PART 3: OBJECTS AND CLASSES

This part covers design of objects – key Java skill. Without this you can't write significant Java programs (or pass this course). Based on Chapter 3 of CCJE.

- Designing objects.
- The Student Class.
- Objects and reference values.
- The `null` value.

Designing Objects

Objects provide way to represent almost any entity from real world inside your programs, eg: students, bank accounts, railway networks, etc.

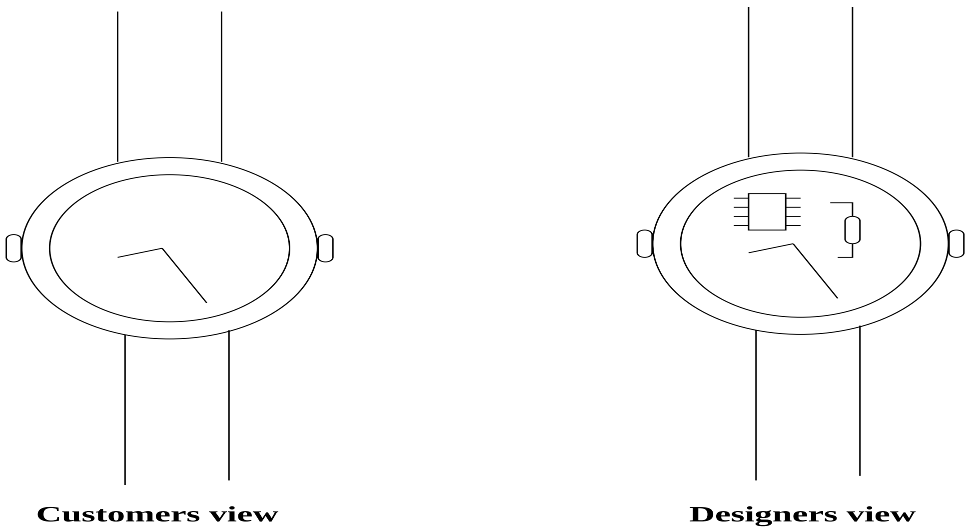
Form basis of programming in languages such as C++ and Java.

What is an object? Two ways of looking at an object: point of view of programmer who uses object (customer), and point of view of programmer who designs object (designer).

Applies to devices such as watches, also to any type of Java object. Some programmer will have designed it. They will know exactly what it consists of, and exactly how the operations that can be performed on the object work.

Other programmers will use this type of object. They are 'customers' who only know some useful operations of the object.

Customer and Designer Views



Objects – Customer and Designer Views

Eg, many different types of objects available in the Java library. Designed by programmers at Sun Microsystems. Java programmers everywhere are the customers.

We know some useful methods provided with each object, such as *println* and *print* methods of *System.out* object.

So one answer to 'What is an object?' is customer's answer: we don't know what it consists of; we just know useful things can do with it.

As programmers we will also design objects – define what data they store and how operations they provide affect and use this data.

Student Database Application

To develop: Database program in Java that will record information about all students studying in Computer Science Department.

This program will use many different kinds of objects:

- objects that represent individual students;
- an object that holds list of all students in database;
- objects that represent courses, lecturers, etc.

Construct *Student* class this week – template for student objects (defines their data and operations).

Class Diagram of Student Entity

Student
idNumber: String **name: String** **degree: String** **year: int**
setDegree(deg: String) **advanceYear()** **getName(): String** **display()** **Student(id: String,** **nn: String,** **deg: String)**

Design of Student Object/Class

The design of a Student object is contained in a piece of program called a *class definition*.

Start by identifying what are the required operations:

1. It should be possible to change student's degree programme. So need a method *setDegree* that replaces student's current degree programme with a new one.
2. It should be possible to change student's year of study on from 1 to 2, 2 to 3 or 3 to 4. So need method *advanceYear* to do this.
3. Need method *getName* to get student's name.
4. It should be possible to display all details of a student on screen. Need method *display* to do this.
5. It will be necessary to create new *Student* objects. Need method that takes student's ID number, name and degree

programme and creates a year 1 student object with this data.

Last operation will be carried out by a *constructor*. A constructor is like a method in that it has a body, a name and parameters. However, name must always be same as the class it belongs to. So name of this constructor will be *Student*.

Members/features

Structure of a class consists of a number of separate items called *members* or features. There are three kinds of members.

1. Variables/attributes. These record data of object. Eg, a *Student* object has variable that contains student's name. Sometimes called *fields* (or *instance variables*).
2. Methods. These carry out operations that can be performed on an object.
3. Constructors. These create objects belonging to the class.

Each member is declared to be either *public* or *private*. Public members can be used by customer or designer programmer. Private members only by designer programmer, within class that lists them.

Instance variables of Student

Assume each Student object contains following 4 items of information:

1. Student ID number.
2. Name.
3. Degree programme.
4. Year of study.

Each will be held in attribute of appropriate type:

```
public class Student
{ private String idNumber;
  // The student's ID number.
  private String name;
  // The student's name.
  private String degree;
  // The student's degree programme.
```

```
  private int year;
  // The student's year
  // of study (1, 2 or 3, 4).
}
```

This is a valid class definition, but not a very useful one as yet, since no constructor to create objects or methods to operate on them!

Could define initial values for attributes, eg:

```
  private int year = 1;
```

although these can be left for constructor to define instead.

It is usual for instance variables to be *private*.

Methods of Student

Earlier decided on four methods of *Student*. All these will be public – otherwise customers would not be able to use them.

First is *setDegree* method – changes student's degree programme. When it is called (i.e. executed) needs to know name of new degree programme. This will be a *parameter* of the method. In general, methods can have any number of parameters. *setDegree* has just this one, called *deg* here.

The body of a method consists of a sequence of statements that will carry out whatever actions are needed.

In this case, only one action needed: new degree programme (i.e. parameter *deg*) should be stored in *degree* attribute:

```
degree = deg;
```

Here is complete definition of *setDegree*.

As usual starts with a comment that says what method does.

```
/* Change the student's degree
   programme to deg. */
public void setDegree(String deg)
{ degree = deg; }
```

Next line is the method heading. This is what the different parts mean.

1. `public` says method can be used anywhere in the program.
2. `void` says method does not return any value.
3. `setDegree` is name of the method.
4. `(String deg)` is list of all the method's parameters.

Finally comes the body of the method, enclosed in {}.

Using Instance Methods

Suppose customer programmer is writing some program code in part of program outside the class Student. Suppose a particular Student object has been created, *fred*.

Then if programmer wants to set this student's degree progamme to `"CS/Man"`, can write statement

```
fred.setDegree("CS/Man");
```

This says: execute method *setDegree* that belongs to Student object *fred*, using parameter value `"CS/Man"`.

Outcome will be that statement in body of method will set the object's degree to `"CS/Man"`.

advanceYear Method

advanceYear is simpler than *setDegree* because it has no parameters. All body of method does is add 1 to year attribute. So just consists of single statement

```
year++;
```

Here is the complete method:

```
/* Increase the student's
   year of study by 1. */
public void advanceYear()
{ year++; }
```


advanceYear method

This is what the heading says:

1. *public* means method can be used in any part of program.
2. *void* means method does not return a value.
3. *advanceYear* is the method's name.
4. () shows there are no parameters.

Outside the *Student* class, following statement could be used to add 1 to year of Student object *fred*:

```
fred.advanceYear();
```

getName method

Next method is *getName*. Unlike two previous methods, it does 'return a value'.

For example, the method is called when Java executes statement:

```
String st = fred.getName();
```

Java will create variable *st*, then it will execute the method call

```
fred.getName()
```

The method will return name of this student, and Java will store it in variable *st*.

fred.getName() is used as an expression, in same way that *Math.sqrt*(1.5) is used as an expression in:

```
double x = 1 - Math.sqrt(1.5);
```

Methods that return values are always used like this.

The body of *getName* is a single statement. It says: return value of *name* attribute:

```
return name;
```

This is called a *return statement*.

Here is the complete method.

```
/* Return the student's name. */
public String getName()
{ return name; }
```

This is what the heading says:

1. `String` – method returns a value of type String. This is called the *return type*. *void* is written as return type when a method does not return a value.
2. *getName* is the method's name.
3. () shows method has no parameters.

display method

The last method is *display*. It displays on screen all the information in a Student object's attributes:

```
/* Display details of the student
   on the screen. */
public void display()
{ System.out.println("Student number: " + idNumber);
  System.out.println("Name: " + name);
  System.out.println("Degree programme: " + degree);
  System.out.println("Year: " + year);
}
```

Heading indicates that method does not return a value, and it does not have any parameters.

Example of use of this method:

```
fred.display();
```


Constructor method

Constructor that customer programmer can use to create new Student objects:

```
/* Create new student with
   the given ID number, name
   and degree.
   The year will be set to 1.
 */
public Student(String id, String nm, String deg)
{ idNumber = id;
  name = nm;
  degree = deg;
  year = 1;
}
```

A constructor is always given the name of its class, *Student* in this case.

A constructor can have any number of parameters. This one has three.

Given three parameter values *id*, *deg* and *num*, it will

1. create a new object,
2. store *id* in *idNumber* of new object,
3. store *nm* in *name* attribute,
4. store *deg* in *degree* attribute,
5. store 1 in *year* attribute.

Nothing in constructor that indicates first step that creates the new object. Java always does this automatically when it executes a constructor.

Use of constructor

Here is this constructor being used in a statement to create a new Student object.

```
Student st = new Student("9912345", "Brown,Alice", "CS");
```

This says: first create a new variable called *st* to hold a *Student* object, then create a new student object with ID number `"9912345"`, name `"Brown,Alice"`, and degree programme `"CS"`. Then store this object in *st*.

Note *new* keyword, and that *st* must be of type *Student* as it will be holding a *Student* object.

EXAMPLE 3.1: Complete Student class

```
/* A Student object is a record of a BSc student. */
public class Student
{ private String idNumber;
  // The student's ID number.
  private String name;
  // The student's name.
  private String degree;
  // The student's degree programme.
  private int year;
  // The student's year
  // of study (1, 2, 3, 4).

  /* Create new student with the given ID number,
     name and degree.  Year will be set to 1. */
  public Student(String id, String nm, String deg)
  { idNumber = id;
```

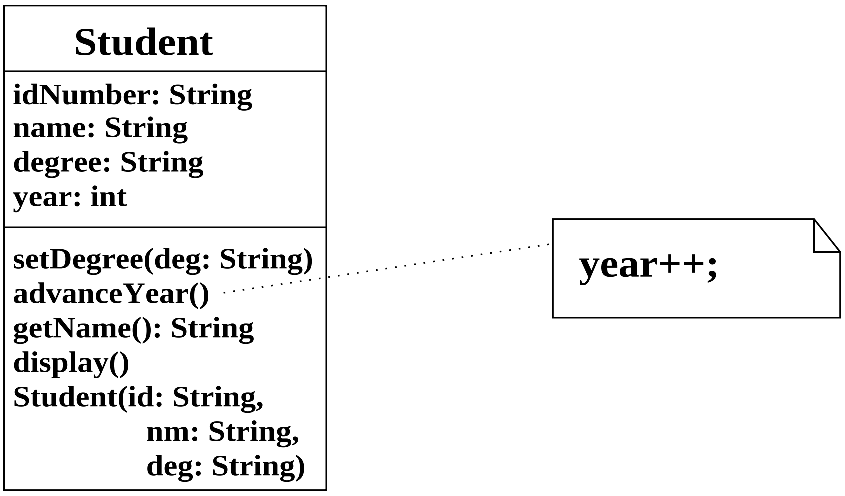


```
    name = nm;
    degree = deg;
    year = 1;
  }

  /* Change this student's degree programme to deg. */
  public void setDegree(String deg)
  { degree = deg; }

  /* Increase this student's year by 1. */
  public void advanceYear()
  { year++; }

  /* Return this student's name. */
  public String getName()
  { return name; }

  /* Display details of this student on the screen. */
```

```
  public void display()
  { System.out.println("Student number: " + idNumber);
    System.out.println("Name: " + name);
    System.out.println("Degree programme: " + degree);
    System.out.println("Year: " + year);
  }
}
```

Can represent *Student* class graphically as a UML class diagram:

Class Diagram of Student



Using Student objects

Finally here is a very simple program that makes use of a *Student* object. It consists of single static method called *main*, placed in class *TestStudent*.

The method first reads information about a student entered by user. Then creates a *Student* object using constructor, and assigns it to variable *st*.

Then it increases year of student by 1, using statement

```
st.advanceYear();
```

Then it sets course to "CS with Man".

Finally details of student are displayed by

```
st.display();
```


Using Student objects

Here is the *TestStudent* class. *EXAMPLE 3.2*

```
public class TestStudent
{ /* Read details of a student.
     Then create the corresponding Student object,
     and display that object's details.
   */
  public static void main(String[] args)
  { /* Create a ConsoleReader for reading
       what the user types. */
    ConsoleReader in =
        new ConsoleReader(System.in);
    /* Read information about a student
       and create an object for them. */
    System.out.println("What is the student's ID number?");
    String i = in.readLine();
    System.out.println("What is their name?");
```

```
    String n = in.readLine();
    System.out.println("What is their degree?");
    String d = in.readLine();
    Student st = new Student(i,n,d);
    /* Change the student's year
       and display the student's details. */
    st.advanceYear();
    st.setDegree("CS with Man");
    System.out.println();
    st.display();
  }
}
```


Using Student objects

When this program is run, could get:

```
What is the student's ID number?
9912345
What is their name?
Brown, Alice
What is their degree programme?
Maths and CS

Student number: 9912345
Name: Brown, Alice
Degree programme: CS with Man
Year: 2
```

Three classes needed to run this program: *TestStudent*, *Student* and *ConsoleReader*.

They should be in three files called TestStudent.java, Student.java and ConsoleReader.java, all in the same directory.

Compile them by invoking

```
javac TestStudent.java
```

on DOS command line in this directory.

Next give the command

```
java TestStudent
```


Object References

Primitive values such as integers and floating-point values are always stored in a variable. Eg, if declaration

```
int n = 5;
```

is executed, 32 bits of memory will be set aside for new variable n, and value 5 will be stored there.

However an object is never stored in a variable, only a *reference* to it is stored (cf. strings).

Object References

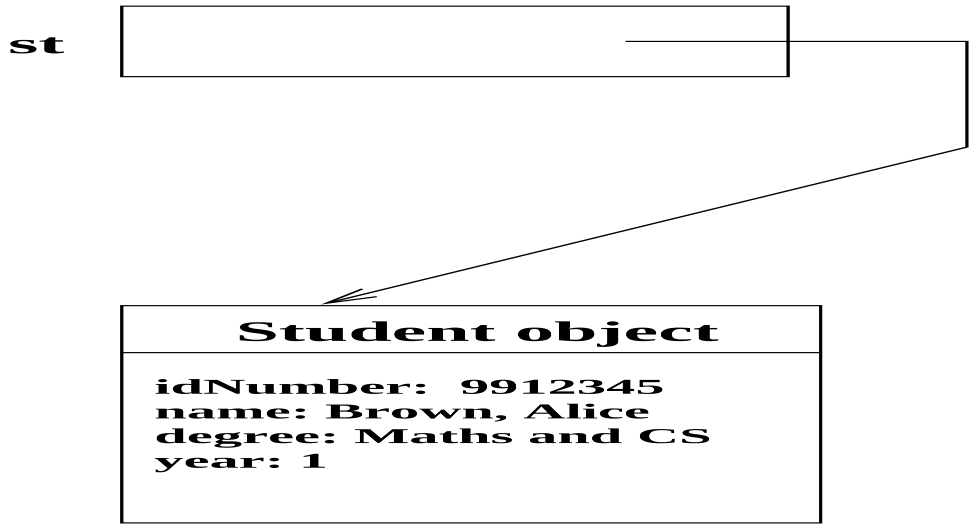
If declaration

```
Student st =
  new Student("9912345","Brown,Alice","Maths and CS");
```

is executed, an area of memory will be allocated to variable, *st*, but new object that is created will not be stored there.

It will be given its own location in memory, quite separate from that of *st*. This location/address is stored in *st*.

Memory layout for objects



Object References

When a Java program is running, each variable either contains a primitive value such as an integer or boolean value, or it contains a reference to an object.

Similarly, if an expression is evaluated, result will either be a primitive value or a reference to an object.

We never get to see what reference values look like.

Java imposes strict rule that only a variable that has been declared to be of type *Student*, for example, can contain reference to a *Student* object.

[This helps you avoid simple program errors, like trying to assign a string to a student variable]

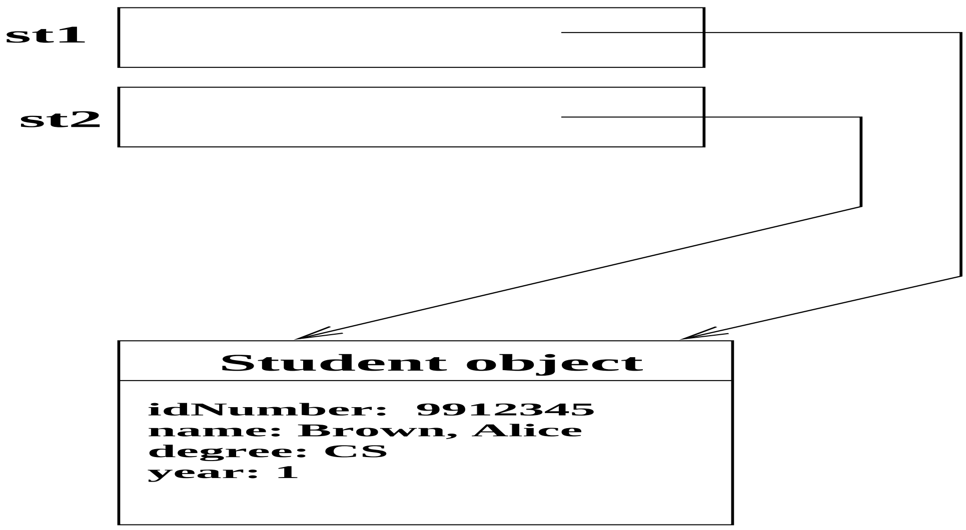
Object References

Consider the code:

```
Student st1 = new Student("9912345",
                          "Brown,Alice",
                          "Maths and CS");
Student st2 = st1;
st1.setDegree("CS");
st2.display();
```

The degree programme displayed for $st2$ is `"CS"` because $st1$ and $st2$ are *aliased* by the assignment $st2 = st1$ – they hold the same object reference, so a method call $s.m(x)$ on either affects *both* variables identically.

Aliasing of $st1$ and $st2$



Object References

If actual objects were stored in variables, then first statement would store Student object in $st1$.

Second statement would put a copy of it in $st2$. In that case, there would be two objects, one in $st1$ and one in $st2$.

Third statement would change degree of object in $st1$. It would not change object in $st2$.

Fourth statement would output details of object in $st2$, including original degree programme, `"Maths and CS"`.

The `null` *Value*

null reference can be assigned to any variable that normally holds a reference value.

Eg:

```
Student st = null;
```

In student database program could define method which searches database for student with particular ID number.

If right Student object is found, method returns reference of object. If no student with the given ID number, return *null*.

If *st* is *null* cannot invoke any method on it:

```
st.setDegree("CS");
```

A run-time error, *NullPointerException*, will occur if Java tries to obey this statement.

Default Initial Values

If instance variable is not given an initial value in its declaration by the programmer, Java will choose a default value for you.

If attribute holds any kind of number, default value will be zero.

If attribute denotes an object (i.e. it contains an object reference), default value will be *null*.

For example, three attributes of a *Student* object are of type String. Strings are objects, so they will initially contain null.

null is not same as the empty string. Empty string is a normal string that has a reference value.

Advice: Explicitly initialise instance variables where possible to ensure they have valid initial value.

Key Points

- **What is a class?** A class is a group of attributes (data) and methods. If class C represents a program, then must be declared as *public class C* in a file *C.java*, and must have a *main* method, which is where execution starts when we type *java C* on command line.
- **What is a method?** A method is a named piece of executable code. When method is executed, statements inside it are executed in order (from top to bottom of the method text).
- **What is a variable?** A variable is a named piece of computer memory. Once declared, variables can be updated and read. We can store integers in an *int* variable, rational numbers in a *double* variable, and strings – sequences of characters – in a *String* variable.

Part 4: GRAPHICS

Java 2 has rich set of graphics classes, providing drawing and painting operations, using variety of geometrical shapes and a wide range of colours and patterns. Text and existing images such as photographs can be incorporated.

- Using graphics
- Writing applets
- Rectangles and ellipses
- Introducing colour.
- Painting shapes.
- Writing text.
- Reading input.