COMPUTER SCIENCE
(ITC315118)

Information Booklet

Last updated: 13/7/18
TABLE OF CONTENTS

ALGORITHMS ......................................................................................................................... 3
The initially/when model ........................................................................................................ 3

JAVA ........................................................................................................................................ 4
Keywords ................................................................................................................................. 4
Comments ............................................................................................................................... 4
Variables and data types ......................................................................................................... 5
Arrays .................................................................................................................................... 7
Conditionals and loops .......................................................................................................... 8
Strings .................................................................................................................................... 9
Graphics methods .................................................................................................................. 11
Math methods ......................................................................................................................... 11
Event-driven programming .................................................................................................... 12
Methods ................................................................................................................................. 12
Scope of variables .................................................................................................................. 13
Objects and classes ................................................................................................................ 13

BOOLEAN LOGIC ..................................................................................................................... 16
Logic gates (Boolean operators) ............................................................................................. 16
Logic laws ............................................................................................................................... 16
Karnaugh maps ....................................................................................................................... 17

COMPUTER ARCHITECTURE .................................................................................................. 18
Structure of the Central Processing Unit (CPU) ..................................................................... 18
The TOY machine .................................................................................................................. 19
TOY programming and Java .................................................................................................. 20

DATA REPRESENTATION ...................................................................................................... 22
Representation of boolean ....................................................................................................... 22
Representation of numbers ...................................................................................................... 22
Representation of characters .................................................................................................. 26
Representation of arrays ......................................................................................................... 26
Representation of images ....................................................................................................... 27
Representation of sounds ....................................................................................................... 27
Data compression .................................................................................................................. 27
ASCII table ............................................................................................................................ 28
ALGORITHMS

An algorithm is a set of steps to accomplish a task. Algorithms can help us to break down a problem and evaluate the effectiveness of a particular approach.

The *initially/when* model

In an event-driven program, the order in which actions are taken is determined by responses from the user when interacting with the program’s GUI (Graphical User Interface). Event-driven programs should be described in terms of the initial state of the program and what happens when a particular condition occurs.

Algorithms for applets based on the *initially/when* model should work correctly no matter in what order the buttons or textfields are used.

**Example - Bus tickets fare calculator**

The following algorithm calculates the cost of a bus ticket based on the following specifications, assuming that valid data in entered. A possible screen for this applet is shown to the right.

Tickets can be purchased for 1, 2, 3 zones. The costs are:

- 1 zone => $2.80
- 2 zones => $3.50
- 3 zones => $4.20

Fares can be Full (i.e. full price) or Concession (50% discount).

```plaintext
Initially
zones = 1
fare_type = F
cost = 2.80
display cost

When a value is entered into the "zones" textfield
set zones to value in "zones" textfield

When a value is entered into the "fare_type" textfield
set fare_type to value in "fare_type" textfield

When the "calculate" button is pressed
switch zones
  case '1': cost = 2.80
  case '2': cost = 3.50
  case '3': cost = 4.20
  default: display error
if fare_type = 'C'
cost = cost * 0.5
display cost

When the "reset" button is pressed
zones = 1
fare_type = F
cost = 2.80
display cost
```

Buy your tickets here

Zones (1, 2 or 3): 
Fare type ('F' or 'C'): 
Calculate

Total cost: $2.80
Reset

Buy your tickets here

Zones (1, 2 or 3): 1
Fare type ('F' or 'C'): F
Calculate

Total cost: $2.80
Reset
JAVA

Java is a high-level 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.

Java applets are programs that can be embedded in a web page, whereas an application is a standalone Java program. Both require a Java virtual machine (JVM) to run.

Java was intended to be platform independent. Java programs are compiled into bytecodes that can run on any operating system that has a Java virtual machine (JVM) installed, without needing recompilation. This is in contrast to platform dependent programming languages (e.g. C, C++) in which the source code is directly compiled into machine code.

Whereas Java is platform independent, the JVM is necessarily platform dependant.

Applets typically contain the following standard methods:
- The init() method is called once when the applet is created and is typically used to set up the user interface and define the initial values of any variables.
- The paint() method is used to display the applet. It runs automatically after the init() method and whenever the applet window is refreshed. It takes a Graphics object as parameter that can be used to draw shapes and text in the applet window. To update the display, the repaint() method can be used to force the paint() method to be called again.

Example - HelloWorld Applet

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello World", 20, 20);
    }
}
```

Keywords

```
abstract  const  finally  int  public  throw
boolean  continue  float  interface  return  throws
break  default  for  long  short  transient
byte  do  goto  native  static  true
byvalue  double  if  new  super  try
case  else  implements  null  switch  void
catch  extends  import  package  synchronized  volatile
case  false  instanceof  private  this  while
class  final
```

Comments

Java supports two types of comments:

```
/* This is a multiline comment. */
Everything between /* and */ is ignored by the compiler.

// single line comment
Everything following // to the end of the line is ignored by the compiler.
```
Variables and data types

A variable is a location of memory that is associated with a name (or identifier) and contains some information (or value).

In Java, a variable holds a value that is either a primitive data type or a reference type.

Java has eight primitive data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>true or false</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>16-bit Unicode characters</td>
<td>Alphanumeric characters e.g. the characters on the keyboard</td>
<td>0 (&quot;\u0000&quot;)</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit two’s complement integer</td>
<td>-128 to 127</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>16-bit two’s complement integer</td>
<td>-32768 to 32767</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>32-bit two’s complement integer</td>
<td>-2,147,483,648 to +2,147,483,647</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>64-bit two’s complement integer</td>
<td>-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>32-bit single precision floating point number (using IEEE 754 standard)</td>
<td>+/- about 10^38</td>
<td>0.0</td>
</tr>
<tr>
<td>double</td>
<td>64-bit double precision floating point number (using IEEE 754 standard)</td>
<td>+/- about 10^317</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Reference types include arrays and class instances (i.e. objects). Whereas a primitive variable holds exactly one value, a reference variable holds a pointer (or reference) to a location in memory that many contain multiple values and/or methods.

Variable names in Java:
- can contain letters, numbers, underscore (_), or the dollar sign ($)
- may only begin with a letter, an underscore or a dollar sign (by convention, variable names begin with a lowercase letter)
- are case-sensitive
- cannot be keywords.

The keyword final can be used when declaring a variable to stop the value from being accidentally changed. By convention, the names of constants are typically written in uppercase.

final double PI = 3.14159;
double r = 20;
double areaOfCircle = PI * r * r;
Operators
This table shows the precedence for frequently-used Java operators. Operators higher up the table are evaluated before operators listed below them. Within an expression, operators with the same precedence are evaluated based on their associativity.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>member selection</td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>subscript</td>
<td>left-to-right</td>
</tr>
<tr>
<td>( )</td>
<td>method</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td></td>
</tr>
<tr>
<td>&lt;type&gt;</td>
<td>typecasting</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td>object creation</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left-to-right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulo (remainder)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition or string concat</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>is equal to (equality)</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to (inequality)</td>
<td></td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>logical AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>ternary (if-then-else)</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+=</td>
<td>addition assignment (x = x + y)</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>subtraction assignment</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>multiplication assignment</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>division assignment</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>modulus assignment</td>
<td></td>
</tr>
</tbody>
</table>

Promoting and casting
When expressions that contain different primitive data types are evaluated by the compiler, each value in the expression is automatically promoted (converted) to the larger type involved in the expression where possible. This is called a widening conversion and has no permanent effect on the variables in the expression. A widening conversion changes a value to a data type that will not lose information about the magnitude of the value (e.g. int to double).

Java’s widening conversions are:

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

Converting a larger primitive type to a smaller primitive type is called a narrowing conversion. This type of conversion may lose information, because it involves changing a value to a data type that might not be able to hold some of the possible values. To perform a narrowing conversion (e.g. double to int), a cast operator
is required. A cast operator consists of the name of the smaller type in brackets, placed directly before the value to be converted.

```java
double a = 2 * 2.4;       // a = 4.8
int b = (int)(2 * 2.4);   // b = 4
double c = 5 / 4;         // integer division, c = 1.0
double d = 5.0 / 4.0;     // floating-point division, d = 1.25
int e = 5 % 4;            // modulo, e = 1
```

Another use of casting is to convert a character to its ASCII/Unicode value or the reverse.

```java
char charValue = (char) 65; // charValue = 'A'
int numValue = (int) 'B';  // numValue = 66
```

Any type may be converted to type String by string conversion. If only one value in an expression is of type String, then string conversion is performed on the other operand to produce a string at run time.

```java
String s1 = "" + 42;       // s1 = "42"
String s2 = "" + 10 + 66;  // s2 = "1066"
String s2 = "" + (10 + 66); // s2 = "76"
```

**Arrays**

An *array* is an object that contains a fixed number of values of a single data type.

The *index* of the first element in an array is always 0. If the length of the array is *n*, then the index of the last element is (*n* - 1).

**Example – Array of integers**

```java
int[] a = {6, 4, 9, 1, 5};
sum = 0;
for (int i = 0; i < a.length; i++)
    sum = sum + a[i];
```

**Example – Two-dimensional array of integers**

A two-dimensional array is effectively an array of arrays.

```java
int[][] scores = {{93,74,77,55,81},
                 {78,77,72,75,80},
                 {92,88,82,83,69}};
int sum = 0;
for (int i=0; i < scores.length; i++)
    for (int j=0; j < scores[0].length; j++)
        sum = sum + scores[i][j];
```
Conditionals and loops

To write useful programs we almost always need ways to check conditions and change the behaviour of the program accordingly. We also frequently need to be able repeat sections of code.

The if-else statement

```java
if (hours > 40) {
    overtime = hours - 40; // if with single statement

if (hours > 40) // if...else with multiple statements
{
    overtime = hours - 40;
    normal = 40;
} else {
    overtime = 0;
    normal = hours;
}
```

The switch statement

- Only works with some primitive data types (byte, short, int, and char) and the classes that wrap them (Byte, Short, Integer, Character), enumerated types, and the String class.
- The break statement terminates the execution of a switch statement. Without it, statements after the matching case are executed.
- Can have a default case that handles all other values.

```java
switch (operation) {
    case '+': g.drawString("5+6 = " + (5+6), 10, 10); break;
    case '-': g.drawString("5-6 = " + (5-6),10,10);
               break;
    default:  g.drawString("wrong operator", 10,10); break;
}
```

The for statement

For statements define a loop that will run a predetermined number of times and describe the behaviour of the iterator in the loop header.

```java
for (int i=1; i<=10; i++)
    g.drawString("Computing is great", 20, 20*i);
```

The while statement

In a while statement, the test condition is evaluated before entering the loop. If the condition is false when the loop is first entered it will not be executed.

```java
while (condition) {
    statements;
}
```

The do-while statement

In a do-while statement, the test condition is evaluated after body of loop is executed so it is always performed at least once.

```java
do {
    statements;
} while (condition);
Strings
A string is a sequence of characters. Strings are objects, not one of the primitive data types.

Strings are immutable, which means that, once created, their values cannot be changed. String methods that appear to modify a string create and return a new string that contains the result of the operation.

Care needs to be taken when comparing strings. Because strings are objects, the comparison operator (==) only indicates whether two strings are the same object, not whether the character strings they contain are equal. To compare strings, instead use a String method, such as equals(), equalsIgnoreCase(), or compareTo().

Strings can be concatenated (i.e. joined together) using the + symbol.

Selected String methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>charAt(int index)</td>
<td>Gives the character at position number index in a string.</td>
</tr>
<tr>
<td>int</td>
<td>compareTo(String string2)</td>
<td>Returns 0 if the strings are alphabetically equal, a negative value if this string comes before string2, or a positive value otherwise.</td>
</tr>
<tr>
<td>boolean</td>
<td>endsWith(String suffix)</td>
<td>Tests if this string ends with the specified suffix.</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(String string2)</td>
<td>Compares this string to string2.</td>
</tr>
<tr>
<td>boolean</td>
<td>equalsIgnoreCase(String string2)</td>
<td>Compares this string to string2, ignoring case.</td>
</tr>
<tr>
<td>int</td>
<td>indexOf(String substring)</td>
<td>Returns the index within this string of the first occurrence of the specified substring. Returns -1 if the substring isn’t found.</td>
</tr>
<tr>
<td>int</td>
<td>indexOf(String substring, int fromIndex)</td>
<td>Returns the index within this string of the first occurrence of the specified character, starting the search at fromIndex. Returns -1 if the substring isn’t found.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty()</td>
<td>Returns true if, and only if, length() is 0.</td>
</tr>
<tr>
<td>int</td>
<td>length()</td>
<td>Returns the length of this string.</td>
</tr>
<tr>
<td>String</td>
<td>replace(char old, char change)</td>
<td>Returns a new string resulting from replacing all occurrences of old in this string with change.</td>
</tr>
<tr>
<td>boolean</td>
<td>startsWith(String prefix)</td>
<td>Tests if this string ends with the specified prefix.</td>
</tr>
<tr>
<td>String</td>
<td>substring(int beginIndex)</td>
<td>Returns a substring of this string, starting at beginIndex.</td>
</tr>
<tr>
<td>String</td>
<td>substring(int beginIndex, int endIndex)</td>
<td>Returns a substring of this string, starting at beginIndex, and finishing at endIndex-1.</td>
</tr>
<tr>
<td>char[]</td>
<td>toCharArray()</td>
<td>Converts this string to a new character array.</td>
</tr>
<tr>
<td>String</td>
<td>toUpperCase()</td>
<td>Returns a copy of this string converted to upper case.</td>
</tr>
<tr>
<td>String</td>
<td>toLowerCase()</td>
<td>Returns a copy of this string converted to lower case.</td>
</tr>
<tr>
<td>String</td>
<td>trim()</td>
<td>Returns a copy of this string with all leading and trailing whitespace removed.</td>
</tr>
</tbody>
</table>
String examples

String s1 = "Everybody! ";
String s;
char c;
int n;
boolean b;

n = s1.length; // n = 11
s = s1.toUpperCase(); // s = "EVERYBODY! 

s = s1.replace('e', 'a'); // s = "Evarybody! 

s = s1.trim(); // s = "Everybody!"

s = s1.substring(5); // s = "body! 

s = s1.substring(0,5); // s = "Every"

b = s1.startsWith("Every"); // b = true
b = s1.endsWith("body!"); // b = false
b = s1.trim().endsWith("body!"); // b = true

n = s1.indexOf("very"); // n = 1
n = s1.indexOf("very",4); // n = -1

String conversion

Objects of type String contain an array of char values. Strings can be converted to and from a character array.

char[] chars1 = {'o','b','j','e','c', 't'};
String s2 = new String(chars1); // s2 = "object"
String s3 = new String(chars1,1,4); // s3 = "bjec"
char[] chars2 = s3.toCharArray(); // chars2 = {'b','j','e','c'}

For each of the primitive data types (e.g. int, float, double) there is a corresponding wrapper class (e.g. Integer, Float, Double) that includes methods that may be used to convert between to and from strings.

int n1 = 123;
String s4 = "789";
double d1 = 83.5;
char c1 = 'A';
double d;

s = Integer.toString(n); // s = "123"
s = Double.toString(d1); // s = "83.5"
n = Integer.parseInt(s4); // n = 789
d = Double.valueOf(s4).doubleValue(); // d = 789.0
Graphics methods
The Graphics class is part of the java.awt package and includes the following methods. All of the following
Graphics methods return void.

Selected Graphics methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawPolygon(int[] xs, int[] ys, int n)</td>
<td>Draws a closed polygon defined by arrays of x and y coordinates.</td>
</tr>
<tr>
<td>drawLine(int x1, int y1, int x2, int y2)</td>
<td>Draws a line between the points (x1, y1) and (x2, y2).</td>
</tr>
<tr>
<td>drawOval(int x, int y, int w, int h)</td>
<td>Draws the outline of an oval.</td>
</tr>
<tr>
<td>drawRect(int x, int y, int w, int h)</td>
<td>Draws the outline of the specified rectangle.</td>
</tr>
<tr>
<td>drawString(String str, int x, int y)</td>
<td>Draws the text given by the specified string, using the current font and colour.</td>
</tr>
<tr>
<td>fillOval(int x, int y, int w, int h)</td>
<td>Fills an oval bounded by the specified rectangle with the current colour.</td>
</tr>
<tr>
<td>fillPolygon(int[] xs, int[] ys, int n)</td>
<td>Fills a closed polygon defined by arrays of x and y coordinates.</td>
</tr>
<tr>
<td>fillRect(int x, int y, int w, int h)</td>
<td>Fills the specified rectangle.</td>
</tr>
<tr>
<td>setColor(Color c)</td>
<td>Sets the current colour.</td>
</tr>
<tr>
<td>setFont(Font font)</td>
<td>Sets the current font.</td>
</tr>
</tbody>
</table>

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

Selected Math methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>abs(int a)</td>
<td>Returns the absolute value of the argument.</td>
</tr>
<tr>
<td>double</td>
<td>abs(double a)</td>
<td>Returns the smallest (closest to negative infinity) integer that is greater than or equal to the argument.</td>
</tr>
<tr>
<td>double</td>
<td>ceil(double a)</td>
<td>Returns the largest (closest to positive infinity) integer that is less than or equal to the argument.</td>
</tr>
<tr>
<td>double</td>
<td>floor(double a)</td>
<td>Returns the greatest of the two arguments.</td>
</tr>
<tr>
<td>int</td>
<td>max(int a, int b)</td>
<td>Returns the least of the two arguments.</td>
</tr>
<tr>
<td>double</td>
<td>max(double a, double b)</td>
<td>Returns the value of the first argument raised to the power of the second argument.</td>
</tr>
<tr>
<td>double</td>
<td>min(double a, double b)</td>
<td>Generates a random number between 0.0 and 1.0.</td>
</tr>
<tr>
<td>int</td>
<td>round(double a)</td>
<td>Returns the closest integer, with ties being rounded up.</td>
</tr>
<tr>
<td>double</td>
<td>sqrt(double a)</td>
<td>Returns the square root of the argument.</td>
</tr>
</tbody>
</table>

Examples

```java
int a = (int)(Math.random()*4+2); // a is set to one of 2, 3, 4, or 5
int b = Math.abs(-5); // b = 5
int c = Math.abs(5); // c = 5
double d = Math.pow(2,3); // d = 8.0
double e = Math.max(-4,3); // e = 3.0
double f = Math.floor(5.9); // f = 5.0
double g = Math.ceil(5.9); // g = 6.0
```
Event-driven programming

In event-driven programming, the flow of the program is determined by events such as user actions, including keyboard presses and mouse movements. Detecting an event and handling it is called event handling.

A component is a Graphical User Interface (GUI) object that enables the user to interact with the applet. There are many such components, including labels, text fields, and buttons. By default, components are placed in the applet window in the order they are added, from left to right, top to bottom.

Example – Button Count

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

public class ButtonCount extends Applet implements ActionListener
{
    Button enterButton;
    int count;

    public void init()
    {
        enterButton = new Button("Enter");
        add(enterButton);
        enterButton.addActionListener(this);
    }

    public void paint(Graphics g)
    {
        g.drawString("Number of button presses is "+ count, 20, 120);
    }

    public void actionPerformed(ActionEvent event)
    {
        count = count + 1;
        repaint();
    }
}
```

Methods

A method is a section of code that is referred to by name and can be called from elsewhere in a program. They enable code to be written once and used many times, including in other classes (e.g. Math.random(), g.drawString(), etc.).

Methods can take multiple arguments and optionally return a value.

How parameters are passed

A parameter is a variable in a method definition. An argument is the actual value of the variable that gets passed to a method. When variables are passed to a method, they are passed by value regardless of the original data type. This means that for each parameter, a copy is made. Consequently, whatever the method does to the argument, it has no effect on the original variable.

For example, in the following code, the method change() will return 11 but the original value of x = 10 is unchanged.

```java
int x = 10;
int y = change(x);

public int change(int z)
{
    z++;
    return z;
}
```
Passing reference types as parameters

Passing a reference type (i.e. an array or other object) to a method is subtly, but significantly, different from passing a primitive data type. Although Java manipulates objects by reference, it still passes object references to methods by value, just like any other parameter. When an object is passed to a method, a copy of the reference is made, but not the object that the reference points to.

For example, in the following code, the `fillZero()` method will change the values in the array `table` to 0.

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

public void fillZero(int[] array) {
    for (int s=0; s<array.length; s++)
        array[s]=0;
}
```

Because strings are objects, when a string is passed to a method, the reference to the string is passed by value (a copy of the reference is passed). But because strings are immutable, changing the value of a string creates a new string that the copy of the reference now points to. In practice, we can think of strings as behaving like a primitive data type being passed by value.

Scope of variables

The scope of a variable is the part of the program where the variable is accessible.

- **Global variables** are declared just inside the class definition, but outside of any methods, and may be used within any method defined in the class.
- **Local variables** are declared within a method, constructor, or block (e.g. a loop), and can only be accessed within the part of the program where they have been declared. Method parameters can be considered local variables.

When a local variable has the same name as a global variable, the global variable is hidden within the scope of the local variable.

Objects and classes

*Object-oriented programming* is an approach to programming based on *objects* rather than functions.

- An *object* is a data structure that includes values (i.e. *variables*) and ways to manipulate data (i.e. *methods*).
- A *class* is a template for creating an object, including the initial values of any *variables* as well as *method* definitions.
- An object is an *instance* of a class.
- Any number of objects can be created (i.e. instantiated) from a class.

Important principles of object-oriented programming include encapsulation, inheritance, and polymorphism.

- **Encapsulation** – Encapsulation is the process of grouping related data and methods together in a class and makes *data hiding* possible. Variables (and methods) can be kept *private* with public getter and setter methods providing safe access to an object's data.
- **Inheritance** – A class can be derived from another class, inheriting its variables and methods. The class that inherits the properties of another is known as a *child class* (or subclass) and the class that is inherited from is known as a *parent class* (or superclass). For example, when a Java applet is created, it is a child of the parent Applet class.
- **Polymorphism** – Polymorphism is the ability of an object to take on many forms. Two examples of polymorphism in Java are *method overloading* (i.e. a class may have more than one method with the same name, as long as their argument lists are different) and *method overriding* (i.e. declaring a method in a child class that is also declared in a parent class).
Class definitions and constructors
A class has the following general structure:

<class modifiers> class <name of class>
{
    <variables declarations>
    <constructor declarations>
    <method declarations>
}

A constructor is a block of code similar to a method that is called when an object is instantiated (created).

- Constructors must have the same name as the class.
- Every class definition requires at least one constructor.
- Constructors never return a value (other than the object they have created), so they don’t specify a return type (not even void).
- It is recommended to have a constructor that sets the instance variables to default values.
- Multiple constructor methods can be defined, but they still must all have the same name as the class and differ in the number and type of the parameters. This is known as constructor overloading.

Object creation
To use an object, you must first:

- Declare the object by stating its class and giving it a name. By convention, class names begin with a capital letter, and object names begin with lower case.
- Instantiate (and initialise) the object by calling one of the constructors for the object with the new operator.

For example, to create an Enter button:

```java
Button myButton;
myButton = new Button("Enter");
```

Alternatively, an object can be declared and instantiated in one line:

```java
Button myButton = new Button("Enter");
```
Example – Car class definition

class Car
{
    private int maxSpeed;
    private String make;

    public Car(){ // constructor 1
        maxSpeed = 0;
        make = "";
    }

    public Car(int speed, String carMake){ // constructor 2
        maxSpeed = speed;
        make = carMake;
    }

    public void setMaxSpeed(int speed) {
        maxSpeed = speed;
    }

    public int getMaxSpeed(){
        return maxSpeed;
    }

    public void setMake(String carMake) {
        make = carMake;
    }

    public String getMake(){
        return make;
    }
}

Example – Creation and use of Car objects

public class CarSample extends Applet
{
    Car fredsCar, marysCar;

    public void init()
    {
        fredsCar = new Car(); // using constructor 1
        fredsCar.setMaxSpeed(500);
        fredsCar.setMake("Honda");
        marysCar = new Car(600, "Porsche"); // using constructor 2
    }

    public void paint(Graphics g)
    {
        g.drawString("Fred’s car speed " + fredsCar.getSpeed() +
            " and make " + fredsCar.getMake(), 10, 20);
        g.drawString("Mary’s car speed " + marysCar.getSpeed() +
            " and make " + marysCar.getMake(), 10, 40);
    }
}
BOOLEAN LOGIC

Boolean logic (or Boolean algebra) is a branch of Mathematics where variables have two possible values, true or false, often denoted as 1 and 0 respectively.

Logic gates (Boolean operators)

<table>
<thead>
<tr>
<th>Gate</th>
<th>Symbol</th>
<th>Logic circuit</th>
<th>Truth table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$\land$</td>
<td><img src="image" alt="Logic circuit diagram" /></td>
<td>$\begin{array}{c</td>
<td>c</td>
</tr>
<tr>
<td>OR</td>
<td>$\lor$</td>
<td><img src="image" alt="Logic circuit diagram" /></td>
<td>$\begin{array}{c</td>
<td>c</td>
</tr>
<tr>
<td>NOT</td>
<td>$\neg$</td>
<td><img src="image" alt="Logic circuit diagram" /></td>
<td>$\begin{array}{c</td>
<td>c} A &amp; \neg A \ \hline 0 &amp; 1 \ 1 &amp; 0 \end{array}$</td>
</tr>
</tbody>
</table>

Logic laws

*Commutative laws*

L1 $a \land b \equiv b \land a$
L2 $a \lor b \equiv b \lor a$

*Associative laws*

L4 $a \land (b \land c) \equiv (a \land b) \land c$
L5 $a \lor (b \lor c) \equiv (a \lor b) \lor c$

*Law of negation*

L6 $\neg \neg a \equiv a$

*Distribution laws*

L7 $a \lor (b \land c) \equiv (a \lor b) \land (a \lor c)$
L8 $a \land (b \lor c) \equiv (a \land b) \lor (a \land c)$

*De Morgan’s laws*

L10 $\neg(a \lor b) \equiv \neg a \land \neg b$
L11 $\neg(a \land b) \equiv \neg a \lor \neg b$

*Idempotent laws*

L12 $a \land a \equiv a$
L13 $a \lor a \equiv a$

*Law of the excluded middle*

L14 $a \lor \neg a \equiv T$

*Law of contradiction*

L15 $a \land \neg a \equiv F$

*Laws of the constants*

L21 $\neg T \equiv F$
L22 $\neg F \equiv T$
L23 $a \land T \equiv a$
L24 $a \land F \equiv F$
L25 $a \lor T \equiv T$
L26 $a \lor F \equiv a$

*Complement rules of $\land$ and $\lor$*

L27 $a \lor (b \land \neg a) \equiv a \lor b$
L28 $a \land (b \lor \neg a) \equiv a \land b$

*Absorption laws*

L33 $a \land (a \lor b) \equiv a$
L34 $a \lor (a \land b) \equiv a$
**Karnaugh maps**

A *Karnaugh map* is a visual method of simplifying a Boolean logic expression.

**Example 1 – Two variables**

Use a Karnaugh map to simplify the expression \( E \equiv (A \land \neg B) \lor (A \land B) \)

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>E</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>
```

**Example 2 – Three variables**

This is a truth table for a logic device that has inputs A, B, and C, and a single output F. Use a Karnaugh map to determine a simple logic expression for F.

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

**Example 3 – Four variables**

This is a truth table for a logic device that has inputs A, B, and C, D, and a single output G. Use a Karnaugh map to determine a simple logic expression for G.

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

\( F \equiv (\neg B \land \neg C) \lor (A \land \neg C) \)

\( G \equiv (\neg A \land \neg B \land \neg C \land \neg D) \lor (A \land D) \lor (B \land C \land \neg D) \)
COMPUTER ARCHITECTURE

Computer architecture is a collection of rules and methods that describes how a set of hardware components and software technology interacts to form a computer system.

Structure of the Central Processing Unit (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 (ALU) 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 including arithmetic operations and logical comparisons.

Memory  Memory is the part of the computer that holds data and instructions. 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 data whereas the address bus transfers information about the data and where it should go.

THE MACHINE CYCLE

| Fetch   | Get an instruction from memory and place in the instruction register within the control unit. |
| Decode  | Convert the instruction into computer commands that control the ALU and memory |
| Execute | Process the commands using the ALU and memory |

Diagram:

- CPU
- Control Unit
- ALU
- Decode
- Fetch
- Execute
- Memory
The TOY machine

TOY is an imaginary machine that models the structure of an early modern computer. The TOY machine has 16 registers (R[0] to R[F]), 256 words of main memory (00₁₆ to FF₁₆), and 16 different instruction types (given by the opcodes 0 to F) and processes 16-bit words.

- All programs start at memory location 10.
- Memory address FF is connected to the input/output (e.g. keyboard/display).
- Register 0 (R[0]) is permanently set to value 0.

<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 (or 16 bits). The leading (left-most) hex digit encodes one of the 16 opcodes. The second 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. With Format 1 opcodes, the third and fourth hex digits are each interpreted as the index of a register. With Format 2 opcodes, the third and fourth hex digits (the rightmost 8 bits) are interpreted as a memory address.

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format 1</td>
<td>opcode</td>
<td>destination (d)</td>
<td>source (s)</td>
<td>source (t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format 2</td>
<td>opcode</td>
<td>destination (d)</td>
<td>address (addr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Instructions 0 and E have no format listed. Consider the following examples. Instructions 0000, 0A14, 0BC7 would all halt execution. The last three 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 two hex digits are ignored.
## TOY programming and Java

### Example – Addition

Write TOY code equivalent to the following Java code.

```java
int x = 6 + 8;
```

To perform addition, we load the values to be added into registers and use opcode 1.

<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>(0000 0000 0000 0110, 6₁₀)</td>
</tr>
<tr>
<td>01</td>
<td>0008</td>
<td>data</td>
<td>(0000 0000 0000 1000, 8₁₀)</td>
</tr>
<tr>
<td>05</td>
<td>0000</td>
<td></td>
<td>location chosen for variable x</td>
</tr>
<tr>
<td>10</td>
<td>8A00</td>
<td>R[A] ← mem[00]</td>
<td>load 6 into register A</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>R[B] ← mem[01]</td>
<td>load 8 into register B</td>
</tr>
<tr>
<td>12</td>
<td>1CAB</td>
<td>R[C] ← R[A] + R[B]</td>
<td>add 6 and 8 and store result (14) in register C</td>
</tr>
<tr>
<td>13</td>
<td>9C05</td>
<td>mem[05] ← R[C]</td>
<td>copy result (14) to variable x</td>
</tr>
<tr>
<td>14</td>
<td>0000</td>
<td>halt</td>
<td>stop execution</td>
</tr>
</tbody>
</table>

### Example – Multiplication

Write TOY code equivalent to the following Java code.

```java
int x = 3 * 8;
```

Multiplication can be performed by either:

- left shifts (each shift multiplies by 2), e.g. \( x = x \times 10 \) could be implemented as \( x = x \times 8 + x \times 2 \)
- repeated addition, e.g. \( x = x \times 10 \) could be implemented by adding \( x \) to itself 10 times

Similarly, division can be performed using right shifts or repeated subtraction.

<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>(0000 0000 0000 0011, 3₁₀)</td>
</tr>
<tr>
<td>02</td>
<td>0008</td>
<td>data</td>
<td>(0000 0000 0000 1000, 8₁₀)</td>
</tr>
<tr>
<td>03</td>
<td>0000</td>
<td>data</td>
<td>(0000 0000 0000 0000, 0₁₀)</td>
</tr>
<tr>
<td>04</td>
<td>0001</td>
<td>data</td>
<td>(0000 0000 0000 0001, 1₁₀)</td>
</tr>
<tr>
<td>05</td>
<td>0000</td>
<td></td>
<td>location chosen for variable x</td>
</tr>
<tr>
<td>10</td>
<td>8A01</td>
<td>R[A] ← mem[01]</td>
<td>load 3 into register A</td>
</tr>
<tr>
<td>11</td>
<td>8B02</td>
<td>R[B] ← mem[02]</td>
<td>load 8 into register B</td>
</tr>
<tr>
<td>12</td>
<td>8C03</td>
<td>R[C] ← mem[03]</td>
<td>load 0 into register C</td>
</tr>
<tr>
<td>13</td>
<td>8104</td>
<td>R[1] ← mem[04]</td>
<td>load 1 into register 1</td>
</tr>
<tr>
<td>14</td>
<td>CA18</td>
<td>if (R[A] == 0) pc ← 18</td>
<td>if R[A] == 0 exit the loop</td>
</tr>
<tr>
<td>17</td>
<td>C014</td>
<td>pc ← 14</td>
<td>go to 14</td>
</tr>
<tr>
<td>18</td>
<td>9C05</td>
<td>mem[05] ← R[C]</td>
<td>copy result (24) to variable x</td>
</tr>
<tr>
<td>19</td>
<td>0000</td>
<td>halt</td>
<td>stop execution</td>
</tr>
</tbody>
</table>
Example – If-else

To implement an if-else statement in TOY, use opcode C (branch zero) or opcode D (branch positive) and jump over the blocks of code that don’t apply.

<table>
<thead>
<tr>
<th>Java example</th>
<th>Pseudocode (Java version)</th>
<th>Pseudocode (TOY version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x==3) x=x+3; else x=x-1;</td>
<td>if (a==b) &lt;code 1&gt; if (a-b=0) branch to code1 &lt;code 2&gt;</td>
<td>branch to next &lt;code 1&gt;</td>
</tr>
</tbody>
</table>

<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] ← mem[00]</td>
<td>load x into register A</td>
</tr>
<tr>
<td>11</td>
<td>8B01</td>
<td>R[B] ← mem[01]</td>
<td>load constant 3 into register B</td>
</tr>
<tr>
<td>12</td>
<td>2CAB</td>
<td>R[C] ← 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 ← 18</td>
<td>if x==3 go to 18 (code 1)</td>
</tr>
<tr>
<td>14</td>
<td>8B02</td>
<td>R[B] ← mem[02]</td>
<td>code 2: load constant 1 into register B</td>
</tr>
<tr>
<td>15</td>
<td>2CAB</td>
<td>R[C] ← 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>mem[00] ← R[C]</td>
<td>store register C in x (i.e. x=x-1)</td>
</tr>
<tr>
<td>17</td>
<td>C01B</td>
<td>if (R[0]==0) pc ← 1B</td>
<td>go to 1B (cont)</td>
</tr>
<tr>
<td>18</td>
<td>8B01</td>
<td>R[B] ← mem[01]</td>
<td>code 1: load constant 3 into register B</td>
</tr>
<tr>
<td>19</td>
<td>1CAB</td>
<td>R[C] ← R[A] + R[B]</td>
<td>add 3 to x and store result in register C</td>
</tr>
<tr>
<td>1A</td>
<td>9C00</td>
<td>mem[00] ← R[C]</td>
<td>store register C in x (i.e. x=x+3)</td>
</tr>
<tr>
<td>1B</td>
<td>0000</td>
<td>halt</td>
<td>stop execution (or continue program from here)</td>
</tr>
</tbody>
</table>

Example – While / for loop

Remember that for loops can be implemented as while loops.

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

<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) (since condition is &lt;5)</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>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>go to 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]</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>go to 13</td>
</tr>
<tr>
<td>18</td>
<td>0000</td>
<td>halt</td>
<td>stop execution (or continue program from here)</td>
</tr>
</tbody>
</table>
DATA REPRESENTATION

Data representation deals with how data is represented and structured symbolically for storage and communication in a computer system.

Representation of boolean
A boolean variable has the two possible values, true and false. This means a boolean value can be represented by a single bit (binary digit), which has two possible values, 0 (false) and 1 (true).

Representation of numbers
Computers store numbers (and indeed all data) using a binary representation. Whereas the decimal (base 10) number system uses 10 digits (0,1,2,3,4,5,6,7,8,9), binary (base 2) is a system for representing numbers using two digits (0, 1). Any number can be represented as a sequence of bits.

Unsigned binary
In unsigned binary, each bit represents an increasing power of 2 as shown in this table:

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<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>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
</tr>
</tbody>
</table>

- A sequence of bits is referred to as a word.
- A byte is an 8-bit word and a nibble is a 4-bit word.
- The left-most bit of a word is known as the most significant bit and the right-most bit is the least significant bit.
- One byte (ie. an 8-bit word) can represent numbers ranging from 0000 0000$_2$ = 0$_{10}$ to 1111 1111$_2$ = 255$_{10}$
- The largest integer in a $n$-bit unsigned representation is $2^n - 1$ (e.g. the largest number in a 16-bit unsigned representation is $2^{16} - 1 = 65535$).

Example – Binary to decimal conversion

\[
egin{align*}
101_2 & \Rightarrow 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5_{10} \\
1100_2 & \Rightarrow 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 = 12_{10} \\
0.101_2 & \Rightarrow 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625_{10}
\end{align*}
\]

Example – Decimal to binary conversion

To convert a decimal integer to binary, we can repeatedly divide by 2, recording the remainders as shown:

e.g. convert 211$_{10}$ to binary.

<table>
<thead>
<tr>
<th>Dividends</th>
<th>Remainders</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2105</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>252</td>
<td>1</td>
<td>Successive division continues until the quotient becomes zero.</td>
</tr>
<tr>
<td>226</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>0</td>
<td>The binary number is determined by reading the remainders from last to first.</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>Hence 211$_{10}$ = 1101 0011$_2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Example – Decimal fraction to binary conversion

To convert a decimal fraction to binary, we can repeatedly multiply by 2, recording the whole number (i.e. 0 or 1) and fraction parts separately as shown:

Convert 5.2\text{10} to binary.

\[
5_{10} = 101_{2}
\]

\[
0.2 \times 2 = 0 + 0.4 \quad \text{first digit} = 0
\]

\[
0.4 \times 2 = 0 + 0.8 \quad \text{second digit} = 0
\]

\[
0.8 \times 2 = 1 + 0.6 \quad \text{third digit} = 1
\]

\[
0.6 \times 2 = 1 + 0.2 \quad \text{fourth digit} = 1
\]

\[
0.2 \times 2 = 0 + 0.4 \quad \text{fifth digit} = 0
\]

\[
0.4 \times 2 = 0 + 0.8 \quad \text{sixth digit} = 0
\]

\text{etc.}

Writing as a recurring binary number:

\[
0.2_{10} = 0.0011001100110011..._{2}
\]

Rounding off to, say, 5 binary places:

\[
0.2_{10} = 0.00110_{2}
\]

Therefore, 5.2\text{10} = 101.00110\text{2} (to 5 binary places)

Notes

Convert the whole number part separately.

Starting with the fraction part of the original number, repeatedly multiply by 2.

The process continues until enough digits have been obtained or the fractional part becomes zero.

An exact decimal fraction may not have an exact binary equivalent. In that case, go to one more place than asked for, then round.

For example, the decimal number 0.2 cannot be represented exactly in binary. It must be rounded off to a finite number of binary places, giving rise to round-off error.

Two's complement

Two’s complement is a common method of representing signed numbers. The two’s complement of a binary number is its complement with respect to 2\text{n} where \text{n} is the number of bits in the word. In practice, the two’s complement of a number is its inverse.

Example – Taking the two’s complement (Method 1)

1. Take the one’s complement by inverting all the bits.
2. Add 1.

Example – Taking the two’s complement (Method 2)

1. Work from right to left, starting with the least significant bit.
2. Copy any 0 bits until the first 1 bit and copy the first 1 bit.
3. Invert all subsequent bits.

Example – Converting a two’s complement binary number to decimal

To interpret a number in two's complement representation:

1. If the most significant bit is 0, the number is positive (or zero).
   \[
   \rightarrow \text{Convert the number to decimal normally.}
   \]

2. If the most significant bit is 1, the number is negative.
   \[
   \rightarrow \text{Take the two's complement to find the number's inverse.}
   \]
   \[
   \text{(Note: Because the original number was negative, the inverse must be a positive number.)}
   \]
   \[
   \rightarrow \text{Convert the (positive) binary number to decimal normally.}
   \]
   \[
   \rightarrow \text{Invert the decimal number to determine the value of the original (negative) binary number.}
   \]
Convert the following 8-bit two's complement numbers to decimal:

a) 0000 0011
The most significant bit is 0, so it is a positive number.
Therefore, 0000 0011<sub>2</sub> = 3<sub>10</sub>

b) 1111 0001
The most significant bit is 1, so it is a negative number. Take the two's complement before converting to decimal.

\[
\begin{align*}
1111 \ 0001 \\
\rightarrow 0000 \ 1111 \\
\rightarrow 15
\end{align*}
\]
Therefore, 1111 0001<sub>2</sub> = -15<sub>10</sub>

**Binary arithmetic**

Arithmetic in binary is very similar to decimal arithmetic.

\[
\begin{array}{c}
1101 \\
+ 1110 \\
\hline
11011
\end{array}
\quad \begin{array}{c}
13 \\
+ 14 \\
\hline
27
\end{array}
\]

An overflow occurs when there aren't enough bits to represent the result of an operation.

An advantage of two's complement representation is that it eliminates the need for a separate subtraction operation. This is because subtracting a number is equivalent to adding its opposite (which is found by taking the two's complement).

**Example – Subtraction using two's complement representation**

To carry out the operation 7 – 5 using an 8-bit two's complement representation, we first take the two's complement of 5 to find -5, and then calculate 7 + (-5).

\[
\begin{array}{c}
7 \\
-5 \\
\hline
2
\end{array}
\quad \begin{array}{c}
0000 \ 0111 \\
\rightarrow \ 0000 \ 0101 \\
+ 1111 \ 1011 \\
0000 \ 0010 = 2
\end{array}
\]

**Hexadecimal**

*Hexadecimal* (hex, base 16) provides a compact way of representing binary numbers.

Because a 4-bit word can represent 16 possible values (2<sup>4</sup> = 16), it can be also represented by a single hex digit.

**Example – Binary to hexadecimal conversion**

A simple way to convert a binary number to hexadecimal is to split the binary number into 4-bit words, then convert each word to its hex equivalent.

Convert 0100111010101<sub>2</sub> into hexadecimal.

\[
\begin{align*}
0100111010101_2 \\
\Rightarrow 01 \ 0011 \ 1101 \ 0101_2 \\
\Rightarrow 1 \ 3 \ D \ 5_{16}
\end{align*}
\]
Floating-point representation

Floating-point representation is a method of storing numbers in which a computer word is divided three parts that represent the sign, the exponent, and the mantissa.

\[ \pm M \times 2^E \quad \text{where} \quad M \text{ is the mantissa} \]

\[ E \text{ is the exponent in two's complement} \]

Floating-point representations support a trade-off between range (determined by the number of bits used to represent the exponent) and precision (determined by the number of bits used to represent the mantissa).

Example – A 16-bit floating-point representation

Consider a 16-bit floating-point representation in which:
- The most significant bit is the sign bit, with 0 for positive numbers and 1 for negative numbers.
- The following 5 bits represent the exponent in 2's complement.
- The remaining 10 bits represent the mantissa, where \( 0 \leq \text{mantissa} < 1 \)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign (S)</td>
<td>Exponent (E)</td>
<td>Mantissa (M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 bit</td>
<td>5 bits</td>
<td>10 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The largest possible number using this format is: \( 0011111111111111 \rightarrow 0.1111111111 \times 2^{1111} \)
- The smallest positive number is: \( 0100001000000000 \rightarrow 0.1 \times 2^{-10000} \)

Using this representation, interpret the following 16-bit words as decimal numbers.

a) \( 0111001100000000 \)

- sign = 0 \( \rightarrow \) positive (+)
- exponent = 11100 \( \rightarrow -4 \)
- mantissa = 0.11 \( \rightarrow 0.5 + 0.25 = 0.75 \)

decimal value \( = +0.75 \times 2^{-4} \)
\( = +0.046875 \)

b) \( 1001111010000000 \)

- sign = 1 \( \rightarrow \) negative (-)
- exponent = 00111 \( \rightarrow 7 \)
- mantissa = 0.101 \( \rightarrow 0.5 + 0.125 = 0.625 \)

decimal value \( = -0.625 \times 2^7 \)
\( = -80 \)
Representation of characters

Characters are represented using an encoding system in which each character is represented by a number, stored as a binary word. Two of the most common character encoding systems are ASCII and Unicode.

ASCII

American Standard Code for Information Interchange (ASCII) is the system that most modern character encoding systems are based on. It was originally based on English and encodes 128 characters into 7-bit integers, but numerous 8-bit variations of ASCII have been developed. These typically preserve the original ASCII character mapping and add extra character definitions after the first 128 characters.

Unicode

Unicode is the character encoding system used by Java. It 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. The Unicode standard was initially designed using 16 bits to encode characters. The numerical values are unsigned 16-bit values between 0 and 65535 (65536 characters in total).

Representation of arrays

Consider the following definition for an array:

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

This can be represented diagrammatically as follows:

```
data

<table>
<thead>
<tr>
<th>length</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Consider the following definition for a two-dimensional array:

```java
int[][] data2D = new int[3][2];
```

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

One way to represent an image is to consider it as a collection of *pixels* (picture elements). An image that is 400 pixels wide and 300 pixels high has a total of $400 \times 300 = 120,000$ pixels.

Black and white images only need one bit per pixel. For example, an uncompressed black and white 400 x 300 image requires 15,000 bytes (120,000 bits/8).

Colour pictures need additional bits to represent the range of possible colours. *Colour depth* is the number of bits used to represent the colour of a single pixel. For example, a 16-bit colour system supports up to $2^{16} = 65,536$ colours.

An uncompressed 16-bit 400 x 300 image requires 240,000 bytes ($400 \times 300 \times 16 = 1,920,000$ bits/8).

Representation of sounds

*Sampling* is the process of turning analogue sound waves into digital (binary) signals. The system samples the sound by taking snapshots of its amplitude at regular intervals. In general, sampling should happen at twice the highest frequency to be recorded. Sample depth is the number of bits used to encode each sample. In general, the higher the sample rate and sample depth, the more accurately the digital sound reflects its real-life source and the larger the disk space required to store it.

For example, a 2-minute audio file sampled at a rate of 44.1KHZ and a depth of 16 bits requires 10,584,000 bytes ($120 \times 44100 \times 16 = 84,672,000$bits/8).

Data compression

Data compression is the process of encoding information using fewer bits than an unencoded representation would use.

Lossless compression schemes are reversible, which means that the original data can be reconstructed. Lossy schemes accept some loss of data in order to achieve higher compression.
### ASCII table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Char</th>
<th>Binary</th>
<th>Dec</th>
<th>Char</th>
<th>Binary</th>
<th>Dec</th>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>0000 0000</td>
<td>48</td>
<td>0</td>
<td>0011 0000</td>
<td>96</td>
<td>`</td>
<td>0110 0000</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>0000 0001</td>
<td>49</td>
<td>1</td>
<td>0011 0001</td>
<td>97</td>
<td>a</td>
<td>0110 0001</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>0000 0010</td>
<td>50</td>
<td>2</td>
<td>0011 0010</td>
<td>98</td>
<td>b</td>
<td>0110 0010</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>0000 0011</td>
<td>51</td>
<td>3</td>
<td>0011 0011</td>
<td>99</td>
<td>c</td>
<td>0110 0011</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>0000 0100</td>
<td>52</td>
<td>4</td>
<td>0011 0100</td>
<td>100</td>
<td>d</td>
<td>0110 1000</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>0000 0101</td>
<td>53</td>
<td>5</td>
<td>0011 0101</td>
<td>101</td>
<td>e</td>
<td>0110 1001</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>0000 0110</td>
<td>54</td>
<td>6</td>
<td>0011 0110</td>
<td>102</td>
<td>f</td>
<td>0110 1010</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>0000 0111</td>
<td>55</td>
<td>7</td>
<td>0011 0111</td>
<td>103</td>
<td>g</td>
<td>0110 1011</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>0000 1000</td>
<td>56</td>
<td>8</td>
<td>0011 1000</td>
<td>104</td>
<td>h</td>
<td>0110 1000</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>0000 1001</td>
<td>57</td>
<td>9</td>
<td>0011 1001</td>
<td>105</td>
<td>i</td>
<td>0110 1001</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>0000 1010</td>
<td>58</td>
<td>;)</td>
<td>0011 1010</td>
<td>106</td>
<td>j</td>
<td>0110 1010</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>0000 1011</td>
<td>59</td>
<td>;</td>
<td>0011 1011</td>
<td>107</td>
<td>k</td>
<td>0110 1011</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>0000 1100</td>
<td>60</td>
<td>&lt;</td>
<td>0011 1100</td>
<td>108</td>
<td>l</td>
<td>0110 1100</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>0000 1101</td>
<td>61</td>
<td>=</td>
<td>0011 1101</td>
<td>109</td>
<td>m</td>
<td>0110 1101</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>0000 1110</td>
<td>62</td>
<td>&gt;</td>
<td>0011 1110</td>
<td>110</td>
<td>n</td>
<td>0110 1110</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>0000 1111</td>
<td>63</td>
<td>?</td>
<td>0011 1111</td>
<td>111</td>
<td>o</td>
<td>0110 1111</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
<td>0001 0000</td>
<td>64</td>
<td>@</td>
<td>0100 0000</td>
<td>112</td>
<td>p</td>
<td>0111 0000</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
<td>0001 0001</td>
<td>65</td>
<td>A</td>
<td>0100 0001</td>
<td>113</td>
<td>q</td>
<td>0111 0001</td>
</tr>
<tr>
<td>18</td>
<td>DC2</td>
<td>0001 0010</td>
<td>66</td>
<td>B</td>
<td>0100 0010</td>
<td>114</td>
<td>r</td>
<td>0111 0010</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
<td>0001 0011</td>
<td>67</td>
<td>C</td>
<td>0100 0011</td>
<td>115</td>
<td>s</td>
<td>0111 0011</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
<td>0001 0100</td>
<td>68</td>
<td>D</td>
<td>0100 0100</td>
<td>116</td>
<td>t</td>
<td>0111 0100</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
<td>0001 0101</td>
<td>69</td>
<td>E</td>
<td>0100 0101</td>
<td>117</td>
<td>u</td>
<td>0111 0101</td>
</tr>
<tr>
<td>22</td>
<td>SYN</td>
<td>0001 0110</td>
<td>70</td>
<td>F</td>
<td>0100 0110</td>
<td>118</td>
<td>v</td>
<td>0111 0110</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
<td>0001 0111</td>
<td>71</td>
<td>G</td>
<td>0100 0111</td>
<td>119</td>
<td>w</td>
<td>0111 0111</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
<td>0001 1000</td>
<td>72</td>
<td>H</td>
<td>0100 1000</td>
<td>120</td>
<td>x</td>
<td>0111 1000</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
<td>0001 1001</td>
<td>73</td>
<td>I</td>
<td>0100 1001</td>
<td>121</td>
<td>y</td>
<td>0111 1001</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
<td>0001 1010</td>
<td>74</td>
<td>J</td>
<td>0100 1010</td>
<td>122</td>
<td>z</td>
<td>0111 1010</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
<td>0001 1011</td>
<td>75</td>
<td>K</td>
<td>0100 1011</td>
<td>123</td>
<td>{</td>
<td>0111 1011</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
<td>0001 1100</td>
<td>76</td>
<td>L</td>
<td>0100 1100</td>
<td>124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
<td>0001 1101</td>
<td>77</td>
<td>M</td>
<td>0100 1101</td>
<td>125</td>
<td>}</td>
<td>0111 1101</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
<td>0001 1110</td>
<td>78</td>
<td>N</td>
<td>0100 1110</td>
<td>126</td>
<td>~</td>
<td>0111 1110</td>
</tr>
<tr>
<td>31</td>
<td>US</td>
<td>0001 1111</td>
<td>79</td>
<td>O</td>
<td>0100 1111</td>
<td>127</td>
<td>DEL</td>
<td>0111 1111</td>
</tr>
<tr>
<td>32</td>
<td>Space</td>
<td>0010 0000</td>
<td>80</td>
<td>P</td>
<td>0101 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>0010 0001</td>
<td>81</td>
<td>Q</td>
<td>0101 0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>0010 0010</td>
<td>82</td>
<td>R</td>
<td>0101 0010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>0010 0011</td>
<td>83</td>
<td>S</td>
<td>0101 0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>0010 0100</td>
<td>84</td>
<td>T</td>
<td>0101 0100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>0010 0101</td>
<td>85</td>
<td>U</td>
<td>0101 0101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>0010 0110</td>
<td>86</td>
<td>V</td>
<td>0101 0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>0010 0111</td>
<td>87</td>
<td>W</td>
<td>0101 0111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>0010 1000</td>
<td>88</td>
<td>X</td>
<td>0101 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>0010 1001</td>
<td>89</td>
<td>Y</td>
<td>0101 1001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>0010 1010</td>
<td>90</td>
<td>Z</td>
<td>0101 1010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>0010 1011</td>
<td>91</td>
<td>[</td>
<td>0101 1011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>0010 1100</td>
<td>92</td>
<td>\</td>
<td>0101 1100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>0010 1101</td>
<td>93</td>
<td>]</td>
<td>0101 1101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>0010 1110</td>
<td>94</td>
<td>^</td>
<td>0101 1110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>0010 1111</td>
<td>95</td>
<td>_</td>
<td>0101 1111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>