Lecture 4: Programming in the Large I

Instructor: Musard Balliu, `musard@chalmers.se`

`http://www.cse.chalmers.se/~musard`

QUESTIONS?

Plan

- Last time
  1. conditionals: if and switch
  2. loops: for and while

- Today's Plan:

  1. subroutines

  2. arrays

Minimum of three numbers

Let us revisit the problem of computing the minimum of three numbers.

We can start with the observation that the minimum of `n1, n2, n3` is the minimum of `n1` and the minimum of `n2` and `n3`.

This leads to the following code:

New Min3 program

The problem is that we have basically copied the code from the `Min2.java` twice.

Ordering three numbers

Similarly, let us revisit the problem of ordering three numbers `n1, n2, n3` in ascending order.

We observe that the first number must be the minimum of `n1, n2, n3`. We compute that minimum and swap it with `n1`.

Now we know that `n1` contains the minimum, so the middle number should be the minimum of `n2` and `n3`. We compute that and swap it with `n2`. After that, `n3` must be the largest number of the three.

This leads to the following code: New Order3 program

Now we have copied the code from `Min2` and `Swap1` three times!

Integer expressions

Still, with the means at our disposal, we cannot do better. We must change the values of the numbers, and we can only do that by means of assignments of the type

```
min = intExpression
```

The `intExpression` can be something like `n1 + n2` or `n1 % n2`, but we have no `intExpression` that gives us the minimum of `n1` and `n2`.

Subroutines

With subroutines, we can extend the set of allowed expressions.

For example, just as we have an expression that gives us the sum `n1 + n2`, we can introduce an expression `min(n1, n2)` which has as value the minimum of `n1` and `n2`.

Subroutines

The general form of a subroutine is

```
typeS name(type1 arg1, type2 arg2, ..., typeN argN) {
        statements
}
```

Magic: We always have to prefix the whole by `static` (will be explained next time).

This extends the expressions of type `typeS` with a new one, namely

```
    name(x1, x2, ..., xN)
```

The subroutine **must** end with a statement of the form

```
    return x;
```

where `x` has type `typeS`. (There is one exception we'll see later.)

Subroutines

The semantics of

```
x = name(x1, x2, ..., xN)
```

is:

```
{ // A new block is started!
    type1 arg1 = x1;
    type2 arg2 = x2;
    // ...
    typeN argN = xN;

    statements; // subroutine block statements
} // The new block has finished
  // arg1, ..., argN no longer in scope!
x = returned value // (last statement in block)
```

Subroutines

The number of arguments of a subroutine, `N`, can be 0. For example, a subroutine to model a dice roll would have the signature

```
int roll()
```

Additionally, we can have subroutines that return "nothing". For example, a subroutine that prints a given message on the screen (like `println`). In this case, the return type is `void`, and the last statement can be something other than `return` (a `return` by itself is also a legal way to end such a subroutine).

Subroutine exercises

Write subroutines to

1. calculate the minimum of two numbers
2. calculate the minimum of three numbers
3. calculate the maximum of two numbers
4. calculate the maximum of three numbers
5. calculate the middle of three numbers

Initial code
Solution

An impossible exercise

Write a subroutine which swaps the values of its arguments, i.e.:

```
void swap(int x, int y){
}

// assume x = X and y = Y
swap(x, y);
// x = Y and y = X
```

This cannot be done, because inside the subroutine we only have access to the values of `x` and `y`, not the names themselves.

Arrays

Arrays do for data what loops do for code: they make it possible for us to work with huge quantities of data without having to address each piece individually.

Arrays allow us to introduce a number of variables *of the same type* in one go, corresponding to the common mathematical usage "let $x_0, x_1, \ldots, x_{n-1}$ be n variables of type `type`". (Note that the numbering starts with zero).

The type of the variables *must* be the same. The n variables are also called *array elements*. n is referred to as *the size of the array*.

Arrays

The general form of an array declaration is

```
type[] arrayName;
```

This is read as "`arrayName` will be used to refer to several variables $\texttt{arrayName}_0$, $\texttt{arrayName}_1$, ..., $\texttt{arrayName}_{n-1}$, all of type `type`".

In Java, instead of $\texttt{arrayName}_k$, we write `arrayName[k]`. This is known as *indexing*.

Initializing arrays

Before initializing an array, its size *n* must be known and enough space must be allocated to store the values of the variables. The general form of this instruction is:

```
arrayName = new int[n];
```

The array elements can then be initialized individually, by assignment. For example:

```
for(int i = 0; i < n; i++)
  arrayName[i] = expression;
```

is a common way of initializing the array elements.

Array initialization

Example:

Create an array `x` of 100 elements, such that `x[i] = i * i`.

Initial code

Solution

Array initialization

If the array is small and the values of the array variables are known, then the array can be initialized by enumeration at the same time with the declaration. For example:

```
char[] grades = {'A', 'B', 'C', 'D', 'E'};
```

This line declares the variable grade of type char[] (array of characters) and initializes the array elements with the given characters, in the order given.

Values lists can also be used outside declarations, but in conjunction with memory allocation:

```
char[] grade;

// grade = {'A', 'B', 'C', 'D', 'E'}; Error!

grade = new char[] {'A', 'B', 'C', 'D', 'E'}; // OK
```

`char[]` versus `String`

In some languages, `char[]` and `String` are synonymous or at least closely related. In Java, that is not the case. In particular, you cannot index a string, or use `.length` to find out its length.

The main operation on strings is *concatenation*, which we have used many times already in printing messages to the screen. The important thing to know is that “adding” a string to something, e.g. an integer, will result in the concatenation of that string with a textual representation of that something. Thus

`"x = " + 4` evaluates to the string `"x = 4"`
`"x = " + 2 + 2` evaluates to `"x = 22"`
`"x = " + (2 + 2)` evaluates to `"x = 4"`.

Array exercises

1. By default, printing an array will just print a less-than-helpful representation. Write a subroutine to convert an array of integers to a string representation, with the values of the array elements appearing between square brackets, separated by commas.

   Initial code
   Solution

2. Write a subroutine to compute the minimum of an array.

   Initial code
   Solution

Exercises

Make sure you understand and can compile and run all the examples and exercises from this lecture.

Read sections 4.1, 4.3, and 4.4 from Eck's book.