COMP6700/2140 Classes

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Class members
2. Access and scope
3. Constructors
4. Creation of objects

What's inside a class?

In Java, objects are created in accordance with a template which determines their members and the very process of creation. This template is defined via **class declaration**. The class definition includes:

1. Class declaration — access, scope, *name*, superclass name, implemented interfaces — followed by the class body which consists of:
2. zero, one or many field declarations (an instance of a class without fields is a stateless object, eg. `java.lang.Object`)
3. zero, one or many method declarations (an instance of a class without methods is possible, but very interesting; when they are required, better to use `enam` types — we'll learn this later)
4. zero, one or many so-called *constructors*, which are like methods but strictly speaking are not — instead of returning a value, they build and return a *reference to an object* of this class; **constructors have the same name as their class**

If you have a reference variable `o` (or a literal *object*, *eg* a string), the operator `.` (*dot*) applied to `o` (or a literal object) gives an access to the object internal resources (object's members), if such access is allowed: `o.f` evaluates the field `f`, `o.m()` invokes the method `m()`.

Constructors are needed to control the creation of an object; when we need to declare and create an object (also called an *instance* of the class), we have to use the operator `new` "applied" to the class constructor:

```
MyType myObject = new MyType();
```

Syntax of class declaration

Class name defines type of its instances; class members (methods + fields) define their properties (what can be done with them). In other words — *Object-oriented language provides the programmer with possibility to define their own type*.

The type definition is achieved by *class declaration*

Formal ("dry") syntax rules:

```
[<modifier>]* class ClassName [extends ParentClass] [implements AnInterface]* {
   [<access-modifier>] [<scope-modifier>] returnType fieldName [ = initValue];
    ...  repeated as many times as there are fields ...

   [<access-modifier>] [<scope-modifier>] returType methodName([arguments]) {
       body of a method;
   }

   [<access-modifier>] [<scope-modifier>] ClassName(...) {
     ... body of a constructor ...
   }
  ...  more methods, constructors and inner classes ...
}
```

can be made more vivid in an example: let's declare the class *Planet* and create a few planet objects (and make them move ☺).

Class *Planet*

The class *Planet* describes a celestial body:

```
class Planet {
  public long idNum; public String name; // two declaration on one line
  public Planet orbits; // not all fields are nouns
  public long id; // the catalog number of the new planet
  private static long nextID = 0; //the total planet counter

  public Planet(String s) { //one constructor
    this.name = s;
    id = nextID++; // here the post-increment matter!
  }
  public static long getNumberOfPlanets() { return nextID; }

  public Planet(String s, Planet p) {  //another constructor
    this.name = s;
    this.orbits = p;
    id = nextID++;
  }
}
Planet moon; // variable declaration
Planet sun = new Planet("Sun"); // declaration and instantiation
Planet venus = new Planet("Venus", sun); // another instantiation
```

Constructors and methods

A class can provide *several* ways to create its objects. If a class declaration include more than one constructor, and they all must have the same name, the only way to discriminate them is via the type and number of parameters each is declared with (and by exceptions they throw).

When no constructors are defined, the default *parameterless* constructor is available (it's inherited from parent class). If at least one constructor is defined, the default constructor is no longer available by default and must be defined explicitly (if needed).

```
class Student {
    String name;
    Degree degree;
    ArrayList<Course> courses = new ArrayList<Course>();

    public Student(String n) {…}
    public Student(String n, Degree d) {…}

    public void enrollCourse(Course c) { courses.add(c); }
    public void study() {…}
    public Mark sitExam(Course c) {…}
     ... ...
}
```

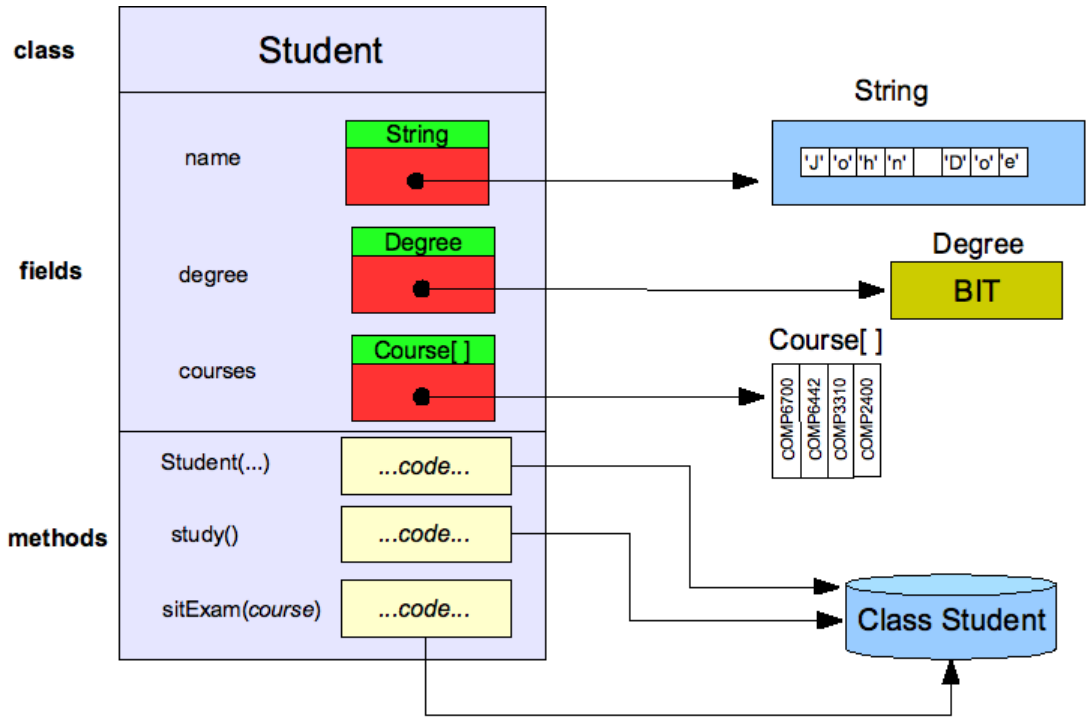
Objects born and lived

Once classes are defined, one can declare, create and manage objects of those classes:

```
Course comp6700 = new Course("comp6700"); // creating a new course
Degree bit = new Degree("MITS"); // new degree
Student stud = new Student("John Doe", bit); // start a degree
stud.enrollCourse(comp6700); // enrolling in a course
stud.study(); // "studying hard"
comp6700.setMark(stud,stud.sitExam(comp6700)); // earning a mark
...
stud.receiveGrades(comp6700);
...
stud.completeDegree();
```

"Student in memory"

It is useful to visualise an object memory layout:



object diagram

Objects in memory: Fields and Methods

With every constructor call, a memory block structured in accordance with the class definition is allocated, as illustrated by *the object diagram*. The fields of a newly created object are initialised first to their defaults, and *then* (after the parent constructor executes) to the values in assignment statements *outside the constructors*, and then the fields are assigned values as defined by the constructor.

The memory layout for (non-static) fields and methods is different: the object methods aren't grouped with the fields in memory (that would be wasteful). Memory is allocated for the instance fields of each new object, but there's no need to allocate memory for methods. All an instance needs is access to its methods, and all instances of the same class share access to the same set of methods. There's only one copy of the methods in memory, no matter how many instances of the class are created. ***However***, whether access to the methods requires a reference to an *object*, or the methods can be called by the *class* name — this depends on the scope identifier in the methods declaration:

- Static *methods* can be called without even a **single** instance of the class in existence
- Static *fields* can be read (and reset, if their are not final) without a single object of the class being created. There is a **unique copy** of a static field — one per class; for multiple instances, the value of their static field is the same
- Examples: `String.format())`, `Math.PI`, `System.out`
- If a method makes use of a static member, it itself must be declared `static` ("why?")

Member access

Fields and methods can be accessed by referencing the *object name*, or *class name* for `static` members:

```
System.out.println("The name of this planet is " + sun.name);
System.out.println("The total number of planets is " + Planet.nextID);
```

The access modifiers regulate how and which class member is visible — can be read, (re-)assigned or called (for methods):

1. `public` — visible from anywhere outside
2. `"friend"` (implicit, assumed if no access modifier is used; **no such keyword**) — visible within the same package
3. `protected` — visible only to objects which are instances of subclasses of the class
4. `private` — totally invisible from outside, can be only used inside the class; *Different objects of the same class* **can access** the other's private members, see Shtuka.java

The general policy is to declare fields *private* and (non-auxiliary) methods *public*, but it really depends on design considerations. The value of a private field is *read* by calling the *"getter"* method, and its value is *changed* with a *"setter"* method (not every class provides setter methods, immutable classes data, like in *String*, are not meant to be altered).

Member scope

Class members can have either an *object scope*, or a *class scope*, when they are declared *static*. Non-static members are associated with individual instances of the class (objects), and two values of the same field but from two different objects will have uncorrelated values:

```
public class Parameter {
   private int x; // declared static in StaticParameter class
   public Parameter(int x) { this.x = x; }

   public int getX() { return x; } // allows outside reading private field
   public void setX(int y) { this.x = y; } // allows changing it

   public String toString() { return "" + x; }
}
Parameter p1 = new Parameter(1);
Parameter p2 = new Parameter(1);
System.out.println("p1 has " + p1.getX() +
      " and p2 has " + p2.getX()); // prints "... 1 ... and ... 1 ..."
p1.setX(10);
System.out.println("p1 has " + p1.getX() +
   " and p2 has " + p2.getX()); // prints "... 10 ... and ... 1 ..."
```

The modified *Parameter* class (in which `static` modifiers are used) (the code *StaticParameter*) will produce *different* output (study!).

Member variability

The class members can be also controlled by presence of the modifier `final`.

- `final` field makes the value unchangeable, constant; its initialisation can be delayed (so called *blank finals*), but once initialised, further attempt to change its — even to the same value! — value will result in a compile time error:

```
final int i = 0;
i = 1;  // Error: This object cannot be modified
```

- `final` method *cannot be overridden* in a subclass (when inheritance is used, see **)3**)

Normally, all modifiers which can be used in a method declaration, can be also used in a class declaration, with the similar meaning. So far unmentioned modifiers include:

- `abstract` (for classes and methods) — will learn in **O3** and **O4**
- `native` (for methods) — indicates non-Java implementation
- `strctfp` (for methods) — used to indicate strict floating point arithmetic (not discussed)
- `annotation` (for classes and methods) — will be mentioned in a later week (time permitting)

Some modifiers cannot be used together, *eg* `final` and `abstract` together "do not compute".

Method overloading

Taste of OO power

The method `println()`, which we have used many times already, seems was one and the same despite the arguments were different (values of all types which we wanted to be printed). The sameness of the `println()` was an illusion — those were, in fact, *different methods!* Different by the type (and number) of parameter which one can pass to them. But they all can have the same name! A method whose name used in other methods which differ by the type and the number of their parameters is said to have been *overloaded*:

```
public class DataArtist {
    ...
    public void draw(String s) { ... }
    public void draw(int i) { ... }
    public void draw(double f) { ... }
    ...
}
```

It is illegal to overload a method name with the same number and type of arguments, because the compiler cannot tell them apart. Also, the compiler does not consider return type when differentiating methods, and thus one cannot declare two methods with the same parameters but with different return types. General advice — not overuse the overloading, since it precludes readability. Especially, do not overload methods with the same number (but different type) of parameters, including *varargs* (**J9**).

Constructors, defaults, `this` self-reference

The constructor definitions can contain any code, but their primary purpose is to initialise the class fields. Fields which are not initialised by the constructor *explicitly*, are given *default values*:

- primitive number types default to 0 (floating types to `+0.0`)
- a boolean type defaults to `false`
- a character type defaults to `'\u0000'`
- a reference type defaults to `null`

`static` fields can sometimes be assigned in the constructor:

```
public Planet(String s, Planet p) {
    this.name = s; // the self-reference this is optional
    this.orbits = p; // it's a good practice to avoid errors
    idNum = nextID++; // counting number of created objects
}
```

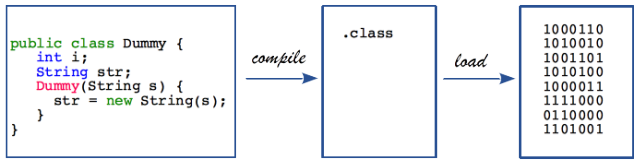
`this` is used for object self-referencing (*eg*, it can be passed as a parameter).

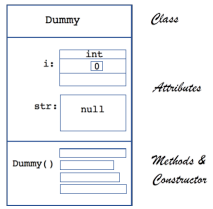
```
public ArrayList<Planet> satellites; // ArrayList is like an array
public Planet(String s, Planet p) {
    this.name = s; this.orbits = p; // apologies for two statements here
    p.sattellites.add(this);
}
```

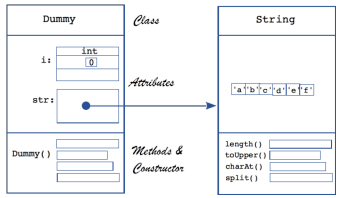
Object creation

Three stages — when class **is loaded**, then when its constructor is called but **before** it's executed, and finally **after** the constructor executed:

class compiled and loaded (verification etc)	before constructor executes	after constructor executes







Notice the appearance of an additional *String* object at the second stage due to presence of `new` operator inside the *Dummy* constructor which results in a creation of *new* String object. If the assignment of *str* value was different: `str = "abcdefg";` — then no new *String* object was created, but the reference *str* was given a value of the reference to the constant string "abcdefg" (remember *String*'s pool).

Static initialisers

Constructors perform object initialisation. A class also allow to initialise its (static) fields, but there is no special constructor-like declaration — simple include the initialisation statements in a block (or blocks) preceded by the `static` keyword `static { ... }`. The location and the number of static blocks inside a class is arbitrary. They are executed only once.

```
public class ArticlesAndPronouns {
   private static int counter = 0;
   private static Set<String> ARTICLES = new HashSet<>();
   static {
      ARTICLES.add("a");
      ARTICLES.add("an");
      ARTICLES.add("the");
      System.out.println(ARTICLES);
   }
}
```

Similar to a class method, a static initialiser cannot use the `this` keyword or any instance fields or instance methods of the class. Static blocks are useful when static fields (esp. complex container types) require non-trivial initialisation code, and it's useful to have it close to the field declaration.

Instance initialisers

Classes also can have instance initialisers — same blocks but without the `static` keyword:

```
private Set<String> thirdPersonPronouns;
{
   thirdPersonPronouns = new HashSet<>();
   thirdPersonPronouns.add("instance " + counter);
   thirdPersonPronouns.add("he"); thirdPersonPronouns.add("she");
   thirdPersonPronouns.add("it"); thirdPersonPronouns.add("they");
}
```

The use of instance initialisers is rare. Also, the above block can be placed in a **doubly-delineated** block which immediately follows the reference assignment (code below is identical to that above):

```
private Set<String> thirdPersonPronouns = new HashSet<String>() {{
    add("instance " + counter);
    add("he"); add("she"); add("it"); add("they");
}};
```

This is more *compact but* rather *quaint*; also (?) — the type inference which allows to drop the type parameter value in the constructor (feature allowed since *Java 7*), doesn't work in the "doubly-bracey" version (☹). See the complete code in ArticlesAndPronouns.java

Where to look for this topic in the textbook?

- Hortsmann's *Core Java for the Impatient*, Ch. 2.1, 2.2, 2.4
- *Java Tutorial's* Classes and Objects