}
} // end Car

public class CarSample extends Applet
{
    Car fredsCar, marysCar;

    public void init()
    {
        setBackground(Color.white);
        fredsCar = new Car();
        fredsCar.setMaxSpeed(500);
        fredsCar.setWeight(1000);
        fredsCar.setMake("Honda");
        marysCar = new Car(600, 900, "Porsche"); // using constructor 2
    }

    public void paint(Graphics g)
    {
        int speed;
        int weight;
        String make;
        speed = fredsCar.getSpeed();
        weight = fredsCar.getWeight();
        make = fredsCar.getMake();
        g.drawString("Fred's car speed “ + speed + “ weight “ + weight + “ and make “ + make, 10, 20);
        g.drawString("Mary's car speed “ + marysCar.getSpeed() + “ weight “ + marysCar.getWeight() + “ and make “ + marysCar.getMake(), 10, 40);
    }
} // end CarSample

Creating your own Data Model

To create your own Data Model you create classes and objects from the class. In doing so you need to consider what data and methods you will require. For example to create a class to represent a die we may need:

Data such as
    face
the current value of the die.

Methods such as
    Die()
    Constructor method to create a new dice object with an initial random value
    throwDie()
    Imitates throwing a die and generates a new value for the die
    dieValue()
    Returns the current value of the die
    isEven()
    Returns a boolean value indicating if the die value is an even number.
The following program **CircleObject** uses a data model for a circle by creating a Circle class so Circle Objects can be created in the program and you can change their size and display them on the screen.

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

public class CircleObject extends Applet {
    Circle C1, C2;

    public void init() {
        C1 = new Circle(100, 100, 50);
        C2 = new Circle(200, 200, 30);
    }

    public void paint(Graphics g) {
        g.drawString("Showing circle outputs", 20, 20);
        C1.drawCircle(g);
        C2.drawCircle(g);
        C1.changeSize(25);
        C1.drawCircle(g);
    }
}

class Circle {
    private int radius;
    private int xCoord, yCoord;

    public Circle(int initialX, int initialY, int initialRadius) //constructor
    {
        radius = initialRadius;
        xCoord = initialX;
        yCoord = initialY;

        public void changeSize(int size) {
            radius = radius + size;
        }

        public void drawCircle(Graphics g) {
            g.drawOval(xCoord, yCoord, radius*2, radius*2);
        }
    }
}
```
**Computer Fundamentals and Computer Limitations**

**Number Systems**

A numbering system is a way of representing numbers. The **decimal (base 10)** system for counting was used because we have 10 digits (fingers). Computers use the **binary (base 2)** system because it is easy to produce electronic components which are in one of 2 states (on / off) rather than 10 states. **Hexadecimal** (base 16) is a simple compact way to represent a binary number.

**Converting from binary to decimal**

The easiest way to convert from binary to decimal (with small numbers) is to refer to the table below:

<table>
<thead>
<tr>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
<th>$2^{-3}$</th>
<th>$2^{-4}$</th>
<th>$2^{-5}$</th>
<th>$2^{-6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>.5</td>
<td>.25</td>
<td>.125</td>
<td>.0625</td>
<td>.03125</td>
<td>.015625</td>
</tr>
</tbody>
</table>

eg $101_2$ has the value $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5$

$1100_2$ has the value $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 = 12$

$0.101_2$ has the value $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = .5 + .125 = 0.625$

**Converting from decimal to binary**

Convert $211_{10}$ to binary.

<table>
<thead>
<tr>
<th>Dividends</th>
<th>Remainders</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>211</td>
</tr>
<tr>
<td>2</td>
<td>105</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Notes: Successive division continues until the quotient becomes zero. The number to the new base is then determined by taking the remainders from last to first.

Hence $211_{10} = 11010011_2$

**Converting the fractional parts from decimal to binary**

eg $1.25_{10}$ to binary

$0.25 \times 2 = 0 + 0.5$ \(\text{first digit} = 0\)

$0.5 \times 2 = 1 + 0$ \(\text{second digit} = 1\)

Giving $0.25_{10} = 0.01_2$

Notes: Only the fractional part is multiplied by 2. The process continues until enough digits have been obtained or the fractional part becomes zero.

eg $0.1_{10}$ to binary

$0.1 \times 2 = 0 + 0.2$ \(\text{first digit} = 0\)

$0.2 \times 2 = 0 + 0.4$ \(\text{second digit} = 0\)

$0.4 \times 2 = 0 + 0.8$ \(\text{third digit} = 0\)

$0.8 \times 2 = 1 + 0.6$ \(\text{fourth digit} = 1\)

$0.6 \times 2 = 1 + 0.2$ \(\text{fifth digit} = 1\)

$0.2 \times 2 = 0 + 0.4$ \(\text{sixth digit} = 0\)

Notes: An exact decimal fraction may not have an exact binary equivalent. For divisions which do not go exactly go to one more place than asked for, then round.

The decimal number 0.1 cannot be represented exactly in binary. It must be rounded off to a finite number of decimal places, giving rise to round-off error.

etc etc.

Writing as a recurring binary number:

$0.1_{10} = 0.0001100110011_{2}$

Notes:
Rounding of to, say, 6 binary points:

0.110 = 0.000110₂

**eg 3**  
5.125 to binary
5 becomes 101 in binary  
.125 × 2 = 0 + .25  *first digit = 0*
.25 × 2 = 0 + .5  *second digit = 0*
.5 × 2 = 1 + 0  *third digit = 1*

Thus 5.125₁₀ = 101.001₂

**Converting from binary to hexadecimal**

Because 16 is equal to 2⁴ each hexadecimal digit represents 4 binary digits. Thus to convert a binary number to hexadecimal the number is divided into groups of four binary digits. Each of these groups is then converted to its hexadecimal equivalent from the table on the right.

For example the binary number

01001111010101₂

is divided as

01 0011 1101 0101₂

and these groups are then converted to Hexadecimal

1 3 D 5₁₆

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

**Arithmetic in the Binary System**

**Addition**

Arithmetic in binary follows exactly the same rules as for decimal arithmetic.

(Check by decimal conversion)

\[
\begin{array}{c}
\text{eg} \\
1101 \\
+ 1110 \\
\hline
11011
\end{array}
\]

\[
\begin{array}{c}
13 \\
+ 14 \\
\hline
27
\end{array}
\]

**Note**

\[
\begin{array}{c}
1 + 0 = 1 \\
0 + 1 = 1 \\
1 + 1 = 0 + \text{carry of } 1 \\
1 + 1 + 1 = 1 + \text{carry of } 1
\end{array}
\]

**Representation of Unsigned Integers**

It wastes storage to represent numbers character by character, because one byte is used for each digit. So numbers are converted to binary and represented in that form.

One 8-bit byte can represent numbers ranging from \(0000 \ 0000₂ = 0₁₀\) to \(1111 \ 1111₂ = 255₁₀\)

This range is not generally large enough to show
- integers greater than 255
- negative numbers

The group of bits that the computer handles as a whole is referred to as a word (ie a 16-bit machine can handle 16 bits at a time).
Bit 15  Most significant bit  eg a 16-bit word  
Bits 1-14  Balance of the word  
Bit 0  Least significant bit  

The largest integer in a 16-bit word is \(2^{16} - 1\) (65535).

To show larger integers you need a longer word but there is always some fixed limit to integer size and the need for a larger integer would cause the system to break down.

If it is not possible to store the results of an operation the system 'overflows' the register provided, this is **arithmetic overflow**.

### Representation of Signed Integers

#### Twos Complement

The twos complement of a binary value is the complement of the value with respect to \(2^n\) where \(n\) is the number of bits in the word.

eg, Consider 1010, now \(n = 4\) therefore \(2^4 = 10000\) hence complement is 10000 - 1010 = 0110

**A simpler theory:**

A method to obtain twos complement

1. Take the one’s complement and add “1”.
2. A shortcut technique works as follows:
   - Start from the right of the number and work towards the left.
   - Any ‘0’ bits remain the same until the first ‘1’ bit and the first ‘1’ bit remains the same.
   - After the first ‘1’ bit all subsequent bits are reversed.

Examples (assuming an 8 bit twos complement representation):

<table>
<thead>
<tr>
<th>Original</th>
<th>Binary representation</th>
<th>2’s complement</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0000 0110</td>
<td>1111 1010</td>
<td>-6</td>
</tr>
<tr>
<td>-6</td>
<td>1111 1010</td>
<td>0000 0110</td>
<td>6</td>
</tr>
</tbody>
</table>

**Interpreting numbers in twos complement representation**

**Positive integers**

- The most significant bit is “0”.
- The value is equal to the binary string. For example, 0000 0011 represents \(3_{10}\).

**Negative integers**

- The most significant bit is “1”.
- To obtain the value, “take a twos complement”. For example 11110110 represents a negative value. Taking a twos complement gives 00001010 which represents the positive number \(6_{10}\). Hence 11110110 represents \(-6_{10}\).

**Arithmetic in twos complement representation**

Addition using the twos complement representation the same as addition of binary numbers except that the size of the number is limited by the number of bits in the representation. If the sum is too big it will cause an overflow error.

Subtraction is done by adding a negative value. That is to carry out the operation \(7 - 5\) the computer will calculate \(7 + (-5)\). The (-5) is represented by the twos complement of 5.
For example using an 8 bit word:

\[
\begin{array}{c}
7 & 00000111 \\
-5 & 00000101 \\
2 & 00000010
\end{array}
\]

becomes …

\[
\begin{array}{c}
00000111 \\
+ 11111011 \\
00000100
\end{array}
\]

= 2

### Representation of Floating Point Numbers (Used For Reals)

A floating point number may be represented by 3 parts in a single computer word. 

\[ m \times b^e \]

where \( m \) is the mantissa and \( 0 \leq \text{mantissa} < 1 \)

\( b \) is the base

\( e \) is the exponent (the power of the base as a positive or negative integer)

**eg** 67.79 \( \times 10^{-3} \)

**Normalised notation** has no significant digits before the decimal point, multiplied by the base (10) to the appropriate power. Floating point numbers are stored in normalised form as this is the most efficient way.

**eg** 0.6779 \( \times 10^{-1} \)

### Some examples of floating point representation

There are many possible ways of storing real numbers, and here are some examples.

**Example 1 – 16 bit word with 5 bit 2s complement exponent and normalised mantissa**

<table>
<thead>
<tr>
<th>15</th>
<th>bits 14 - 10</th>
<th>bits 9 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent in 2s</td>
<td>Normalised mantissa</td>
</tr>
<tr>
<td>0 (+)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (-)</td>
<td>complement to base 2</td>
<td></td>
</tr>
</tbody>
</table>

Largest number that can be represented: 0 01111 1111111111 → 0.1111111111111 \( \times 2^{111} \) \( (15) \)

Smallest positive number that can be represented: 0 10000 1000000000 → 0.1 \( \times 2^{-10000} \) \( (-16) \)

**eg1** 0 00001 1000000000

\[ + 1 \]

\[ .1 \]

\[ = \]

\[ + .12 \times 2^1 \]

\[ = + 0.5 \times 2^1 \]

\[ = 1 \]

\[ \text{sign} = \text{plus} \]

\[ \text{mantissa} = 0.1 \]

\[ \text{exponent} = +1 \]

**eg2** 1 00000 1101000000

\[ - 0 \]

\[ .1101 \]

\[ = - .1101 \times 2^0 \]

\[ = - 0.8125 \times 2^0 \]

\[ = - 0.8125 \]

**Example 2 – IEEE 754 Floating Point**

The internal binary format used for floating point and double numbers. The format assigns a meaning to every possible combination of bits. There are also representation for NaN (Not a Number) and plus and minus infinity.
### Floating Point Bit Representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Size in Bytes</th>
<th>Size in Bits</th>
<th>Accuracy</th>
<th>Range</th>
<th>Fields</th>
<th>Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>8 bytes</td>
<td>64 bits</td>
<td>14 to 15 significant digits</td>
<td>±4.94065645841246544e-324d to ±1.79769313486231570e+308d</td>
<td>is formed of 3 fields: 1-bit sign, 11-bit base 2 exponent, 52-bit fraction, lead 1 implied</td>
<td>a double can exactly represent integers in the range -2^{53} to +2^{53}</td>
</tr>
<tr>
<td>Float</td>
<td>4 bytes</td>
<td>32 bits</td>
<td>6 to 7 significant digits</td>
<td>±1.40129846432481707e-45 to ±3.40282346638528860e+38</td>
<td>is formed of 3 fields: 1-bit sign, 8-bit base 2 exponent, 23-bit fraction, lead 1 implied</td>
<td>a float can exactly represent integers in the range -2^{24} to +2^{24}</td>
</tr>
</tbody>
</table>

### Representation of Characters and Strings

Computer memory is arranged in **words**. Words are groups of **bytes** (i.e., 8 bits = 1 byte) or a number of memory locations each able to hold 1 bit (binary digit). Characters are stored in the computer by using a code for each character. There are a number of standard binary codes used in computers to represent characters. **ASCII** and **Unicode** are common.

**ASCII**

American Standard Code for Information Interchange is the code commonly used with PCs or microcomputers. Initially the code used 7 bits but changed to 8 bits to allow up to 256 characters. ASCII needs 8 bits to represent one character. In both cases an extra **parity bit** would be added. (see below)

**Unicode**

This is the character set **Java** uses. Unicode includes the standard ASCII character set as its first 256 characters (with the high byte set to 0), but also includes several thousand other characters representing most international alphabets. To represent the **primitive** java type **char**, Unicode uses the 16 bit code for alphanumerics. The numerical values are unsigned 16 bit values between 0 and 65535. Unicode allows 16 bits or 2 bytes for each character (in total can represent 65536 character combinations) and so takes up twice as much memory for data storage as ASCII. In order to find the character connected to a particular unicode we use in Java the following:

```java
char characterVariable = (char) integer_value
int k = 65;
char c = (char) k;
```

This will cause `c` to take the value ‘A’, since ‘A’ is the character with unicode and ASCII code equal to 65.

A table of the ASCII codes is on the next page.

### Representation of Boolean

A boolean variable has the two possible values **True** and **False**. This means a boolean value can be represented by one bit which has two values 0 and 1. In Java **1** represents **True** and **0** represents **False**. Although only one bit is required a particular implementation may use a byte to store the 1 or 0. In this case 8 bits would be used to store the boolean values.
<table>
<thead>
<tr>
<th>Dec Char</th>
<th>Binary</th>
<th>Dec Char</th>
<th>Binary</th>
<th>Dec Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  NUL</td>
<td>0000</td>
<td>58   :</td>
<td>0011</td>
<td>117  u</td>
<td>0111</td>
</tr>
<tr>
<td>1  SOH</td>
<td>0001</td>
<td>59   ;</td>
<td>0011</td>
<td>118  v</td>
<td>0111</td>
</tr>
<tr>
<td>2  STX</td>
<td>0010</td>
<td>60   &lt;</td>
<td>0011</td>
<td>119  w</td>
<td>0111</td>
</tr>
<tr>
<td>3  ETX</td>
<td>0011</td>
<td>61   =</td>
<td>0011</td>
<td>120  x</td>
<td>0111</td>
</tr>
<tr>
<td>4  EOT</td>
<td>0100</td>
<td>62   &gt;</td>
<td>0011</td>
<td>121  y</td>
<td>0111</td>
</tr>
<tr>
<td>5  ENQ</td>
<td>0101</td>
<td>63   ?</td>
<td>0011</td>
<td>122  z</td>
<td>0111</td>
</tr>
<tr>
<td>6  ACK</td>
<td>0110</td>
<td>64   @</td>
<td>0100</td>
<td>123  {</td>
<td>0111</td>
</tr>
<tr>
<td>7  BEL</td>
<td>0111</td>
<td>65   A</td>
<td>0100</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>8  BS</td>
<td>0100</td>
<td>66   B</td>
<td>0100</td>
<td>125  }</td>
<td>0111</td>
</tr>
<tr>
<td>9  HT</td>
<td>0101</td>
<td>67   C</td>
<td>0100</td>
<td>126  ~</td>
<td>0111</td>
</tr>
<tr>
<td>10  LF</td>
<td>0101</td>
<td>68   D</td>
<td>0100</td>
<td>127  DEL</td>
<td>0111</td>
</tr>
<tr>
<td>11  VT</td>
<td>0101</td>
<td>69   E</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12  FF</td>
<td>0110</td>
<td>70   F</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13  CR</td>
<td>0111</td>
<td>71   G</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14  SO</td>
<td>0110</td>
<td>72   H</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15  SI</td>
<td>0111</td>
<td>73   I</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16  DLE</td>
<td>0001</td>
<td>74   J</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17  DC1</td>
<td>0001</td>
<td>75   K</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18  DC2</td>
<td>0001</td>
<td>76   L</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19  DC3</td>
<td>0001</td>
<td>77   M</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20  DC4</td>
<td>0001</td>
<td>78   N</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21  NAK</td>
<td>0001</td>
<td>79   O</td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22  SYN</td>
<td>0001</td>
<td></td>
<td></td>
<td>90   Z</td>
<td>0101</td>
</tr>
<tr>
<td>23  ETB</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24  CAN</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25  EM</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26  SUB</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27  ESC</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28  FS</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29  GS</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30  RS</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31  US</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32  Space</td>
<td>0010</td>
<td></td>
<td></td>
<td>91   [</td>
<td>0101</td>
</tr>
<tr>
<td>33 !</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34 &quot;</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35 #</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36 $</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37 %</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38 &amp;</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39 '</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40 (</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41 )</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42 *</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43 +</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 ,</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45 -</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46 .</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47 /</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48 0</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49 1</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 2</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51 3</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52 4</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53 5</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54 6</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55 7</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56 8</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57 9</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Representation of Arrays**

Consider the following definition for a single dimensional array:

```java
int[] data = new int[3]; data[0] = 0; data[1] = 0; data[2] = 0;
```

This can be diagrammatically represented as follows:

![Single Dimensional Array Diagram]

Consider the following definition for a two dimensional array:

```java
int[][] data = new int[3][2];
```

This can be diagrammatically represented as follows (assuming array has been initialised to all zeroes):

![Two Dimensional Array Diagram]
Representation of Images and Sounds

**Data Compression** (Source Wikipedia)
In computer science and information theory, **data compression** or **source coding** is the process of encoding information using fewer bits (or other information-bearing units) than an unencoded representation would use through use of specific encoding schemes.

**Lossless** compression schemes are reversible so that the original data can be reconstructed, while **lossy** schemes accept some loss of data in order to achieve higher compression.

Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely without error. Lossless compression is possible because most real-world data has **statistical redundancy**. For example, in English text, the letter 'e' is much more common than the letter 'z', and the probability that the letter 'q' will be followed by the letter 'z' is very small.

Lossy data compression is possible if some loss of fidelity is acceptable. Generally, a lossy data compression will be guided by research on how people perceive the data in question. For example, the human eye is more sensitive to subtle variations in **luminance** than it is to variations in color. JPEG image compression works in part by "rounding off" some of this less-important information. Lossy data compression provides a way to obtain the best fidelity for a given amount of compression. In some cases, **transparent** (unnoticeable) compression is desired; in other cases, fidelity is sacrificed to reduce the amount of data as much as possible.

**Still Images**
One way to represent a picture is to consider the picture as a collection of dots called pixels (picture elements). In this representation the picture can be stored as a long string of bits representing rows of pixels in the picture. Each bit (0 or 1) corresponds to whether the pixel is black or white. Black and white images were economical because they only needed one bit per pixel.

If a picture has a resolution of 1000 x 5000 then there are 1000 pixels across and 5000 going down. e.g. a 1000 x 5000 image requires 625000 bytes (5,000,000 bits/8).

Colour pictures need a number of bits to represent each possible colour. Colour depth is the number of separate colours that a graphics file can handle.

Early images were then stored so only the best sixteen colours of an image were used. In this case 4 bits were needed to store information about each pixel. With the introduction of 256 colours images kept getting bigger.

**Audio**
Sampling is the process of turning analogue sound waves into digital (binary) signals. The system samples the sound by taking snapshots of its frequency and amplitude at regular intervals. The higher the sample rate the more accurate the digital sound to its real-life source and the larger the disk space required to record it. Original audio adapters provided 8-bit audio where 8 bits were used to digitise each sound sample. This meant that there were 256 possible digital values which is usually adequate for speech but not for music.

Basically sampling should happen at twice the highest frequency to be recorded. A sound sampled at 44 KHz requires as much as 10.5 Mb of disk space per minute.

Many audio adapters include their own data-compression capability. One such algorithm is Adaptive Differential Pulse Code Modulation which reduces file size by up to 50% but loses sound quality.
## Logic Circuits

The basic gates are as follows:

<table>
<thead>
<tr>
<th>Function / Gate</th>
<th>Symbol</th>
<th>Truth Table</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| **AND**         | ![AND symbol] | A B f | A \( \land \) B  
= 1 if both A and B are 1  
= 0 otherwise |
| **OR**          | ![OR symbol] | A B f | A \( \lor \) B  
= 0 if both A and B are 0  
= 1 otherwise |
| **NOT**         | ![NOT symbol] | A f | \( \neg A \)  
= 0 if A is 1  
= 1 if A is 0 |

From these basic gates the following gates can be built:

<table>
<thead>
<tr>
<th>Function / Gate</th>
<th>Symbol</th>
<th>Truth Table</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| **NAND** (NOT AND) | ![NAND symbol] | A B f | \( \neg (A \land B) \)  
= 0 if A both and B are 1  
= 1 otherwise |
| **NOR** (NOT OR) | ![NOR symbol] | A B f | \( \neg (A \lor B) \)  
= 1 if both A and B are 0  
= 0 otherwise |
| **XOR** (Exclusive OR) | ![XOR symbol] | A B f | A \( \oplus \) B  
= 1 when A or B is 1  
= 0 when A and B are the same (ie both 0 or both 1) |
## List of Logic Laws

### Commutative Laws
- **L1** \( a \land b = b \land a \)
- **L2** \( a \lor b = b \lor a \)

### Associative Laws
- **L4** \( a \land (b \land c) = (a \land b) \land c \)
- **L5** \( a \lor (b \lor c) = (a \lor b) \lor c \)

### Law of Negation
- **L6** \( \sim \sim a = a \)

### Distribution Laws
- **L7** \( a \land (b \land c) = (a \land b) \land (a \land c) \)
- **L8** \( a \lor (b \lor c) = (a \lor b) \lor (a \lor c) \)

### De Morgan's Laws
- **L10** \( \sim (a \lor b) = \sim a \land \sim b \)
- **L11** \( \sim (a \land b) = \sim a \lor \sim b \)

### Idempotent Laws
- **L12** \( a \land a = a \)
- **L13** \( a \lor a = a \)

### Law of the Excluded Middle
- **L14** \( a \lor \sim a = T \)

### Law of Contradiction
- **L15** \( a \land \sim a = F \)

### Laws of the Constants
- **L21** \( \sim T = F \)
- **L22** \( \sim F = T \)
- **L23** \( a \land T = a \)
- **L24** \( a \land F = F \)
- **L25** \( a \lor T = T \)
- **L26** \( a \lor F = a \)

### Complement Rules of \( \land \) and \( \lor \)
- **L27** \( a \lor (b \land \sim a) = a \lor b \)
- **L28** \( a \land (b \lor \sim a) = a \land b \)

### Absorption Laws
- **L33** \( a \land (a \lor b) = a \)
- **L34** \( a \lor (a \land b) = a \)
Karnaugh Maps

A Karnaugh map (K-map) is a graphical method of simplifying a Boolean logic expression. K-map can be used to simplify expressions involving 2, 3 or 4 Boolean variables.

Drawing K Maps

A K-maps is a pictorial arrangement of a truth tables and easier way to determine the minimum number of terms needed to express the function algebraically.

Minterms are formed by ANDing the variables in the table together. For example the truth table shown below contains 4 minterms. When expressed in a table, a function of n variables will have $2^n$ minterms.

Every function can be written as a sum of minterms.

To find the sum of midterms function:

1. pick the rows in the table where the output is 1
2. OR these minterms together.

![Truth table](image)

Adding the minterms in this way will always give you a function to describe the relationship between A and B. However it will not always give you the simplest function. Arranging the minterms into a K-map allows us to look for patterns we may not otherwise see and find a simpler function to describe the relationship.

![Equivalent K-map](image)
Simplifying Expressions with two variables using K-maps

When using K-maps to simplify an expression we look for the 1’s in the table and group them with a rectangle. Groupings can contain 1, 2 or 4 terms and must never contain a 0!

Example 1
Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f = (A \land \sim B) \lor (A \land B) \]

Equivalent K-map

\[
\begin{array}{cc}
A & B \\
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

Minterms adjacent in bottom row ⇒ sum of minterms can be simplified to A

\[ f = (A \land \sim B) \lor (A \land B) = A \]

Example 2
Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ f = (\sim A \land B) \lor (A \land \sim B) \]

Equivalent K-map

\[
\begin{array}{cc}
A & B \\
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Selected minterms are not adjacent in the same row or column ⇒ no simplification possible

\[ f = (\sim A \land B) \lor (A \land \sim B) \]

Example 3
Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f = (\sim A \land B) \lor (A \land \sim B) \lor (A \land B) \]

Equivalent K-map

\[
\begin{array}{cc}
A & B \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

In this case there are two groups of 1’s we can select.

Overlapping groups are allowed in K-map simplification.

Min terms sum \((A \land \sim B) \lor (A \land B)\) can be simplified to A. \(\Rightarrow f = (A \land \sim B) \lor (A \land B) = A \lor B\)

Min terms sum \((\sim A \land B) \lor (A \land B)\) can be simplified to B. \(\Rightarrow f = (\sim A \land B) \lor (A \land \sim B) \lor (A \land B) = A \lor B\)

Note: an OR is placed between the two simplified expressions to form the final simplified solution.
**Simplifying Expressions with three variables using K-maps**

Groupings can contain 1, 2, 4 or 8 terms and must never contain a 0!

**Example 4**

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sum of minterms = \((\sim A \land \sim B \land C) \lor (A \land \sim B \land C)\)

Selected minterms are both in the \sim B and the C column so the expression can be simplified to:

\[ f = \sim B \land C \]

**Example 5**

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Sum of minterms

\[ f = (\sim A \land \sim B \land C) \lor (A \land \sim B \land C) \lor (\sim A \land B \land C) \lor (A \land B \land C) \]

Selected minterms are all in both the C columns so the expression can be simplified to:

\[ f = C \]

**Example 6**

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum of minterms

\[ f = (\sim A \land \sim B \land \sim C) \lor (A \land \sim B \land \sim C) \lor (A \land B \land \sim C) \]

Selected minterms can be simplified to:

\[ f = (\sim B \land \sim C) \lor (A \land \sim C) \]

**Note:** There are a couple of ways to tackle example 6.

(i) Think of the left and right edges of our table as ‘glued’ together to form a cylinder.

![Glued Cylinder](image)

The minterms \((A \land \sim B \land \sim C)\) and \((A \land B \land \sim C)\) are now adjacent and can be summed as shown in the table above.

(ii) The table can be rearranged by sliding all columns to the right and forming the new table shown below.

The adjacent cells are now obvious.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Simplifying Expressions with four variables using K-maps**

Groupings can contain 1, 2, 4, 8 or 16 terms and must never contain a 0!

**Example 7**

**Truth table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Equivalent K-map**

Using the minterms selected:

\[ f = (\neg A \land \neg B \land \neg C \land \neg D) \lor (A \land D) \lor (B \land C \land \neg D) \]
Example 8

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The corners can be a little difficult to see how to group. Think about wrapping in two directions so that the corners come together in one block, almost like wrapping around a sphere.

Selected minterms can be simplified to:
\[ f = (\lnot B \land \lnot D) \lor (B \land \lnot C \land D) \]

OR the table can be rearranged by sliding all columns to the right and all rows down.

K-map

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Selected minterms can be simplified to:
\[ f = (\lnot B \land \lnot D) \lor (B \land \lnot C \land D) \]

Example 9

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Overlapping the groups gives the minimal number of groupings.

Selected minterms can be simplified to:
\[ f = (B \land D) \lor (C \land D) \]

Example 10

K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Selected minterms can be simplified to:
\[ f = D \]
**Example:** Design a logic circuit that will take a binary number in the range 0 - 3 and produce as output three times the input in unsigned binary representation.

Let the input bits be $i_1$ and $i_0$ and the output bits be $o_3$, $o_2$, $o_1$ and $o_0$

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_1$</td>
<td>$i_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The black box (machine) we need will have 2 inputs and 4 outputs.

As there are 4 outputs, each with their own Boolean expression, you need 4 expressions.

- $o_3 = i_0 \land i_1$
- $o_2 = \neg i_0 \land i_1$
- $o_1 = i_0 \land \neg i_1 \lor \neg i_0 \land i_1$
- $o_0 = i_0$

Thus the circuit would be:
THE TOY MACHINE

The TOY machine has 256 words of main memory, 16 registers (R[0] is permanently set to value 0), and 16-bit instructions. All programs start at memory location 10 (Hex). Memory address FF is connected to the input/output unit. There are 16 different instruction types; each one is designated by one of the opcodes 0 through F. Each instruction manipulates the contents of memory, registers, or the program counter in a completely specified manner. The 16 TOY instructions are organized into three categories: arithmetic-logic, transfer between memory and registers, and flow control.

Instruction Representation – main memory only

Summary of TOY instructions:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>DESCRIPTION</th>
<th>FORMAT</th>
<th>PSEUDOCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>-</td>
<td>exit</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>1</td>
<td>R[d] ← R[s] + R[t]</td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>1</td>
<td>R[d] ← R[s] - R[t]</td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>1</td>
<td>R[d] ← R[s] &amp; R[t]</td>
</tr>
<tr>
<td>4</td>
<td>xor</td>
<td>1</td>
<td>R[d] ← R[s] ^ R[t]</td>
</tr>
<tr>
<td>5</td>
<td>left shift</td>
<td>1</td>
<td>R[d] ← R[s] &lt;&lt; R[t]</td>
</tr>
<tr>
<td>6</td>
<td>right shift</td>
<td>1</td>
<td>R[d] ← R[s] &gt;&gt; R[t]</td>
</tr>
<tr>
<td>7</td>
<td>load address</td>
<td>2</td>
<td>R[d] ← addr</td>
</tr>
<tr>
<td>8</td>
<td>Load</td>
<td>2</td>
<td>R[d] ← mem[addr]</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>2</td>
<td>mem[addr] ← R[d]</td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>1</td>
<td>R[d] ← mem[R[t]]</td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>1</td>
<td>mem[R[t]] ← R[d]</td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>2</td>
<td>if (R[d] == 0) pc ← addr</td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>2</td>
<td>if (R[d] &gt; 0) pc ← addr</td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>-</td>
<td>pc ← R[d]</td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>2</td>
<td>R[d] ← pc; pc ← addr</td>
</tr>
</tbody>
</table>

Each TOY instruction consists of 4 hex digits (16 bits). The leading (left-most) hex digit encodes one of the 16 opcodes. The second (from the left) hex digit refers to one of the 16 registers, which we call the destination register and denote by \( d \). The interpretation of the two rightmost hex digits depends on the opcode. Each opcode has a unique format.

With Format 1 opcodes, the third and fourth hex digits are each interpreted as the index of a register, which we call the two source registers and denote by \( s \) and \( t \).

With Format 2 opcodes, the third and fourth hex digits (the rightmost 8 bits) are interpreted as a memory address, which we denote by \( \text{addr} \).
Note: two instructions have no format listed ie. 0 and E. Consider the following examples. Instructions 0000, 0A14, 0BC7 would all halt execution. The last 3 hex digits are ignored. Instructions E500, E514, E5C7 would all change the value of the program counter to the contents of register 5. The last 2 hex digits are ignored.

Toy Programming and Java

Variables – use memory locations 00 to 0F for variables and constants. Default data type is short integer.

Example 1: Assignment – x=3;  // x is stored in memory location 5

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0003</td>
<td>data</td>
<td>The integer 3 is in memory address 00</td>
</tr>
<tr>
<td>05</td>
<td>0000</td>
<td>data</td>
<td>Location chosen for variable x</td>
</tr>
<tr>
<td>10</td>
<td>8A00</td>
<td>R[A] ← mem[00]</td>
<td>The integer 3 is copied to register A</td>
</tr>
<tr>
<td>11</td>
<td>9A05</td>
<td>mem[05] ← R[A]</td>
<td>Contents of register A stored in x</td>
</tr>
<tr>
<td>12</td>
<td>0000</td>
<td>halt</td>
<td>Stop execution</td>
</tr>
</tbody>
</table>

Example 2: Addition – x=6+8;  // x is stored in memory location 5

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0006</td>
<td>data</td>
<td>The integer 6 is in memory address 00</td>
</tr>
<tr>
<td>01</td>
<td>0008</td>
<td>data</td>
<td>The integer 8 is in memory address 01</td>
</tr>
<tr>
<td>05</td>
<td>0000</td>
<td>data</td>
<td>Location chosen for variable x</td>
</tr>
<tr>
<td>10</td>
<td>8A00</td>
<td>R[A] ← mem[00]</td>
<td>The integer 6 is copied to register A</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>R[B] ← mem[01]</td>
<td>The integer 8 is copied to register B</td>
</tr>
<tr>
<td>12</td>
<td>1CAB</td>
<td>R[C] ← R[A] + R[B]</td>
<td>Integers 6 and 8 are added and the result (14) stored in register C</td>
</tr>
<tr>
<td>13</td>
<td>9C05</td>
<td>mem[05] ← R[C]</td>
<td>The integer 14 is copied to x</td>
</tr>
<tr>
<td>14</td>
<td>0000</td>
<td>halt</td>
<td>Stop execution</td>
</tr>
</tbody>
</table>

Multiplication can be performed by either

left shifts (each shift multiplies by 2), eg \( x = x \times 10 \); would be implemented as \( x = x \times 8 + x \times 2 \);

or by adding the first number to itself, eg \( x = x \times 10 \); would be implemented as \( x = x + x + x + x + x + x + x + x + x + x \);

(Note division can use right shifts or repeated subtraction)
Example 3: Multiplication – \( x = x \times 8 \); // \( x \) is stored in memory location 5

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0003</td>
<td>data</td>
<td>Number of places to left shift by</td>
</tr>
<tr>
<td>05</td>
<td>0004</td>
<td>data</td>
<td>Location chosen for variable ( x ) (with value 4 in it)</td>
</tr>
<tr>
<td>10</td>
<td>8A05</td>
<td>( R[A] \leftarrow \text{mem}[05] )</td>
<td>Load ( x ) (value 4) into register ( A )</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>( R[B] \leftarrow \text{mem}[01] )</td>
<td>Load number of places to shift left (3) in register ( B )</td>
</tr>
<tr>
<td>12</td>
<td>5AAB</td>
<td>( R[A] \leftarrow R[A] \ll R[B] )</td>
<td>Register ( A ) is shifted left 3 places (becomes 20*)</td>
</tr>
<tr>
<td>13</td>
<td>9A05</td>
<td>\text{mem}[05] \leftarrow R[A]</td>
<td>Contents of register ( A ) stored in ( x ) (20*)</td>
</tr>
<tr>
<td>14</td>
<td>0000</td>
<td>halt</td>
<td>Stop execution</td>
</tr>
</tbody>
</table>

* 20 is hexadecimal for 32.

The if statement

Requires jumping over blocks of memory used for the condition true and condition false cases.

<table>
<thead>
<tr>
<th>Java</th>
<th>TOY Processing Sequence</th>
<th>Example 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a==b) code1 else code2</td>
<td>code2 if (a-b=0) branch to code1 op op branch to cont</td>
<td>if (x==3) x=x+3; else x=x-1;</td>
</tr>
<tr>
<td></td>
<td>code1 op op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cont op op</td>
<td></td>
</tr>
</tbody>
</table>

Example 4: If statement

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>data</td>
<td>variable ( x )</td>
</tr>
<tr>
<td>01</td>
<td>0003</td>
<td>data</td>
<td>constant 3</td>
</tr>
<tr>
<td>02</td>
<td>0001</td>
<td>data</td>
<td>constant 1</td>
</tr>
<tr>
<td>10</td>
<td>8A00</td>
<td>( R[A] \leftarrow \text{mem}[00] )</td>
<td>load ( x ) into register ( A )</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>( R[B] \leftarrow \text{mem}[01] )</td>
<td>load constant 3 into register ( B )</td>
</tr>
<tr>
<td>12</td>
<td>2CAB</td>
<td>( R[C] \leftarrow R[A] - R[B] )</td>
<td>form test for ( x-3==0 )</td>
</tr>
<tr>
<td>13</td>
<td>CC18</td>
<td>if ( (R[C]==0) ) PC \leftarrow 18</td>
<td>if ( x==3 ) goto 18 (code1)</td>
</tr>
<tr>
<td>14</td>
<td>8B02</td>
<td>( R[B] \leftarrow \text{mem}[02] )</td>
<td>code2 : load constant 1 into register ( B )</td>
</tr>
<tr>
<td>15</td>
<td>2CAB</td>
<td>( R[C] \leftarrow R[A] - R[B] )</td>
<td>take 1 off ( x ) and store result in register ( C )</td>
</tr>
<tr>
<td>16</td>
<td>9C00</td>
<td>\text{mem}[00] \leftarrow R[C]</td>
<td>store register ( C ) in ( x ) (ie ( x=x-1 ))</td>
</tr>
<tr>
<td>17</td>
<td>C01B</td>
<td>if ( (R[0]==0) ) PC \leftarrow 1B</td>
<td>goto 1B (cont)</td>
</tr>
<tr>
<td>18</td>
<td>8B01</td>
<td>( R[B] \leftarrow \text{mem}[01] )</td>
<td>code1 : load constant 3 into register ( B )</td>
</tr>
<tr>
<td>19</td>
<td>1CAB</td>
<td>( R[C] \leftarrow R[A] + R[B] )</td>
<td>add 3 to ( x ) and store result in register ( C )</td>
</tr>
<tr>
<td>2A</td>
<td>9C00</td>
<td>\text{mem}[00] \leftarrow R[C]</td>
<td>store register ( C ) in ( x ) (ie ( x=x+3 ))</td>
</tr>
<tr>
<td>2B</td>
<td>0000</td>
<td>halt</td>
<td>cont – program continues here after if</td>
</tr>
</tbody>
</table>
While and for loops

Similar to if statement (to exit loop). For loops can be implemented as while loops

<table>
<thead>
<tr>
<th>Java while loop</th>
<th>Java for loop</th>
<th>Example 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0;</td>
<td>for (int x=0; x&lt;5; x++)</td>
<td>start x=0</td>
</tr>
<tr>
<td>while (x&lt;5)</td>
<td>{</td>
<td>here</td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>}</td>
<td>if (x - 4 &gt; 0) goto next</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>x = x + 1</td>
</tr>
</tbody>
</table>

Example 5: while / for loop

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Pseudocode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>data</td>
<td>constant 0 (loop first value)</td>
</tr>
<tr>
<td>01</td>
<td>0004</td>
<td>data</td>
<td>constant 4 (loop last value) (4 since the loop is &lt;)</td>
</tr>
<tr>
<td>02</td>
<td>0001</td>
<td>data</td>
<td>constant 1 (loop increment)</td>
</tr>
<tr>
<td>03</td>
<td>0000</td>
<td>data</td>
<td>variable</td>
</tr>
<tr>
<td>10</td>
<td>8A00</td>
<td>R[A] ← mem[00]</td>
<td>start - load initial value for x into register A</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>R[B] ← mem[01]</td>
<td>load constant 4 into register B</td>
</tr>
<tr>
<td>12</td>
<td>8D02</td>
<td>R[D] ← mem[02]</td>
<td>load constant 1 into register D</td>
</tr>
<tr>
<td>14</td>
<td>DC18</td>
<td>if (R[C]&gt;0) PC ← 18</td>
<td>goto 18 (next) if loop finished</td>
</tr>
<tr>
<td>15</td>
<td>1AAD</td>
<td>R[A] ← R[A] + R[D]</td>
<td>store x+1 in register A</td>
</tr>
<tr>
<td>16</td>
<td>9A03</td>
<td>mem[03] ← R[A] + R[D]</td>
<td>store value of x in memory</td>
</tr>
<tr>
<td>17</td>
<td>C013</td>
<td>if (R[0]==0) PC ← 13</td>
<td>goto 13</td>
</tr>
<tr>
<td>18</td>
<td>0000</td>
<td>halt</td>
<td>next : - program continues here after loop</td>
</tr>
</tbody>
</table>

Structure of the CPU

Control Unit

The control unit of the CPU contains circuitry that directs the entire computer system to carry out, or execute, stored program instructions. The control unit communicates with both the arithmetic/logic unit and main memory.

Register

Registers are temporary storage areas for instructions or data. They are not a part of memory. They are special additional storage locations that offer the advantage of speed. Registers work under the direction of the control unit to accept, hold, and transfer instructions or data.

ALU

Stands for Arithmetic/Logic Unit. This is the part that executes the computer's commands. A command must be either a basic arithmetic operation: + - * / or one of the logical comparisons: > < = not =.
Memory

Memory is the part of the computer that holds data and instructions for processing. Although closely associated with the central processing unit, memory is separate from it.

Bus

A bus is a collection of wires and connectors through which the data is transmitted. A bus is used to connect the components of the CPU and Memory. The bus has two parts -- an address bus and a data bus. The data bus transfers actual data whereas the address bus transfers information about the data and where it should go.

The Machine Cycle

<table>
<thead>
<tr>
<th>The Machine Cycle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></td>
<td>get an instruction from Memory and place in Instruction Register in Control Unit</td>
</tr>
<tr>
<td><strong>Decode</strong></td>
<td>converts instruction into computer commands that control the ALU and Memory</td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td>actually process the commands using the ALU and Memory</td>
</tr>
</tbody>
</table>
This question paper and any materials associated with this examination (including answer booklets, cover sheets, rough note paper, or information sheets) remain the property of the Office of Tasmanian Assessment, Standards and Certification.