COMP6700/2140 Names and Values

Alexei B Khorev and Joshua Milthorpe

Research School of Computer Science, ANU

February 2017

Topics

1. Names as Symbols of Data
2. Declaration of Names
3. Initialisation
4. `final` variables and (Im-)Mutable Objects

Declaration

Variables of `char`, `int`, `double`, *String* and an array type, discussed above, are just names which determine storage locations, "boxes" in which one can "place" a value of the right type (can fill in), then swapped with another similar typed value and so on (assign and reassign a value of the correct type). Variables can appear as: *class fields*, *local variables* inside a block of code, and *method parameters*.

Variables are introduced as program's entities through *declaration*:

```
[@annotation-type] [modifier]* type identifier
```

where (let's ignore `@annotation`) `modifiers` are optional and possibly a few (like `private static final`), `type` is necessary (it determines the *size* of allocated memory and the *behaviour* of declared entity), and `identifier` is the variable name. The declaration is often accompanied by initialisation (assigning a value for the first time), or the initialisation can be performed later in the code. The variables of the same type can be declared in a comma separated list, but the variables of different type must be declared in separate statements.

```
float x, y; // declared but not initialised
int z, u = 2; // two declared, one initialised
```

Initialisation

A variables **must** be initialised before it can be used (*except for formal method parameters*). Once a variable is initialised, its value can be used in expressions or assignments.

```
int x; //not initialised, cannot be used
int y = 10;
x = y * y; // now x has a value
int z = x; // now x can be used and given a new value
```

Field variables are declared inside the class definition. *Non-static fields* exist during the life of an object to which they belong. If not initialised explicitly, they are given the *default values* when the object is created.

```
class MyClass {
   int x; // default value 0
   static double y = 10.0;
}
```

Non-field variables

Local, or block, variables are declared anywhere inside a block (method's body is an example); unlike field variables, local variables must be initialised explicitly before they can be used; they cease to exist when the flow of control (the computation path which the program execution goes through) reaches the end of the block or method body in which they were declared. Local variables can have only one modifier, `final`.

Parameter variables are the parameters declared in *methods*, *constructors* or `catch` clauses of the `try-catch-finally` blocks. A parameter declaration includes an optional modifier (an `@annotation` or `final`), type and an identifier (parameter's name). During a method declaration or its invocation in the bodies of *other* methods, the method variable is a *formal parameter* (it has no value), but during the actual invocation of the method, it is an *actual parameter* (it must have value). Parameter variables cease to exist when their method completes (by reaching its logical end, or terminating with error).

Constant variables

An initialised variable can be thought as an association between a name and a value. This association can change — a variable can be assigned another value during the lifetime **unless it was declared** `final`. `final` variables are constants:

- constant as *values* for primitive type variables,
- constant as *reference* for reference type variables,
- constant as *values* for immutable reference type variables.

An attempt to change the value of a `final` variable will result in a compile error:

```
final int x; // if not initialised during the declaration,
             // the final variable is a blank constant
x = 2; // ok, now the constant x is given a value
x = 4; // can't do!
```

`final` variables are normally initialised immediately after declaration. One can postpone the assignment, but modern IDEs issue a warning in such case. Such *blank* `final`s are mostly used if the field value is determined by the constructor arguments (example next slide).

Immutable of objects

`final` variables of a reference type are not to be confused with references to *immutable objects*: the former cannot be reassigned, but the object to which the variable is assigned to, can change its state if it is mutable. An immutable object (like string) cannot change their state, but a non-`final` variable assigned to this object can be reassigned to another object (of the same type).

*String*s are example of *immutable* objects — manipulations with their content involve creation of a new *String* object with modified value (like `substring()`).

```
class Student {
    final String name; //below is the Student class constructor
    Student(String name) { this.name = name; }
}
```

Where to look for this topic in the textbook?

Hortsmann's *Core Java for the Impatient*, Ch. 1.3, 1.5