CMP-4008 Programming I

Geoff McKeown - ***Lecture Notes Week 2, Semester 1***

Some More Basic Java. Introduction to Java Classes. Assignment statements. Arithmetic and Logical expressions

Lecture Objectives

- Another simple Java program.
- Introduction to Java class definitions.
- String concatenation in Java.
- Assignment statements and expressions.

A Java application using integer division and remainder operators

- When the division operator (`/`) is applied to a pair of `int`s in Java, the result is also of type `int` and is the integer part of the division. Thus
  - `14/5` gives the result `2`

  The remainder from an integer division may be obtained by using the `%` operator:
  - the expression `14 % 5` gives the result 4.

Example

Write and test a Java application with just a `main` method class called `DigitSum` that reads an integer between 0 and 1000 and adds all the digits in the integer. For example, for the integer 739 the result is 19.

```
package digitsum;

import java.util.Scanner;

/*
 * To read an integer between 0 and 1000 and add all the digits in the integer.
 * For example, for the integer 739 the result is 19.
 * Author gpm
 */
public class DigitSum {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter a three digit number: ");
        int number = scan.nextInt();

        int firstDigit = number % 10;
        number = number / 10;
        int secondDigit = number % 10;
        number = number / 10;
        int thirdDigit = number % 10;

        System.out.println("The sum of the digits of the "
            + "three digit number input is: " + (firstDigit + secondDigit
            + thirdDigit));
    }
}
```

Classes

- So far, each of the classes we have written has simply contained a `main` method.
- We shall see, we can write classes which contain more than one method.
- However, in Java we can also use the *class* construct to define a particular type of object.
- Such a class is a *blueprint* for a type of object:
  - an object of a particular class is created during the execution of a program whenever a special method of that class known as a *constructor* is executed, e.g.

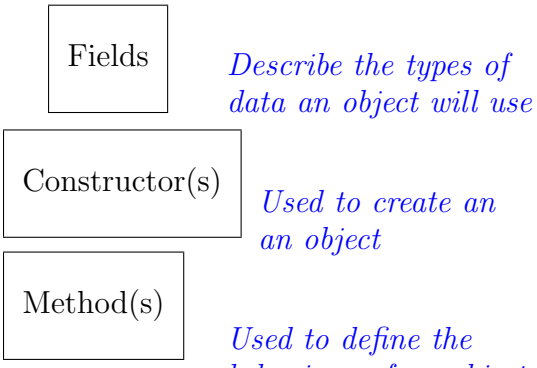
```
Scanner scan = new Scanner(System.in);
```

  - Many different objects of the same class type may be created during the execution of a program.

- Such classes also provide *methods* that can be invoked on any object of that class.

Structure of a Typical Class Defining a New Type of Object

`public class` ⟨*class name*⟩
{



}

- `public` and `class` are key words of the Java programming language.
- We will discuss the key word `public` (which is known as an *access modifier*) later in the module.
- The pair of curly braces, { and }, which are used to enclose the *body* of the class are essential.
- An object of a particular class is also referred to as an ***instance*** of that class
  - multiple instances can be created from a single class.
- The class defines what methods an object has.
- All instances of the same class have the same methods.

- A java class is saved in a file whose name is that of the class followed by the extension `.java`
  - e.g. a class called `Myclass` would be saved in a file called `Myclass.java`

Example: a simple model of a bank account

- We will present a class to represent a simple bank account.
- We will also present another class that enables us to test our bank account class.
- Both of these classes contain features that we will explain only partially at this stage.
- We will return to these classes in later lectures where everything will be explained more fully.
- We will develop a less naive bank account class in a later lecture.

- The definition of a class consists of a ***header*** followed by a ***body***.
- The body is enclosed between curly braces, { and }.
  - The body of a class defining a new type consists of Java code specifying the fields, constructor(s) and methods.

Class Header

We will call our class `SimpleBankAccount`, so our class header is:

```
public class SimpleBankAccount
```

Fields

Our class will have three fields:

- a string of characters representing the account name
  - we will denote this field by the name (or *identifier*) `accountName`
- an integer (i.e. a whole number) representing the amount of money (in pence) in the account
  - we will denote this field by the identifier `balance`
- an integer representing the value (in pence) of an agreed overdraft limit
  - we will denote this field by the identifier `overdraftLimit`

```
private String accountName;

// The two monetary fields, balance and overdraftLimit,
// are modelled as int instance variables.

private int balance;
private int overdraftLimit;
          // overdraftLimit should be non-negative.
```

Another form of comment

- `private` and `int` are more key words of the Java programming language.
- Like `public`, `private` is an access modifier and is discussed in a later lecture.
- `int` is Java's word for an integer; the (programming) statement

```
private int balance;
```

  defines the field `balance` to be of type `int`, i.e. a whole number.

- Although the word `String` is not a Java key word, it is the name of a class in the ***Java Standard Library*** (of which more in a later lecture), and should not be used other than in the way it is here.

```
private String accountName;
```

  defines the field `accountName` to be of type `String`, i.e. a character string.

  Recall that Java is a *case-sensitive language*, so `String` should always be typed with a capital `S`.

- In general, when an object is constructed, an *instance variable* is created corresponding to each field of the class. For example, if a `SimpleBankAccount` object called `myAccount` is constructed, then three instance variables

myAccount.accountName

myAccount.balance

myAccount.overdraftLimit

are created.

◇ The values of an object's set of instance variables is called its ***state***.

Constructor

◇ In order to create an object (an instance) of a given class, a program must invoke a constructor of that class.

◇ As part of our definition of `SimpleBankAccount` we must therefore define a constructor.

◇ A constructor consists of a ***header*** followed by a ***body***.

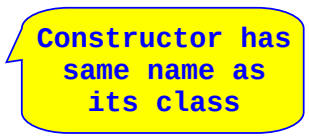
◇ A body consists of a sequence of statements enclosed between curly braces, { and }.

◇ A constructor has

- ▷ the same name as the class in which it is defined;
- ▷ a list of zero or more *parameters*, enclosed between round brackets, `(` and `)`.

◇ You have written a number of programs that use the methods of the `Scanner` class to read input from the keyboard.

◇ In order to use these methods, your programs had first to create a `Scanner` object:

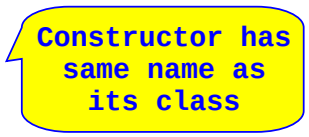



```
Scanner scan = new Scanner(System.in);
```

◇ A constructor for SimpleBankAccount:

```
/*
* A constructor for SimpleBankAccount objects.
* This constructor has a parameter for the
* account name and two monetary parameters.
*
* initialBalance represents the opening balance of this
* new SimpleBankAccount.
*
* agreedOverdraft represents the amount this
* SimpleBankAccount may become overdrawn before
* incurring penalties.
*/

public SimpleBankAccount( String name,
                  int initialBalance, int agreedOverdraft )
{
    accountName = name;
    balance = initialBalance;
    overdraftLimit = agreedOverdraft;
}
```



Methods

- The methods of a class define the possible behaviour of an object.
- If, in a running program, a `Scanner` object called `scan` has been created then the expression

```
scan.nextInt();
```

  causes the `nextInt()` method of the `Scanner` class to be applied to `scan`
  - the application of this method delivers as its result the value of the integer typed at the keyboard.
- Like the definition of a constructor, the definition of a method consists of a ***header*** followed by a ***body***.
- Again, the body consists of a sequence of statements enclosed between curly braces, { and }.
- The header for a method is similar to that for a constructor, except:
  - we can give it any meaningful name (not the name of the class);
  - as well as a (possibly empty) parameter list, a method has a ***return type***:
    - a method may return information about an object via a ***return value***;

Some methods for the `SimpleBankAccount` class

```
/*
* The accessor method getBalance() returns
* the current balance of this SimpleBankAccount.
*/
public int getBalance()
{
     return balance;
}
```

```
/*
* The accessor method getOverdraftLim() returns
* the overdraft limit of this SimpleBankAccount.
*/
public int getOverdraftLim()
{
    return overdraftLimit;
}


/*
* The mutator method deposit( int amount )
* adds amount to the balance of this SimpleBankAccount.
*/
public void deposit( int amount )
{
     balance += amount;  // adds the value of
                         // amount to balance
}

public String toString()
{
    return accountName;
}
```

- The code for the `SimpleBankAccount` class can be found in the NetBeans Projects folder on BlackBoard.
- The NetBeans project containing this class also contains a class called `SimpleBankAccountDriver`.
- When the main method in `SimpleBankAccountDriver` is executed, code in the `SimpleBankAccount` class is executed.

```
public class SimpleBankAccountDriver {

    public static void main( String [ ] args )
    {
        String name = "Geoff";
        int initialBalance;
        int overDlimit;

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter a value (a whole number)"
                           + " for the opening balance: " );
        initBal = scan.nextInt();
        System.out.println("Enter a value (a whole number) "
                           + "for the agreed overdraft limit: " );
        overDlimit = scan.nextInt();

        // Create a SimpleBankAccount object called myAccount
        SimpleBankAccount myAccount =
        new SimpleBankAccount(name, initialBalance, overDlimit);

        System.out.println("The current balance of account " + myAccount +
                             " is " + myAccount.getBalance() );
        System.out.println( "Finished" );
    }

}
```

String concatenation

- The `System.out` object is built into the Java language.
- It represents an output device or file
  - by default, this is the monitor screen.
- The `print` and `println` methods of `System.out` object enables us to print messages:

```
System.out.println("someText");
```

  simply prints the string enclosed in double quotes and ends the line.
- The difference between the `System.out.print()` and `System.out.println()` methods is that `System.out.println()` always ends the line being printed.
  - If you want to leave a blank line in your printout, this can be achieved by the statement

```
System.out.println();
```

- Note that `"someText"` is the *actual parameter* in the first invocation above of the `println` method.
- `System.out.print` and `System.out.println` always take a single parameter: a string of characters to be printed.
- However, such a string can be created in quite complicated ways.
- Consider

```
"Man." + " " + "Utd."
```

  This is equal to

```
"Man.  Utd."
```

  So, the "+" signs here denote *string concatenation*.

- A more complicated example:

```
System.out.println("The current balance of account "
    + myAccount + " is "  myAccount.getBalance() );
```

  - Because we included a `toString` method in the class `SimpleBankAccount`, when the name of an object of this type is encountered within the parentheses of an application of `System.out.println`, the `toString` method is automatically applied to the `SimpleBankAccount` object (in this case, `myAccount`) to deliver a `String` (`"Geoff"` for the above example).
  - Although the result of `myAccount.getBalance()` is an `int`, this `int` is automatically converted into its corresponding `String`.

- A `String` literal cannot span more than one line in your program text. If you cannot fit the entire `String` on one line, you must split it into smaller Strings to be linked by the concatenation operator:

```
// This will not compile
System.out.println("Perhaps surprisingly, I remain an optimist
      in matters of conveying understanding. "   );

// But this will
System.out.println("Perhaps surprisingly, I remain an optimist"
   +  "  in matters of conveying understanding. "   );
```

Assignment Statements

- We have seen both of the following statements in programs given previously:
  - `firstDigit = number % 10;`
  - `number = scan.nextInt();`
- These are both examples of *assignment statements*.
- The general form of an assignment statement is:

⟨*variable name*⟩ = ⟨*expression*⟩;

- When an assignment statement is executed, the expression on the right-hand-side is first evaluated and the result is then "assigned" to the variable identified on the left-hand-side of the statement:
  - when a value is assigned to a variable, any previously assigned value is lost.
- Note that in Java we use the `=` symbol for assignment.
- All variables must be *declared* before they can be assigned values, e.g.

  ```
  int firstDigit;
  ```

  declares a variable identified by the name `firstDigit` to be of type `int`:
  - this statement instructs the compiler to set aside space in main memory capable of storing any value of type `int`;
  - when a value is assigned to a variable, that value is stored in the corresponding memory.
- In Java, the type of the result from evaluating an expression may be any of the primitive types (`int`, `double`, `boolean`, etc.) or a *reference type*
  - we will discuss the latter in a later lecture.

Operator Precedence

- Consider the following assignment statement:

  ```
  r = p + q * 5;
  ```

- The expression on the right-hand-side of this statement is an example of an *arithmetic expression*.
- `+` and `*` are *arithmetic operators*, and `p`, `q` and `5` are *operands*
  - 5 is an example of a `literal` operand.
- The other arithmetic operators are `-`, `/` and, for int operands only, the remainder operator `%`.
- In Java, as in mathematics, there are operator *precedence rules*: an expression is evaluated from left to right subject to these precedence rules.
- `*`, `/` and `%` have equal precedence which is greater than that of `+` and `-`. The latter two operators have equal precedence.
- So, in the following expression the value of `q` will be multiplied by `5` and the result of this operation added to the value of `p`

  ```
  p + q * 5
  ```

◇ If we want to add the value of q to that of p *before* doing the multiplication, we must include parentheses and write:

```
(p + q) * 5
```

Logical expressions

◇ Consider the following expression:

```
value < limit
```

◇ This is an example of a *logical expression*

◇ When evaluated, a logical expression yields one of the two (*logical*, or *boolean*) values *true*, *false*.

◇ In Java, the primitive type boolean comprises the two (literal) values `true` and `false`.

◇ As is the case for arithmetic expressions, a logical expression consists of operands and operators.

◇ We begin by considering relational operators.

Relational operators

◇ Java's relational operators (e.g. `<`) are *binary operators*:

▷ so, like arithmetic operators such as `+`, `*`, etc. they take two operands.

◇ Both operands have numeric type, e.g. `int`, and the result type is `boolean`.

◇ The full list of relational operators is:

`<` `<=` `>` `>=`.

Equality operators

◇ There are two binary equality operators:

`==` ("equals") and `!=` ("not-equal").

◇ Each of these operators can be applied to a pair of operands of the *same* primitive type, and each produces a `boolean` result value.

◇ Let `a` have value `2`, `b` have value `3` and `c` have value `5`

`a <= b`	has value `true`
`(a+b) == c`	has value `true`
`(c-b) != a`	has value `false`
`b > c`	has value `false`

Logical (Boolean) Operators

◇ The boolean type has its own operators, the *logical* operators.

◇ Each of these operators takes `boolean` operand(s), and delivers a `boolean` result.

◇ Boolean operators can be used to combine the results of simpler tests.

◇ The most commonly used Boolean operators are defined as follows:

Notation	*Operation*	*Result*
`!B`	`NOT` (unary)	`true` if `B` is `false`, and `false` if `B` is `true`
`B0 && B1`	`AND` (binary)	`true` if `B0` and `B1` are both `true`, and `false` otherwise
`B0 || B1`	`OR` (binary)	`false` if `B0` and `B1` are both `false`, and `true` otherwise

Logical and relational operator precedence

◇ As with arithmetic operators, there are precedence rules that determine the order of evaluation of expressions containing logical and relational operators.

◇ Sub-expressions within parentheses are evaluated first.

◇ Operations involving operators of equal precedence are evaluated left to right.

◇ The relative precedence of the different operators is given in the following list; the higher in the list, the greater the precedence.

(i) `!`

(ii) `<, <=, >, >=`

(iii) `==` and `!=`

(iv) `&&`

(v) `||`

Let `a = 2, b = 3, c = 5`

`!(a <= b)`	has value `false`
`(a <= b) && (b > c)`	has value `false`
`!(a > b) && (b < c)`	has value `true`
`(a < b) || (b > c)`	has value `true`