LECTURE 3: ENCAPSULATION

CS 2110
Fall 2021

Agenda

Previously in 2110:

- Types: strong typing, primitive types
- OOP: objects, classes, methods, fields

Today:

- Encapsulation
  - Access modifiers
  - Constructors
  - Avoiding getters/setters

Big programs

One Million Lines of Code

https://informationisbeautiful.net/visualizations/million-lines-of-code/

Big programs require big teams

Image: https://www.pxfuel.com/en/free-photo-xpbey

Big teams require strong walls

Image: https://www.flickr.com/photos/jlascar/4475570857

Encapsulation

- Encapsulate: enclose something in or as if in a capsule
- Encapsulate the object's implementation
  - Information hiding: fundamental design principle in OOP!
  - "Build a wall," usually around the state
    - **Hide state** behind the wall
    - **Reveal behaviors** that can be used
    - **Preserve assumptions** made by those behaviors
- Implementer of class is free to make changes that don't affect behavior
- Client of class never has to know

Demo

Fractions, v1

Fractions, v1


```
class Fraction1 {
    int num;
    int den;
    double toDouble() { return (double) num / den; }
    void print() { System.out.println(num + "/" + den); }
}
```

$$\frac{1}{0}$$ is irrational...

...let's encapsulate $\boxed{\dfrac{x}{y}}$

to prevent y from being 0

Demo

Fractions, v2

Fractions, v2

Access modifiers

```
class Fraction2 {
    private int num;
    private int den;  //
    /* omitted here: toDouble(), print() */
    public Fraction2(int n, int d) {
        assert d != 0;
        num= n;
        den= d;
    }
}
```

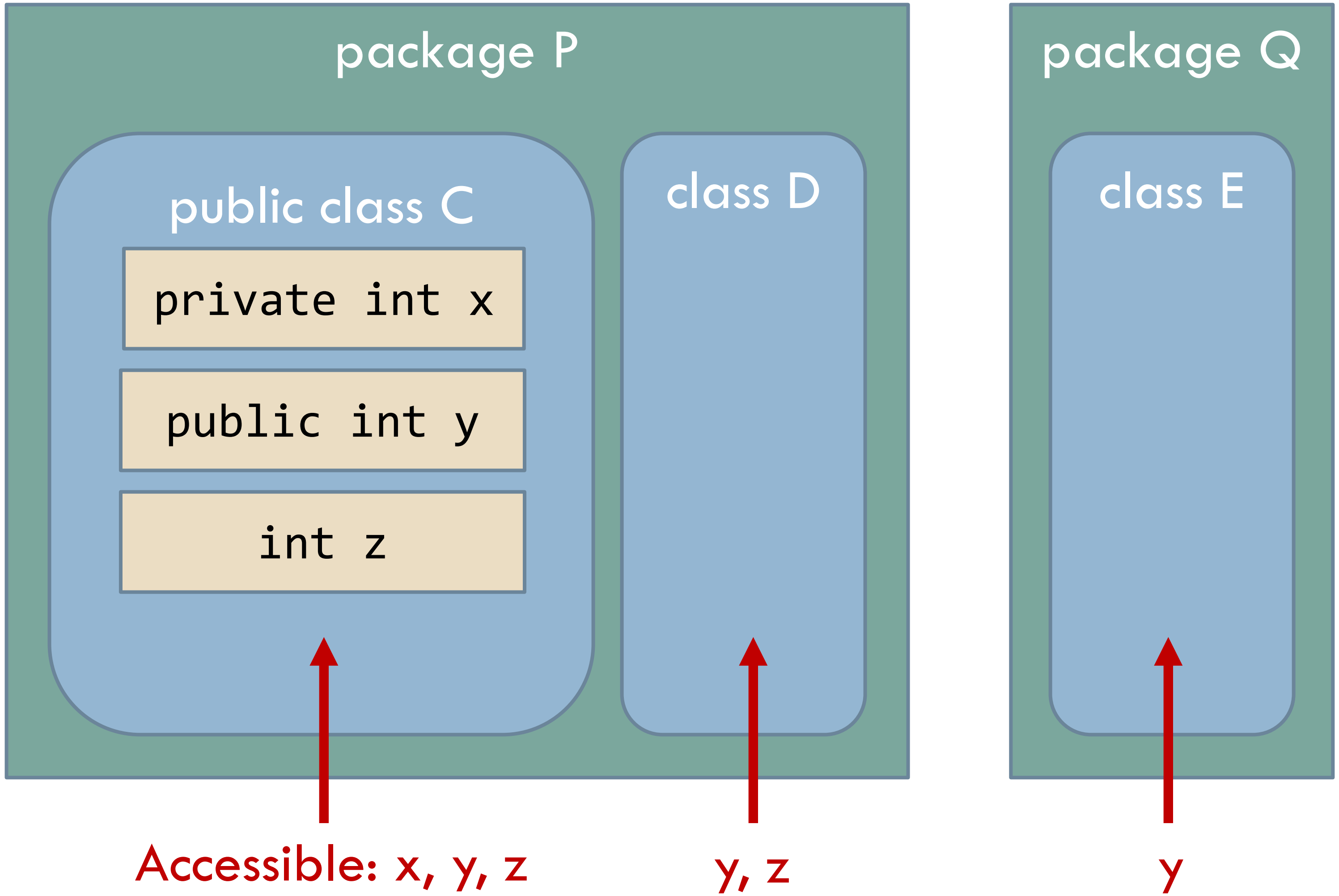
Constructor

Access modifiers

- **private:** can be accessed only by code inside same class
- **public:** can be accessed by code anywhere
- **(none):** (we call it package) can be accessed only by code inside same package
- **protected:** *in a later lecture*

These can be applied to components (fields, methods) and classes…

Access modifiers



LECTURE 3: ENCAPSULATION

PART 2: CONSTRUCTORS

CS 2110
Fall 2021

Fractions, v2

Access modifiers

```
class Fraction2 {
    private int num;
    private int den;  //
    /* omitted here: toDouble(), print() */
    public Fraction2(int n, int d) {
        assert d != 0;
        num= n;
        den= d;
    }
}
```

Constructor

Constructors

- Constructor: Purpose is to initialize object's state at creation

- Definition:

```
ClassName(parameter declarations) {
    …
}
```

- Definition looks like other methods except:
  - Name of constructor is name of class
  - No explicit return type

New-expression

- Constructor invoked with new expression:
  `new ClassName(arguments)`

- Evaluation: (revisited from lec 2)
  - Create a new object of class `ClassName`
  - Execute constructor call on new object: `ClassName(`*arguments*`)`
  - Yield object's name (address) as value of expression

New-expression

- We didn't define any constructors in lecture 2, so why we could write `new Counter()`?

- Java inserts default constructor if class does not define any constructors:

```
public className() { }
```

- But if you define a constructor, the default constructor is not inserted
  - So `Fraction2` doesn't have a default (which is sensible)
  - But `Fraction1` does (and it's not very sensible)

Providing many ways to initialize


```
public Fraction2(int n, int d) {
     assert d != 0;
     num= n;
     den= d;
}
```

Recall: overloading is using the same name for 2+ different methods of a class

A different initialization:

```
public Fraction2(int n) {
     num= n;
     den= 1;
}
```

Another way to code that:

```
public Fraction2(int n) {
     this(n, 1);
}
```

Delegating to another constructor

First statement of constructor can delegate to another with **`this`** keyword:

```
ClassName(parameter declarations) {
    this(arguments);
    // more code if desired
}
```

Useful to avoid repeating code between constructors

LECTURE 3: ENCAPSULATION

```java
private int field;

public int getField() {
    return field;
}

public void setField(int f) {
    field= f;
}
```

PART 3: GETTERS AND SETTERS

CS 2110
Fall 2021

Encapsulation, barely


```
private int x;
public int getX() { return x; }
public void setX(int newX) { x= newX; }
```

- **Getters and setters:** public methods to get and set values of a private field
- **Design advice:** do not mindlessly write getters and setters for every field

Instead of getters and setters...

- **Don't** have them at all
- **Do** permit query of object state:
  - e.g., `int numerator() { return num; }`
  - but **don't** call it `getNum()`: don't expose that it's getting the value of a particular field name
- **Do** provide behaviors that model the real world:
  - e.g. `void add(Fraction f) { … }`
  - but **not** `void setNum(int n) { num= n; }`

Demo

Fractions, v3

Fractions, v3

Implementation change:
keep fraction in reduced form

```
public Fraction3(int n, int d) {
    assert d != 0;
    num= n;
    den= d;
    reduce();
}
```

See website for full code

Slides elide details and docs for space!

What would `setNum()` do?

- Suppose `f = new Fraction(2, 4)`
- Then we call `f.setNum(4)`
- What should `f.toDouble()` return?

`f` is a *fraction*, not a pair of integers. The former has implied behavior, while the latter does not

Summary: Encapsulation

- Encapsulation: enclose the object
  - Make design robust
  - Build big programs
  - One of the key techniques of OOP

- Java features we discovered:
  - Access modifiers
  - Constructors
  - Avoid obvious getters/setters

Your turn: Read in JavaHyperText

- Access modifier (public, private), package
- Constructor (call, default), new-expression,
- Overload
- Getter setter