ENCAPSULATION

MSCI 240: Algorithms & Data Structures

slides by Mark Hancock

lecture summary

encapsulation

variable shadowing

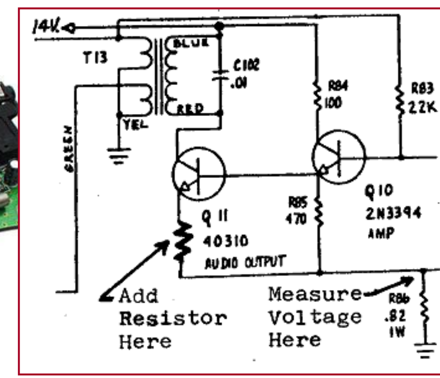
static methods/fields

classes as modules

encapsulation: hiding implementation details from clients

encapsulation forces abstraction

- separates external view (behaviour) from internal view (state)
- protects the integrity of an object's data



private field: a field that can't be accessed outside the class

```
private type name;
```

examples:

```
public class Student {
    private String name;    // each Student object has a
    private double gpa;     // name and gpa field
}
```

client code won't compile if it accesses private fields:

```
The field Fraction.numerator is not visible
at lecture.FractionClient.main(FractionClient.java:15)
```

accessing/modifying private state

```
public int getNumerator() {
    return numerator;
}

public void setNumerator(int numerator) {
    this.numerator = numerator;
}
```

client code will look more like this:

```
System.out.println(sum.getNumerator());
sum.setNumerator(5);
```

```java
// version of Fraction class using encapsulation
public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction(int numerator, int denominator) {
        setFraction(numerator, denominator);
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    public void setFraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

benefits of encapsulation

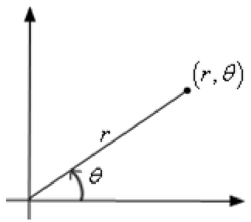
abstraction between object and clients

protects object from unwanted access

example: can't fraudulently increase an `Account`'s balance

can change the class implementation later

example: `Point` originally written using Cartesian coordinates (x, y) could be rewritten in polar coordinates (r, θ) with the same methods (client doesn't need to change)



can constrain objects' state (invariants)

example: don't allow `denominator` to be `0`

example: only allow `Date`s with a month from 1-12

```java
// version of Fraction class using encapsulation
// that checks for 0 denominator
public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction(int numerator, int denominator) {
        setFraction(numerator, denominator);
    }

    // ...

    public void setFraction(int numerator, int denominator) {
        if (denominator == 0) {
            throw new IllegalArgumentException("Can't be zero");
        }
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

the **this** keyword

`this` refers to the implicit parameter inside your class

a variable that stores the object on which a method is called

refer to a field:

```
this.field
```

call a method:

```
this.method(parameters);
```

one constructor can call another:

```
this(parameters);
```

variable shadowing: two variables with same name in same scope

normally illegal, except when one variable is a field

```
public class Fraction {
    private int numerator;
    private int denominator;
    //...
    public void setFraction(int numerator, int denominator) {
        //...
    }
}
```

in most of the class

`numerator` and **`denominator`** refer to the fields

in `setFraction`

`numerator` and `denominator` refer to the method's parameters

fixing shadowing

```java
public class Fraction {
    private int numerator;
    private int denominator;
    //...
    public void setFraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

inside `setLocation`,

to refer to the data field numerator, say **this**.numerator

to refer to the parameter numerator, say numerator

calling another constructor

```java
public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction() {
        this(0,1);
    }

    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

avoids redundancy between constructors

only a constructor (not a method) can call another constructor

`static` methods/fields

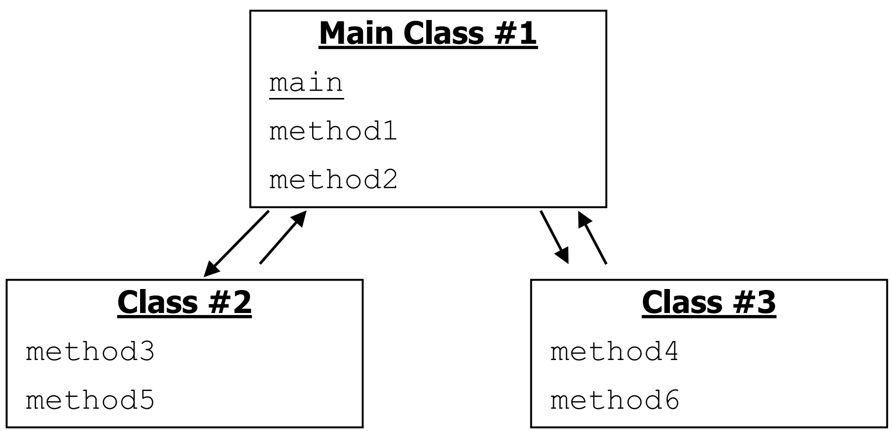
most large software systems consist of many classes

one main class runs and calls methods of the others

advantages:

code reuse

splits up the program logic into manageable chunks



example: two programs that deal with prime numbers

primes program 1

```java
// This program sees whether some interesting numbers are prime
public class RedundantPrimes1 {
    public static void main(String[] args) {
        int[] nums = { 1234517, 859501, 53, 142 };
        for (int i = 0; i < nums.length; i++) {
            if (isPrime(nums[i])) {
                System.out.println(nums[i] + " is prime");
            }
        }
    }

    // ...
```

```java
    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }

    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++; // i is a factor of the number
            }
        }
        return count;
    }
} // End of RedundantPrimes1 class
```

primes program 2

```
// This program prints all prime numbers up to a maximum
public class RedundantPrimes2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Max number? ");
        int max = console.nextInt();
        for (int i = 2; i <= max; i++) {
            if (isPrime(i)) {
                System.out.print(i + " ");
            }
        }
        System.out.println();
    }
```

```java
    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }

    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++; // i is a factor of the number
            }
        }
        return count;
    }
} // End of RedundantPrimes2 class
```

classes as modules

module: a reusable piece of software, stored as a class

example module classes: Math, Arrays, System

```
//This class is a module that contains useful methods
//related to factors and prime numbers.
public class Factors {
    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }

    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++; // i is a factor of the number
            }
        }

        return count;
    }
}
```

a module is a partial program, not a complete program

- it does not have a `main`; you don't run it directly
- modules are meant to be utilized by other client classes

syntax:

```
ClassName.method(parameters);
```

example:

```
int factorsOf24 = Factors.countFactors(24);
```

using a module

```
//This program sees whether some interesting numbers are prime.
public class Primes1 {
    public static void main(String[] args) {
        int[] nums = { 1234517, 859501, 53, 142 };
        for (int i = 0; i < nums.length; i++) {
            if (Factors.isPrime(nums[i])) {
                System.out.println(nums[i] + " is prime");
            }
        }
    }
}
```

using a module

```java
//This program prints all prime numbers up to a given maximum.
public class Primes2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Max number? ");
        int max = console.nextInt();
        for (int i = 2; i <= max; i++) {
            if (Factors.isPrime(i)) {
                System.out.print(i + " ");
        }   }
        System.out.println();
    }
}
```

modules in Java libraries

```java
// Java's built in Math class is a module
public class Math {
    public static final double PI = 3.141592653589793238462;

    // ...

    public static int abs(int a) {
        if (a >= 0) {
            return a;
        }
        else {
            return -a;
        }
    }

    public static double toDegrees(double radians) {
        return radians * 180 / PI;
    }
}
```

static: part of a class, rather than part of an object

object classes can have **static** methods and fields

not copied into each object;
shared by all objects of that class

class

state:
```
private static int staticFieldA
private static String staticFieldB
private int field1
private double field2
```
behaviour:
```
public static void someStaticMethodC()
public static void someStaticMethodD()
public void method3()
public int method4()
public void method5()
```

object #1

state:
```
int field1
double field2
```
behaviour:
```
public void method3()
public int method4()
public void method5()
```

object #2

state:
```
int field1
double field2
```
behaviour:
```
public void method3()
public int method4()
public void method5()
```

object #3

state:
```
int field1
double field2
```
behaviour:
```
public void method3()
public int method4()
public void method5()
```

`static` fields

```
private static type name;
```

or,

```
private static type name = value;
```

example:

```
private static int theAnswer = 42;
```

static field: stored in the class instead of each object

- a "shared" global field that all objects can access and modify
- like a class constant, except that its value can be changed

accessing **static** fields

from inside the class where the field was declared:

```
fieldName                           // get the value
fieldName = value;                  // set the value
```

from another class (if the field is public):

```
ClassName.fieldName                 // get the value
ClassName.fieldName = value;        // set the value
```

generally **static** fields are not **public** unless they are **final**

(counter-)example:

`System.out` is a static field of type `PrintStream`

`static` methods

```
// the same syntax you've already used for methods
public static type name(parameters) {
    statements;
}
```

static method: stored in a class, not in an object

- shared by all objects of the class, not copied
- does not have any implicit parameter, **`this`**;
- therefore, cannot access any particular object's fields
- (unless it is passed as an explicit parameter)

example: adding `Fraction` objects

```
Fraction f1 = new Fraction(3, 4);
Fraction f2 = new Fraction(2, 3);
```

option 1: instance method as mutator

```
f1.add(f2); // f1 becomes sum of f1 & f2
```

option 2: instance method as accessor (returns new Fraction)

```
f1 = f1.add(f2); // result of sum assigned to f1
```

option 3: static method (no implicit **`this`** parameter passed)

```
f1 = Fraction.add(f1, f2); // result assigned to f1
```

option 1: instance method as mutator

```java
public class Fraction {
    private int numerator;
    private int denominator;

    // ...
    public void add(Fraction other) {
        this.numerator = this.numerator * other.denominator
                + this.denominator * other.numerator;
        this.denominator = this.denominator
                * other.denominator;
    }
}
```

option 2: instance method as accessor

```
public class Fraction {
    private int numerator;
    private int denominator;

    // ...
    public Fraction add(Fraction other) {
        Fraction result = new Fraction();
        result.numerator = numerator * other.denominator
                + denominator * other.numerator;
        result.denominator = denominator * other.denominator;
        return result;
    }
}
```

option 3: static method

```java
public class Fraction {
    private int numerator;
    private int denominator;

    // ...
    public static Fraction add(Fraction a, Fraction b) {
        Fraction result = new Fraction();
        result.numerator = a.numerator * b.denominator
                + a.denominator * b.numerator;
        result.denominator = a.denominator * b.denominator;
        return result;
    }
}
```

summary of Java classes

a class is used for any of the following in a large program:

a program: has a `main` and perhaps other **`static`** methods

- example: most programs in MSCI 121, client code for object classes
- does not usually declare any **`static`** fields (except when **`final`**)

an object class: defines a new type of objects

- examples: `Point`, `Fraction`
- declares object fields, constructor(s), and methods
- might declare **`static`** fields or methods, but these are less of a focus
- should be encapsulated (all fields and static fields **`private`**)

a module: utility code implemented as **`static`** methods

- example: `Math`, `Arrays`

clicker questions

which of these statements from the program below is an illegal statement and will not compile?

A. `private int count;`

B. `Alpha bet = new Alpha();`

C. `bet.count = 1;`

D. `bet.number = 1;`

E. None of the above. The program will compile.

Alpha.java:

```java
class Alpha {
   private int count;
   public int number;
}
```

Program.java:

```java
class Program {
   static void main(String[] args) {
        Alpha bet = new Alpha();
        bet.count = 1;
        bet.number = 1;
   }
}
```

What is the output of the program?

A. 0

B. 2

C. 4

D. Does not compile because `sideLen` is private and cannot be accessed.

E. None of the above.

Alpha.java:

```
class Square {
    private double sideLen;

    public Square(double theLengthOfASide) {
        sideLen = theLengthOfASide;
    }

    public double getArea() {
        return sideLen * sideLen;
    }
}
```

Program.java:

```
class Program {
    public static void main(String[] args) {
        Square bob = new Square(2);
        System.out.println(bob.getArea());
    }
}
```

making a class’ member variables private has the following advantage(s):

A. because private member variables are “hidden” from outside the class, less computer memory is used than with public member variables.

B. also known as encapsulation or information hiding, private member variables allow the programmer to create a public interface that remains unchanged while the implementation of a class can be changed at a later date without affecting other parts of the program.

C. private member variables are more efficient and allow for exceptions to be thrown when something bad happens in a class.

D. A and B.

E. none of the above.

next class:

array vs. `ArrayList`