Introduction to Programming

8. Methods – Part 2

Methods as named actions

- In the first part of this lecture we saw that we can use a method to give a name to an action such as drawing a circle or playing a game
- To perform the action, we *call* the method
- We can pass over any information that the method needs to perform the action using its *parameters*
  - E.g. where to locate the centre of the circle and what radius to give it

Functions

- We also saw that some methods return a result
  - Such methods are called *functions*
  - The function declaration includes a *return-type*, which states what type of value it returns
  - The function body includes a *return statement*, used to return the value

return-type

```
// Returns the square of the number provided
public static double sqr(double number) {
    return number*number;
}
```

return statement (double)

Static methods

- If all the information that a method needs to perform an action is passed to it using its parameters, and it does not need to remember any information from previous calls, you should declare the method as **`static`**
  - The function, sqr(), on the previous slide is an example, the only thing it needs to know to perform its calculation is the value of number

Static variables

- Just as a method can be declared as static, so that it belongs to the class that declares it, so can a variable
- Unlike local variables, static variables are automatically initialised when they are declared
  - To zero if a number, to **`false`** if a **`boolean`**, or to **`null`** if a reference variable such as a String
- Static methods can access static variables declared in the same class without the variable being passed as a parameter
- Consider GuessingGame2 example in section 4.2.4 of the book...

GuessingGame2

- From the textbook (pages 143-144):
  - *As an example, let's add a static member variable to the GuessingGame class that we wrote earlier in this section. This variable will be used to keep track of how many games the user wins. We'll call the variable gamesWon and declare it with the statement "static int gamesWon;". In the playGame() routine, we add 1 to gamesWon if the user wins the game. At the end of the main() routine, we print out the value of gamesWon. It would be impossible to do the same thing with a local variable, since we need access to the same variable from both subroutines.*

GuessingGame2 class

```
public class GuessingGame2 {
  static int gamesWon; // The number of games won
                       // by the user.
  public static void main(String[] args) {
    gamesWon = 0; // This is actually redundant,
                  // since 0 is the default
                  // initial value.
    TextIO.putln("Let’s play a game. I’ll pick" +
                 " a number between 1 and 100");
    TextIO.putln("and you try to guess it.");
    boolean playAgain;
```

main() method continued

```
    do {
      playGame(); // call subroutine to play one
                  // game
      TextIO.put("Would you like to play again? ");
      playAgain = TextIO.getlnBoolean();
    } while (playAgain);
    TextIO.putln();
    TextIO.putln("You won " + gamesWon +
                 " games.");
    TextIO.putln("Thanks for playing. Goodbye.");
  } // end of main()
```

The playGame() method

- The local variables and initialisations are exactly as before

```
static void playGame() {
  int computersNumber; // A random number picked by the
                       // computer.
  int usersGuess; // A number entered by user as a guess.
  int guessCount; // Number of guesses the user has made.
  computersNumber = (int)(100 * Math.random()) + 1;
  // The value assigned to computersNumber is a randomly
  // chosen integer between 1 and 100, inclusive.
  guessCount = 0;
```

playGame() updates gamesWon

```
TextIO.putln();
TextIO.put("What is your first guess? ");
while (true) {
  usersGuess = TextIO.getInt(); // Get the user’s guess.
  guessCount++;
  if (usersGuess == computersNumber) {
    TextIO.putln("You got it in " + guessCount
     + " guesses! My number was " + computersNumber);
    gamesWon++; // Count this game by incrementing gamesWon.
    break; // The game is over; the user has won.
  }
  // continued on next slide, still in the loop
```

Rest of playGame() as before...

```
    if (guessCount == 6) {
       TextIO.putln("You didn't get the number in 6 guesses.");
       TextIO.putln("You lose. My number was " + computersNumber);
       break; // The game is over; the user has lost.
    }
    // If we get to this point, the game continues.
    // Tell the user if the guess was too high or too low.
    if (usersGuess < computersNumber)
       TextIO.put("That's too low. Try again: ");
    else if (usersGuess > computersNumber)
       TextIO.put("That's too high. Try again: ");
    }
    TextIO.putln();
    } // end of playGame()
} // end of GuessingGame2
```

Comments on GuessingGame2

- Declaring the static variable *gamesWon* allowed the two static methods (main() and playGame()) to share and update the same variable
- Remember that in Java a subroutine gets a *copy* of each argument passed as a parameter, so passing *gamesWon* as a parameter to playGame() would not have worked as the increment only changes the copy, not the actual parameter

Side-effects

- In main(), *gamesWon* is accessed just twice:

```
gamesWon = 0;
...  // code here includes a call to playGame()
TextIO.putln("You won " + gamesWon + " games.");
```

- Someone just looking at the code for main() might expect the value of *gamesWon* to be zero when the value is displayed, as main() does not change it
- But the value may have changed as a *side-effect* of the call to playGame() in the loop in main()

Side-effects continued

- *gamesWon* is an example of what, in other programming languages, would be called a *global variable*
  - it is visible everywhere in the program, and its value can be read or changed from anywhere in the program
- In structured programming:
  - using global variables to pass information between subroutines (i.e. without using parameters and return values) is considered bad practice, and relying on side-effects to update such variables means that the whole of the program has to be examined to determine where variables are used and their values changed.
  - variables should be declared to be as local as possible and parameters should be used to pass data between methods

Avoiding the global variable in GuessingGame2

- How could we change GuessingGame2 to make *gamesWon* a local variable?
- Change playGame() to take the number of games won as a parameter and to return its new value, and declare the gamesWon variable locally in main()

Revised playGame() method

```
static int playGame(int gamesWon) {
   ... // code as before except
   while (true) {
      ...
      if (usersGuess == computersNumber) {
        TextIO.putln("You got it in " + guessCount
        + " guesses! My number was " + computersNumber);
       gamesWon++;      // only affects the local copy
       return gamesWon; // so we return the new value
      if (guessCount == 6) {
        TextIO.putln("You lose. My number was " +
                         computersNumber);
        return gamesWon; // return the old value
      }
```

Revised main() method

```
public static void main(String[] args) {
   int gamesWon = 0; // now local, has to be initialised
   TextIO.putln("Let’s play a game. I’ll pick" +
                " a number between 1 and 100");
   TextIO.putln("and you try to guess it.");
   boolean playAgain;
   do {
      gamesWon = playGame(gamesWon);
      TextIO.put("Would you like to play again? ");
      playAgain = TextIO.getlnBoolean();
 } while (playAgain);
 TextIO.putln("You won " + gamesWon + " games.");
 TextIO.putln("Thanks for playing. Goodbye.");
} // end of main()
```

Private static variables

- Sometimes it is almost impossible to avoid using a non-local variable
  - Typically, when a subroutine needs to remember information from a previous call and the information cannot be passed to it as a parameter (e.g. because the caller should not be given access to the information and so cannot pass it over)
- In such cases, you can limit the visibility of the static variable by declaring it as private, so that only methods in the same class can access it, and place the code that calls the subroutine in a different class (so the calling code cannot see the variable, which is then not “global”).

The subroutine **contract** (see Eck section 4.6.1)

- The specification of a subroutine states what the effect of calling it is (this is called its *postcondition*) given its assumptions about the values of its parameters (or other variables) at the time it is called (these assumptions are called its *precondition*)
- Forms a ***contract***
  - Caller guarantees the precondition will be true whenever the subroutine is called
  - Implementer guarantees that if the precondition is true when the subroutine is called that the postcondition will be true when the subroutine returns
    - If precondition is not true when the subroutine is called the contract does not specify what, if anything, the subroutine will do!

Pre- and post-conditions

```
/*
* Precondition:
*   Integer.MIN_VALUE <= first – second <=
*                               Integer.MAX_VALUE
* Postcondition:
*   Compares its two arguments for order. Returns a
*   negative integer, zero, or a positive integer as
*   the first argument is less than, equal to, or
*   greater than the second argument.
*/
static int compare(int first, int second) {
  return first - second;
}
```

The subroutine contract 2

- Try and avoid unnecessary preconditions when designing your subroutines
  - The compare method on the previous slide only needs the precondition because of how it calculates its result – the method will not work if *first* and *second* have opposite signs and have large absolute values (in that case, their difference can be outside the range of **int** values)

Avoiding the Precondition

```
/* Compares its two arguments for order. Returns
   a negative integer, zero, or a positive integer
   as the first argument is less than, equal to,
   or greater than the second. */
static int compare(int first, int second) {
        if (first < second) return -1;
        else if (first == second) return 0;
        else return 1;
   }
}
```

Writing subroutines (homework!)

- Study the guessing games example program in sections 4.2.3 and 4.2.4, and the 3N+1 example in 4.4.3
- Try exercises 1 to 4 of chapter 4

Questions??