COSC 1020

Yves Lespérance

Lecture Notes

Week 12 — Implementing Classes II

Recommended Readings:
Horstmann: Ch. 7 and Ch. 2 Sec. 8
Lewis & Loftus: Ch. 5 Sec. 0 to 2

Class Methods

Suppose we want to add a method `isLegal(int no)` to the `CreditCard` class to allow users to check whether the passed number would be a legal credit card number (before calling the constructor):

```
public static boolean isLegal(int no)
{  return (0 < no && no <= 999999);
}
```

This cannot be an instance method because there is no instance yet. We can make it a *class method*. It would belong to the class, not to one of its instances. The method must be called *on the class*, e.g.

```
if (CreditCard.isLegal(703)) ...
```

In Java, class methods (and attributes) are declared `static`.

Similarly, for the class `Person` discussed earlier, we could define a class method that returned the maximum legal age:

```
public static int getUpperAgeLimit()
{  return 150;
}
```

To use it, you write for e.g.
`if (a > Person.getUpperAgeLimit())`

Class Attributes

Suppose we wanted to add a counter to the class `Person` to keep track of how many `Person` objects have been created. This counter would have to be a *class attribute/variable*.

To declare it, we would add

```
private static int count = 0;
```

to the class definition. We would also change the constructors to increment the counter, e.g.

```
public Person()
{  ...       // as before
   count++;
}
```

And we would add a method:

```
public static int getCount()
{  return count;
}
```

Then, we could use these as follows:

```
IO.println("Person count is "
                   + Person.getCount());
Person p1 = new Person();
IO.println("Person count is "
                   + Person.getCount());
Person p2 = new Person();
IO.println("Person count is "
                   + Person.getCount());
```

Class Constants

A class may also define some *constants* for its users. For e.g., the class `Stock` defines the constant `TSE_URL`, to store the URL used by the refresh method to establish a connection with the TSE. To declare `TSE_URL` for e.g., we would write

```
public static final String TSE_URL =
    "http://tse.com";
```

to the `Stock` class definition.

These are *constant attributes of the class*, not of its instances. You must refer to them using the class name, e.g.

```
IO.println(Stock.TSE_URL);
```

Why Define a Class?

There are two cases where defining a class is useful.

1. Your program needs to work with some kind of data, e.g. Persons. You want to *group together* the *data* and the *operations* that manipulate it.

You also want to *hide* the details of how the data is represented and how the operations are implemented from users of the class. The class will make some operations *public*, i.e. available to the users, and provide information on how to use them. This is the class's *interface*. The rest of the class's definition is *private* and hidden from users.

When such a class allows many different possible implementations, one says that the class defines an *abstract data type*; e.g. stack, list, binary tree, etc.

2. You want to *group together* a set of related operations in a *module*, e.g. the `Math` class. In this case, class users won't create instances of the class. The methods are associated with the class itself. In Java, they are labeled `static`.

Here too, the class supplies some *public* operations to users and provides information on how to use them in its *interface*. The rest of its definition is *private*.

In both cases, we say that the class *encapsulates*, i.e. hides, the details of its definition.

`javadoc`: A Documentation Utility

Important to have good documentation of classes' APIs.

Can use `javadoc` utility to help produce this.

You put special comments in the class's file and then run `javadoc` on it to produce an HTML API documentation file.

`javadoc` comments start with `/**`. Put one immediately before each method, non-private field, and before the class itself.

Special tags (must start line):

`@param` parameter-name description

`@return` description

`@exception` fully-qualified-class-name description

etc.

Can include other HTML tags e.g. `<code>`, `<it>`, etc.

See lab handbook and Horstmann for examples.

`javadoc` automatically adds links to existing classes.

When designing a class, document API using `javadoc` before writing code.

Use normal comments to document class implementation.

Steps to Class Implementation

Study API.

Write 1st version of class with fields and methods required by API, leaving out implementation for now; document using `javadoc`.

Write test harness that tests every feature of the class.

Identify private attributes and declare them.

Implement constructors, accessors, mutators, standard methods, specialized methods. Avoid redundancy by delegating and defining private methods.

Add new test cases as you implement methods. Test methods as early as possible. Fix bugs and run all tests again (bug fix may introduce new bugs).

Algorithms

As we saw in week 1, an algorithm is an unambiguous, executable, and terminating method for solving a problem.

There may be many different algorithms for solving a given problem, some efficient and some not.

Developing algorithms and analysing their efficiency is an important part of computer science.

E.g. Problem: Finding GCD

An integer d is a *divisor* of an integer n iff there exists an integer k such that $n = d \times k$.

g is the *greatest common divisor* of n and m iff g is a divisor of both n and m and there is no $g' > g$ that is also a divisor of both n and m.

E.g. $gcd(15, 55) = 5$
Why?
Identify prime factors:
$15 = 3 \times 5$
$55 = 5 \times 11$

Obvious Algorithm for Finding GCD

Try all integers between smallest of `n` and `m` and `1`; first one that is a divisor of both `n` and `m` is `gcd`.

```
i = min(n,m);
while(n % i != 0 || m % i != 0)
{  i--;
}
gcd = i;
```

Running time is proportional to `k`, where `k = min(n,m)`.

Euclid's Algorithm for Finding GCD

Observe that $gcd(n, m) = gcd(m, n\ mod\ m)$.

```
while(true)
{  r = n % m;
   if(r == 0)
   {  break;
   }
   n = m;
   m = r;
}
gcd = m;
```

Running time is proportional to less than `log(k)`, where `k = min(n,m)`.

E.g. Problem: Checking whether an Integer is Prime

A positive integer n is *prime* iff it has exactly 2 different divisors, 1 and n itself.

E.g. primes: 2, 3, 5, 7, 11, 13, etc.

Obvious Algorithm for Primality Testing

Count number of divisors of n between 1 and n itself.

```
divisors = 0;
for(i = 1; i <= n; i++)
{  if(n % i == 0)
   {  divisors++;
   }
}
isPrime = (divisors == 2);
```

Running time is proportional to n, O(n).

A Better Algorithm for Primality Testing

Only check 2 and odd numbers.

Only check up to $\sqrt{n}$

Return false as soon as an extra divisor is found.

```
if(n <= 1)
{  isPrime = false;
}
else if(n == 2)
{  isPrime = true;
}
else if(n % 2 == 0)
{  isPrime = false;
}
else
{  isPrime = true;
   limit = (int) (Math.sqrt(n)+1);
   for(i = 3; i <= limit; i+= 2)
   {  if(n % i == 0)
      {  isPrime = false;
         break;
      }
   }
}
```

Running time is proportional to $\sqrt{n}$, O($\sqrt{n}$).