

CSC1016S Assignment 2

Simple Classes with Constructors, Methods; and Junit

Introduction

This assignment concerns reinforcing OO concepts through creating and manipulating types of object; and the writing of simple class declarations.

Key points are that an OO programmer (i) often uses predefined program components i.e. classes, (ii) often develops program components, not whole programs and (iii) needs techniques and tools for checking/evaluating their work.

Exercise one concerns writing a simple program using predefined classes.

Exercise two also involves predefined classes. In this case, however, your task is to develop a suite of Junit tests to demonstrate those classes function correctly.

Exercises three and four involve constructing class declarations for simple types of object. Question three uses the classes provided for question two, and question four uses the class developed in question three.

The exercises are themed. They concern modelling aspects of a pay-to-stay car. The kind of car park in question has a ticket machine at the entrance and a cashier at the exit. A driver, on entering the car park receives a ticket stamped with the arrival time. (The arrival time is also recorded on the magnetic strip on the back.) On exit, the driver gives the ticket to the cashier, the duration of the stay is calculated and from that, how much must be paid.

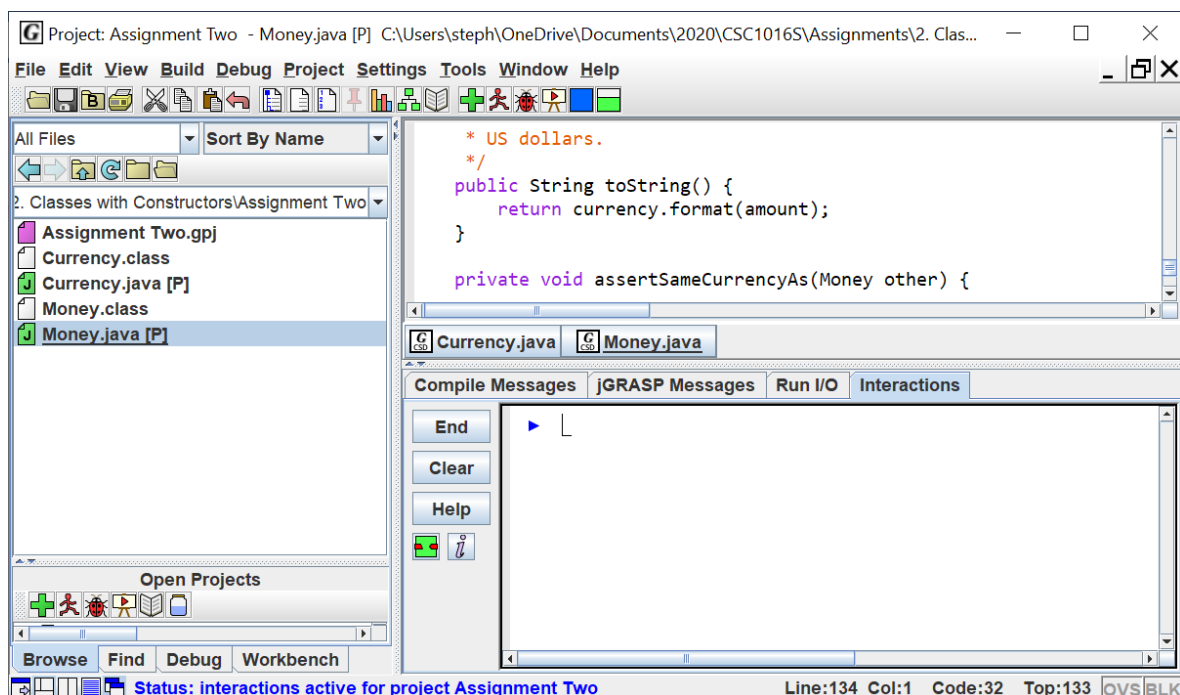
The classes you will encounter are Time, Duration, Money, Currency, Ticket and Register.

JGrasp Interactive feature

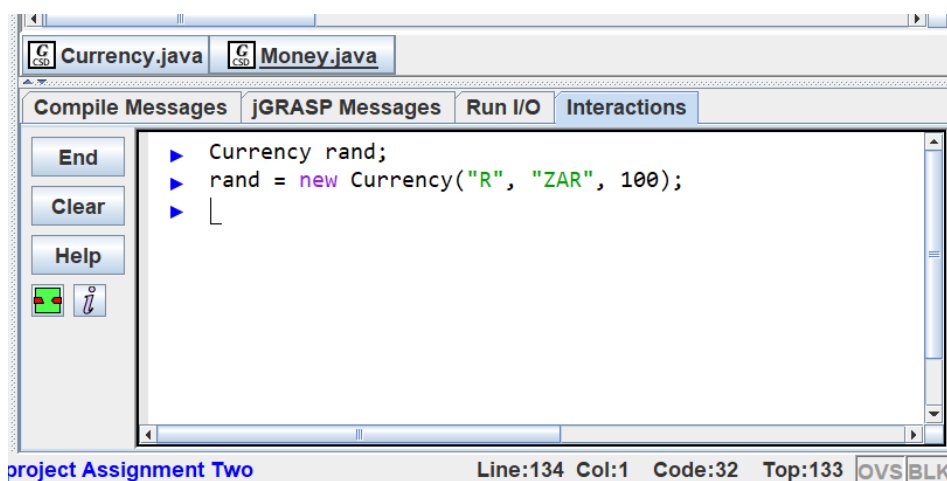
A useful tool evaluating work on the fly and for experimenting with Java is the JGrasp interactive feature. This is a part of JGrasp that allows you to enter Java statements and expressions and have them evaluated in much the same way as the Python Shell in the Wing 101 IDE (used in CSC1015F).

To illustrate, question one involves the use of a Money class and a Currency class. Let's say that we wish to confirm our understanding of these classes by experimenting with creating and manipulating Money and Currency objects; specifically, we would like to see if we can devise the correct sequence of statements to add the sums R25 and R16.50 together.

Assuming we have created a new JGrasp project for the assignment, have added the Money.java and Currency.java files from the Vula assignment page, and compiled them, we start by clicking on the 'Interactions' tab.

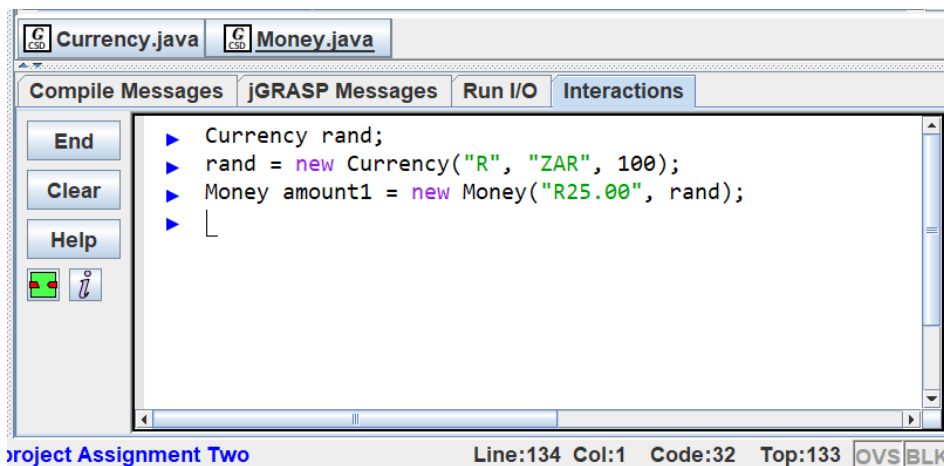


We can then enter our statements. We start by creating a Currency object to represent South Africa Rand:



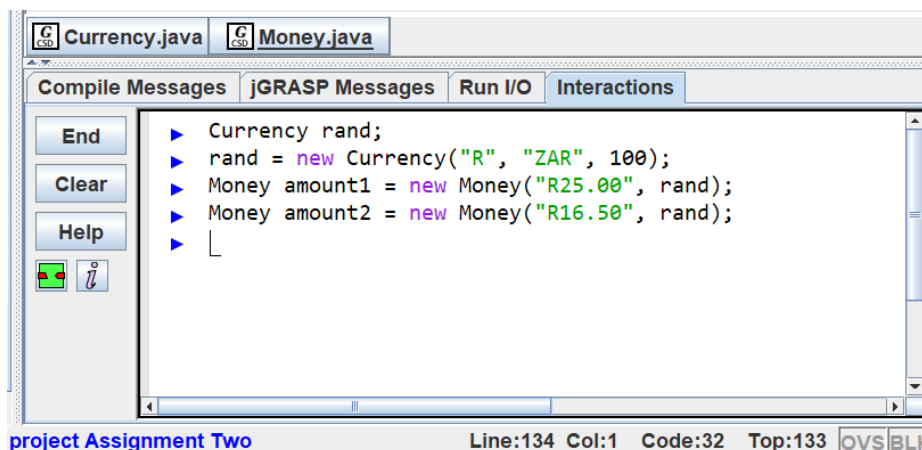
We declare a variable called 'rand' that will store a Currency. Then we create a new Currency object with the actual parameter values "R", "ZAR" and 100 and assign it to the variable. ('R' is the symbol used for South African Rand amounts, 'ZAR' is the ISO 4217 code for the South African Rand, 100 is the number of minor units (cents) in a Rand.)

Now that we've made a Currency object, we can create a Money object representing R25.

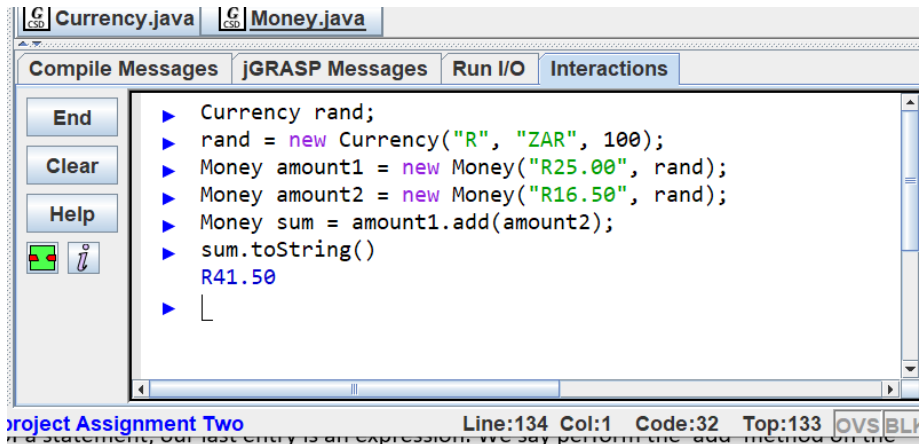


In this case we've declared a variable, created an object and performed an assignment all in one statement. The variable is called 'amount' and stores a Money value. The Money object is created with the actual parameter values "R25.00" and 'rand' the name of the variable referring to our Currency object.

We can create a second Money object representing R16.50.



We've assigned it to a variable called 'amount2'. And now we can try adding the amounts together.



```
▶ Currency rand;  
▶ rand = new Currency("R", "ZAR", 100);  
▶ Money amount1 = new Money("R25.00", rand);  
▶ Money amount2 = new Money("R16.50", rand);  
▶ Money sum = amount1.add(amount2);  
▶ sum.toString()  
  R41.50  
▶ |
```

project Assignment Two Line:134 Col:1 Code:32 Top:133 OVS BLK

We declare a final variable called 'sum' and assign to it the result of adding the two amounts.

We perform the 'add' method on the Money object referred to by 'amount1', passing the Money object referred to by 'amount2' as a parameter. The result of the addition is a new Money object.

Our final entry is not a statement but an expression, we perform the 'toString' method on the Money object referred to by 'sum'. The result is a string representation of the amount. Since we haven't tried to assign it to anything, JGrasp just prints it out.

Exercise One [25 marks]

This exercise involves the use of Money and Currency objects.

Class Money

An object of this class represents an amount of money in a particular currency. Amounts can be added and subtracted. The amount is stored as a quantity of the minor unit of the currency e.g. 1 Rand will be stored as 100 cents.

Constructors

```
public Money(String amount, Currency currency)
    // Create a Money object that represents the given amount of the given currency.
    // The String is assumed to have the following format: <currency symbol><quantity of
    // units>.<quantity of minor units> e.g. in the case of USD, $50.30, $0.34.
```

Methods

```
public Money add(Money other)
    // Add the other amount of money to this amount and return the result. The objects must be of the
    // same currency.

public String toString()
    // Obtain a string representation of the monetary value represented by this object e.g. "€45.10" for
    // a Money object that represents 45 euros and 10 cents.
```

Class Currency

An object of this class represents a Currency such as US Dollars or British Pound Stirling.

A currency has an ISO 4217 currency code and a symbol denoting the currency's major unit. Many currencies have a minor (or fractional) unit such as the cent in the case of US dollars.

A currency object provides facilities for creating strings and for interpreting strings that represent amounts of the currency.

It is assumed that the currency symbol always appears in front of an amount; that negative amounts are represented by a minus sign, '-', that precedes the currency symbol, "-£34.50" for example; that the decimal point is always represented using a full stop; that no attempt is made to group the digits of large quantities, so for example, one million Rand is assumed to be represented as "R1000000" (as opposed to "R1,000,000").

Constructors

```
public Currency(String symbol, String code, int minorPerMajor)
    // Create a Currency object that represents the currency with the given unit symbol (e.g. "£" for
    // Sterling), ISO 4217 code, and number of minor units per major units (e.g. 100 in the case of
    // pennies per British Pound).
```

This style of describing types of object will often be used in the course. A specification gives:

- The name of the class (type) of object and a brief description of what the objects represent.
- A description of constructors, the ways of creating instances of the type of object.
- A description of methods, the ways of manipulating instances of the type of object.

The description of a constructor or method comprises the signature (name, formal parameters), and the behaviour i.e. what it does.

Consider the following code snippet:

```
Currency rand = new Currency("R", "ZAR", 100);
Money money = new Money("R14.50", rand);
System.out.println(money.toString());
```

The code creates a Currency object that represents ZAR (South African Rand), and a Money object that represents the amount R14.50. It uses 'toString()' to get a string representation of the Money object that it then prints out.

Write a program called "SumCosts.java" that asks the user to enter a series of Rand amounts, sums them, and prints the result.

Sample I/O:

```
Enter an amount or '[D]one' to quit:
Done
```

Total: R0.00

Sample I/O:

```
Enter an amount or '[D]one' to quit:
R0.99
Enter an amount or '[D]one' to print the sum and quit:
R15
Enter an amount or '[D]one' to print the sum and quit:
R17.95
Enter an amount or '[D]one' to print the sum and quit:
D
Total: R33.94
```

You must use the classes and methods described.

Your program will require a 'while-loop'. The Java syntax is very similar to that of Python. Here's an example:

```
int i=0;
while (i<10) {
    i=i+1;
    System.out.println(i);
}
```

The code prints out, one per line, the values 1 to 10 .

Unlike Python, the terminating condition is enclosed in brackets. The body of the while loop – the code to be executed – is enclosed in curly braces.

NOTES/HINTS:

- To determine whether one string, s1, is equal to another, s2, use the 'equals' method e.g. 's1.equals(s2)'.
- The string method 'charAt' accepts an index value as a parameter and, when applied to a string, returns the character at that position e.g. the expression "CSC1016S".charAt(3) produces '1'.
- The expression "CSC1016S".charAt(2)=='f' is false while the expression "CSC1016S".charAt(1)=='S' is true.

Exercise Two [25 marks]

Unit testing

Unit testing is an automatic testing technique for individual modules of a large software system. Tests are defined and executed automatically to rapidly test and re-test a class during development without user intervention. This helps to scale up testing when a program gets very large. It also ensures the correctness of each individual class within the system.

JUnit is one of the most popular unit testing frameworks for Java. The advantage of a framework (rather than writing small programs to test each class) is that there is a standard API, common understanding of the testing approach among Java programmers and integrated support in many IDEs.

Time and Duration classes

Study the following specifications for Time and Duration types of object.

Class Time

A Time object represents a twenty-four-hour clock reading composed of hours, minutes and seconds.

Constructors

Time(String reading)

*// Create a Time object from a string representation of a twenty-four-hour clock reading
// of the form 'hh:mm[:ss]' e.g. "03:25", "17:55:05".*

Methods

public Duration subtract(Time other)

// Obtain the Time that results from subtracting the given period from this Time.

public String toString()

// Obtain a String representation of this Time.

Class Duration

A Duration object represents a length of time (with millisecond accuracy).

Methods

public long intValue(String timeunit)

*// Obtain an integer value that represents as much of this duration object that can be expressed
// as a multiple of the given time unit
// For example, given a duration object d that represents 1 hour, 4 minutes, and 30 seconds,
// d.intValue("minute") produces 64.
// Permissible time units are: "millisecond", "second", "minute", "hour"..*

For the Time class we've listed one way of creating it: a Time object can be created from a string representing a particular 24 hour clock time. The following code snippet provides an example:

```
Time t = new Time("13:45");
```

We've given two methods of manipulating a Time object, one is called 'subtract' and the other 'toString'. The subtraction method provides a means for subtracting one time from another to obtain a duration i.e. obtaining the period between them. The other method provides a means of obtaining a string representation of the object i.e. something printable.

Say we had two Time objects referred to as 't1' and 't2', we could subtract t2 from t1 using the expression 't1.subtract(t2)'

The `subtract` method returns a value that is a `Duration` object.

We've given you one method for `Duration` objects, `intValue`. The method accepts a `String` parameter which must be the name of a time unit. It returns an integer representing the number of that unit closest to the duration value. An example is given in the specification: given a duration object, `d`, that represents 1 hour, 4 minutes, and 30 seconds, the expression `d.intValue("minute")` evaluates to 64.

Task

Construct a JUnit test class called `TestOfTime` that contains the following tests of the `Time` and `Duration` classes:

Test Number	Class Name	Test purpose
1.	<code>Time</code>	Check that a <code>Time</code> object does actually store the time value provided as a parameter during creation (by calling <code>toString</code>).
2.	<code>Time</code>	Check that subtracting an earlier <code>Time</code> from a later <code>Time</code> produces a <code>Duration</code> of the correct length.
3.	<code>Time</code>	Check that subtracting a <code>Time</code> from itself produces a zero <code>Duration</code> .
4.	<code>Duration</code>	Check that the <code>intValue</code> method works with a parameter of "millisecond".
5.	<code>Duration</code>	Check that the <code>intValue</code> method works with a parameter of "second".
6.	<code>Duration</code>	Check that the <code>intValue</code> method works with a parameter of 'minute'.
7.	<code>Duration</code>	Check that the <code>intValue</code> method works with a parameter of 'hour'.

Download the JUnit packages from `Resources/Software/JUnit`.

JGrasp has built-in integration with JUnit.

- Go to `Tools/JUnit` and use `Configure` to first set the location of the JUnit packages you downloaded. Then there will be options to run JUnit tests from within JGrasp.
- When you add source files to a JGrasp project, JGrasp will automatically detect which ones are normal source files and which ones are JUnit test classes.

Alternatively, in order to compile your JUnit test class from the command-line, use a command such as:

```
javac -cp junit.jar:hamcrest-core.jar:. TestOfTime.java
```

And in order to execute your JUnit tests from the command-line, use a command such as:

```
java -cp junit.jar:hamcrest-core.jar:. org.junit.runner.JUnitCore
TestOfTime
```


Exercise Three [25 marks]

Your task is to develop a Ticket class of object. A Ticket object represents a car park ticket. It has a unique ID and time of issue (24-hour clock).

Your class must meet the following specification:

Class Ticket

A Ticket object represents a car park ticket. It has a unique ID and time of issue (24 hour clock).

Instance variables

String id;
Time issueTime;

Constructors

Ticket(Time currentTime, String ID)

// Create a new Ticket that has the given issue time and unique ID.

Methods

public String ID()

// Obtain this Ticket's ID.

public Duration age(Time currentTime)

// Obtain this ticket's age i.e. the issue time subtracted from the given time.

public String toString()

// Obtain a String representation of this Ticket object in the form:

// "Ticket[id="dddd", time="hh:mm:ss"]".

Here's a code snippet to illustrate behaviour:

```
//..
Time tOne = new Time("6:50", "800001");
Ticket ticket = new Ticket(tOne);
Time tTwo = new Time("7:19", "8005A3");
System.out.println(ticket.toString());
Duration d = ticket.age(tTwo);
System.out.println(d.intValue("minute"));
//...
```

The output from this code would be:

```
Ticket[id=800001, time=06:50:00].
29
```

The class has a single constructor that is used to (i) assign the current time as the ticket issue time and to (ii) assign a unique ID.

Since this class is not in itself a complete program, how are you to evaluate your work? Consider (i) writing a test program (something like that required for exercise 1), or (ii) developing unit tests using Junit, or (iii) using the jGrasp interactive feature.

Exercise 4 [25 marks]

The job of a Register object in the car park simulation is to store all the tickets that have been issued. When a Ticket is issued, it is stored in the register. When the driver departs, the ticket ID is used to retrieve the Ticket object and calculate the duration of stay.

Class Register

A Register stores a collection of Tickets. A Ticket may be retrieved given its ID.

Instance variables

```
Ticket[] tickets;
int numTickets;
```

Constructors

```
Register()
    // Create a new Register object.
```

Methods

```
public void add(Ticket ticket)
    // Store the given ticket in the register.
public boolean contains(String ticketID)
    // Determine whether a ticket with the given ID is in the collection.
public Ticket retrieve(String ticketID)
    // Get the Ticket with the given ID from the collection.
```

The idea is that, inside a Register object, there is an array in which Ticket objects are stored. Hence the instance variable “tickets”. The variable “numTickets” stores the index of the next free space in the array.

- When a Register object is created, an array of Ticket object is created and assigned to tickets, and numTickets is set to zero.
- When a Ticket is added, it is put in the next free space (the value of numTickets), and numTickets is incremented.
- When a ticket is retrieved, the ticket ID is used to find the relevant Ticket object in the array and return it i.e. “return tickets[i]” where i is the relevant index.

The array should be of length 100.

Here is a snippet of code to illustrate behaviour:

```
//...
Register r = new Register();
Ticket t = new Ticket(new Time("13:00"));
String ID_One = t.ID();
r.add(t);
t = new Ticket(new Time("13:18"));
String ID_Two = t.ID();
r.add(t);
System.out.println(r.contains(ID_One));
System.out.println(r.contains("9236743"));
System.out.println(r.retrieve(ID_Two).toString());
//...
```

The output from the fragment would be:

True

```
False
Ticket[id=800000005, time=13:18:00]
```

You can evaluate your work: by (i) writing a test program (something like that required for exercise 1), or (ii) by developing unit tests using Junit, or (iii) by using the jGrasp interactive feature.

NOTE: Clearly, using an array of length 100 as described is not the best solution for this class. When `numTickets==tickets.length`, the array is full. It's a simplification to make the exercise more accessible at this stage of the course. If you are confident of your Java skills you can try (a) a method for explicitly removing a Ticket from the array by inserting "null", and/or (b) dealing with the situation where the array fills up by making a bigger one, copying the contents of the old into it, and assigning it to `tickets`.

Submission

Submit the `SumCosts.java`, `TestOfTime.java`, `Ticket.java`, and `Register.java` source files to the automatic marker.

Appendices

Java Arrays

Arrays in Java bear a lot of similarity to Python Lists. Given an array, `A`, and an index value, `i`, a value, `v`, can be stored with the expression "`A[i]=v`". Similarly, a value can be retrieved, with the expression "`A[i]`" e.g. retrieving `v` and storing in a variable `n` is written "`n=A[i]`".

The differences are:

- An array stores a TYPE of value, which must be given when declared/described.
- An array is a type of object, and as such, is created using 'new'.
- An array has a fixed size which must be given when created.

Assume the following BMI class:

```
public class BMI {
    double height;
    int weight;

    double calculateBMI() {
        return weight/(height * height);
    }
}
```

Let's say we want an array that holds ten BMI objects. The following code snippet (i) declares a variable that can store an array of BMI, then (ii) creates such an array and assigns it to the variable:

```
//...
BMI[] records;
records = new BMI[10];
// ...
```

The variable declaration looks similar to others that we've used. The type is "`BMI[]`". It's the brackets that indicate the variable can store an array that stores BMI objects. Without the brackets, of course, it would just be a variable that can store a BMI object.

The creation expression is similar. The type of thing being created, “BMI [10]”, is an array that can store BMI objects, the size of the array is ten.

Initially the array does contain any BMI objects. (The value at each index is the special value ‘null’.) Extending the code snippet as follows, we create a BMI object and insert it at location zero:

```
//...
BMI[] records;
records = new BMI[10];

BMI bmi_record = new BMI();
bmi_record.height = 165;
bmi_record.weight = 57;
records[0] = bmi_record;

System.out.println(records[0].height);
System.out.println(records[0].weight);
System.out.println(records[0].calculateBMI());
//...
```

The snippet ends with print statements. Each accesses the BMI object at location zero, i.e. this is what the expression “records[0]” does, and then one of the object’s components. The first print accesses the height field, the second the weight field, and the third the calculateBMI() method.

For completeness, consider the following additional statements:

```
//...
System.out.println(records[0]);
System.out.println(records[1]);
//...
```

You might be inclined to think that the first statement prints out the BMI object stored at location zero, i.e. the height and weight. In fact, it prints something like the following:

```
BMI@7e6c04
```

The output consists of the name of the type of object (BMI) followed by an ‘@’ sign, followed by what’s called a “hashcode”, a kind of identity code, and which we won’t get into here. (If we had another BMI object and tried to print that we would generally get a different hashcode for it.)

The second print statement will output the following, since we haven’t stored a BMI object at that location:

```
null
```

Finally, given an array, A, we can obtain its length with the expression “A.length”.