Abstraction & Classes

MSCI 240: Algorithms & Data Structures

lecture summary

review of classes
- abstraction
- fields

arrays of objects

instance methods

constructors

given a file of fractions (`fractions.txt`), begins with # fractions, each line has numerator then denominator:

```
5
3 4
5 8
-13 16
10 19
5 3
```

write a program that gets the sum of all of the fractions

a bad solution

```
Scanner input = new Scanner(new File("fractions.txt"));
int fractionCount = input.nextInt();
int[] numerators = new int[fractionCount];
int[] denominators = new int[fractionCount];

for (int i = 0; i < fractionCount; i++) {
    numerators[i] = input.nextInt();
    denominators[i] = input.nextInt();
}
// ...
```

parallel arrays: 2+ arrays with related data at same indexes

considered poor style

observations

the data in this problem is a set of fractions

it would be better stored as `Fraction` objects

- a `Fraction` would store a numerator/denominator pair
- we could perform common fraction operations (e.g., simplify)
- the overall program would be shorter and cleaner

class: a program entity that represents either:

1. a program / module, or
2. a template for a new type of objects

a class is a blueprint or template for constructing objects

object: an entity that combines data and behaviour

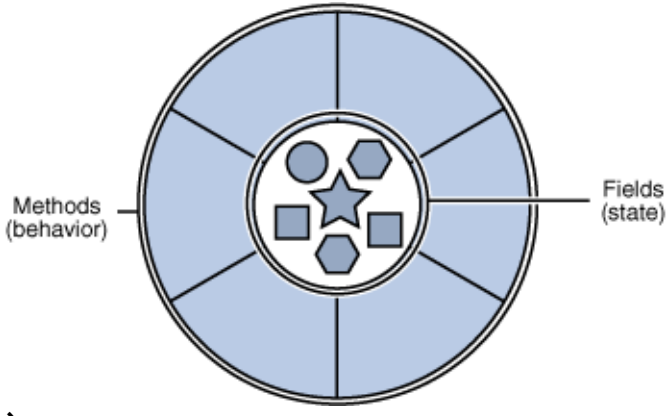
object-oriented programming (OOP): programs that perform their behaviour as interactions between objects

example: the `String` class (type) is a template for creating many `String` objects

Java has 1000s of built-in classes

object: an entity that contains data and behaviour

- data: variables inside the object
- behaviour: methods inside the object
- you interact with the methods; the data is hidden in the object



constructing (creating) an object:

```
Type objectName = new Type(parameters);
```

calling an object's method:

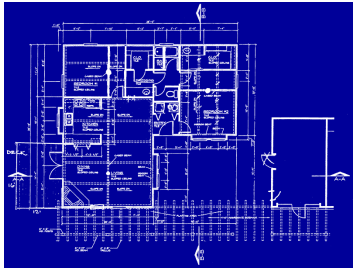
```
objectName.methodName(parameters);
```

blueprint analogy

iPod blueprint/factory

state:
current song
volume
battery life

behaviour:
power on/off
change station/song
change volume
choose random song



creates

iPod #1

state:
song = "I'm Gonna Be"
volume = 17
battery life = 2.5 hrs

behaviour:
power on/off
change station/song
change volume
choose random song

iPod #2

state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs

behaviour:
power on/off
change station/song
change volume
choose random song

iPod #3

state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs

behaviour:
power on/off
change station/song
change volume
choose random song

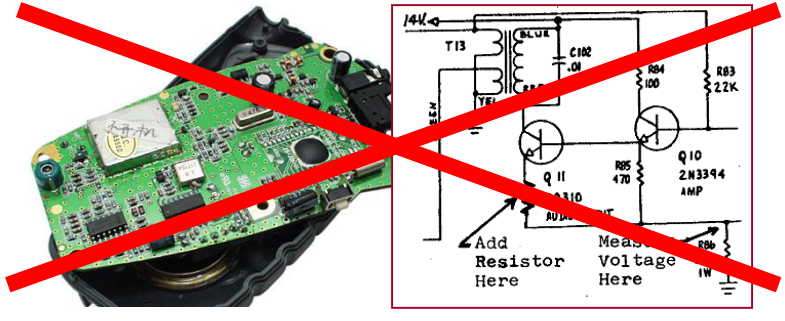
abstraction: a distancing between ideas and details

we can use objects without knowing how they work

abstraction in an iPod:

you understand its external behaviour (buttons, screen)

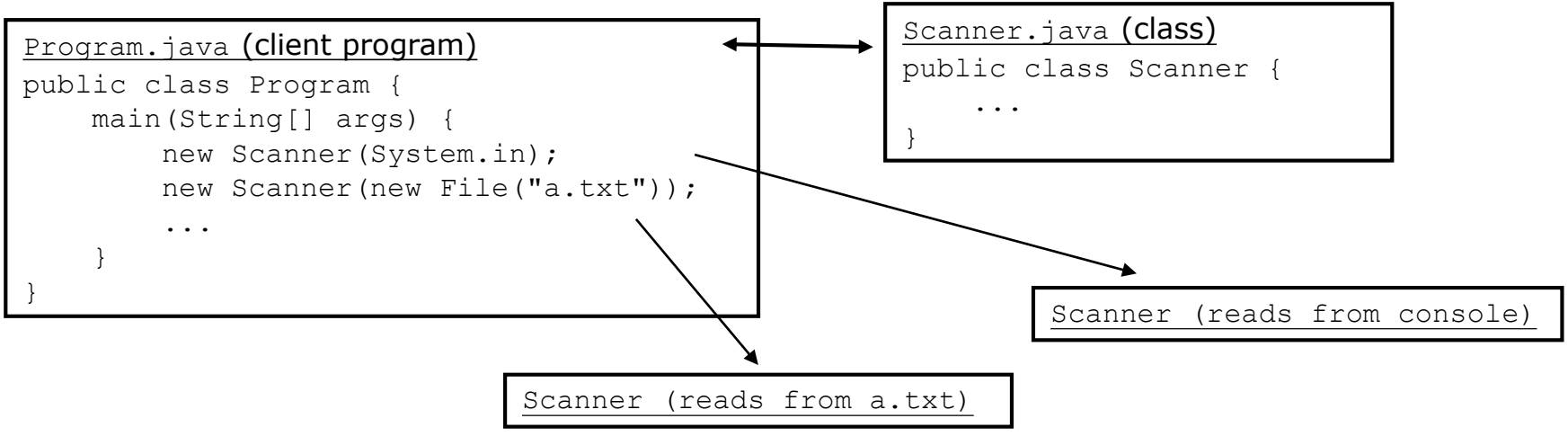
you don't understand its inner details, and you don't need to!



client program: a program that uses objects

example: many programs in 121 were clients of `Scanner` and `String`

you can write a program that uses `Scanner` without inner details



recall: `String` methods

Method name	Description
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (-1 if not found)
`length()`	number of characters in this string

`substring(`**`index1, index2`**`)`
or
`substring(`**`index1`**`)`
	the characters in this string from *index1* (inclusive) to *index2* (exclusive);
	if *index2* is omitted, grabs till end of string

`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

next we will implement a `Fraction` class as a way of learning about defining classes

- we will define a type of objects named `Fraction`
- each `Fraction` object will contain fields for the numerator and denominator
- each `Fraction` object will contain behaviour called methods
- client programs will use the `Fraction` objects

```
Fraction f1 = new Fraction(3, 4); // the fraction 3/4
Fraction f2 = new Fraction();     // zero (e.g., 0/1)
```

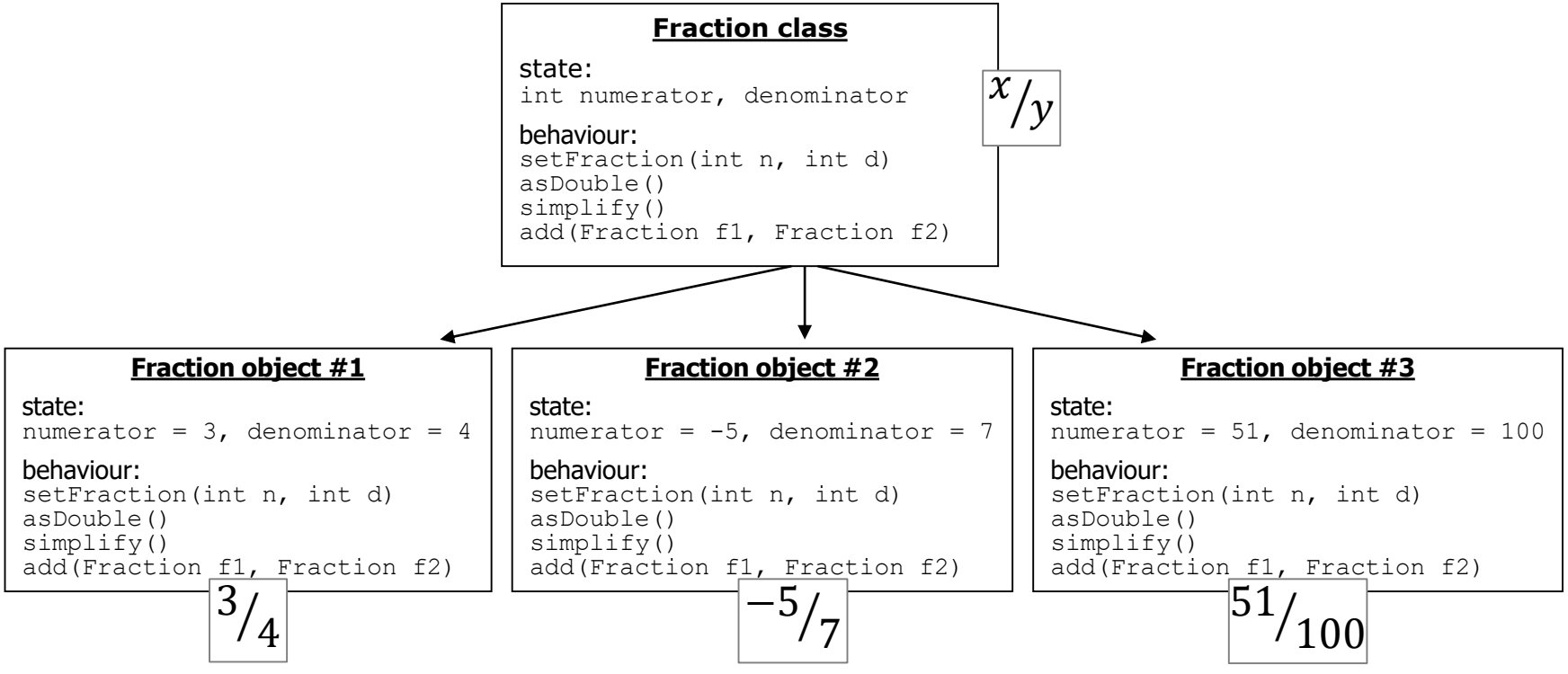
data (fields) in each `Fraction` object:

Field name	Description
`numerator`	the fraction's numerator
`denominator`	the fraction's denominator

methods (behaviour) in each `Fraction` object:

Method name	Description
`setFraction(`**`n`**`, `**`d`**`)`	sets the fraction's numerator and denominator to the given values
`asDouble()`	gets the floating-point value for this fraction
`simplify()`	simplifies this fraction
`add(f1, f2)`	adds two fractions and returns the result as a new fraction (should be `static` – will discuss later)

`Fraction` class as blueprint



the class (blueprint) will describe how to create objects

each object will contain its own data and methods

field: a variable inside an object that is part of its state

each object has its own copy of each field

declaration syntax:

```
type name;
```

example:

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

`Fraction` class, version 1

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

code goes in a file named `Fraction.java`

creates a new type named `Fraction`

each `Fraction` object contains two pieces of data:

- an `int` named `numerator`, and
- an `int` named `denominator`

`Fraction` objects don't contain any behaviour (yet)

accessing fields

other classes can access/modify an object's fields

```
variable.field          // access
variable.field = value; // modify
```

example:

```
Fraction f1 = new Fraction();
Fraction f2 = new Fraction();
System.out.println("numerator is " + f1.numerator); // access
f2.denominator = 13;                                // modify
```

a class and its client

`Fraction.java` is not, by itself, a runnable program

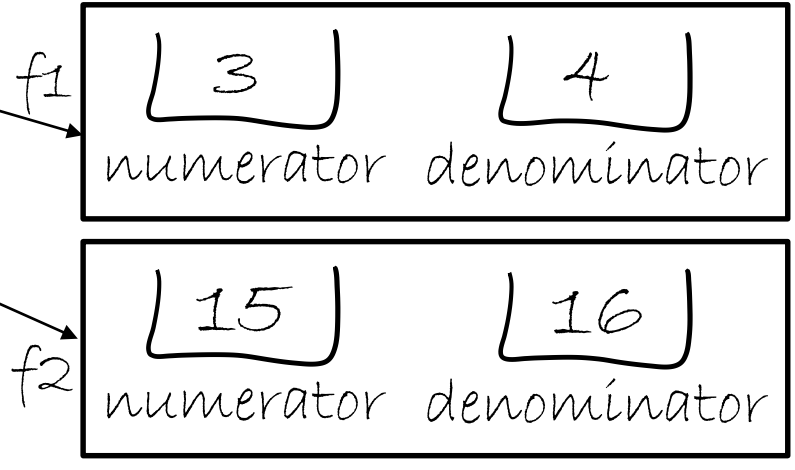
a class can be used by client programs

`FractionMain.java` (client program)

```
public class FractionMain {
    public static void main(String[] args) {
        Fraction f1 = new Fraction();
        f1.numerator = 3;
        f1.denominator = 4;

        Fraction f2 = new Fraction();
        f2.numerator = 15;
        f2.denominator = 16;
        // ...
    }
}
```

`Fraction.java` (class of objects)

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



```
public class FractionMain {
    public static void main(String[] args) {
        Fraction f1 = new Fraction();
        f1.denominator = 4;
        Fraction f2 = new Fraction();
        f2.numerator = 13;

        System.out.println(
                f1.numerator + "/" + f1.denominator); // 0/4

        // adjust f2 and then print it
        f2.numerator += 2;
        f2.denominator++;
        System.out.println(
                f1.numerator + "/" + f1.denominator); // 15/1
    }
}
```

activity:
draw what this looks like in memory

summary so far

an object is an entity that combines data and behaviour

a class can represent a template for a new type of object

a class has:

- fields – the data/state of the object
- methods – the behaviour that can be done to/on/with/etc. the object

you access fields and methods using dot notation

arrays of objects

recall: a class is a template for a new type of object…

… so it's okay to create an array of any type you create!

```
Fraction[] fractions = new Fraction[2]; // works!
```

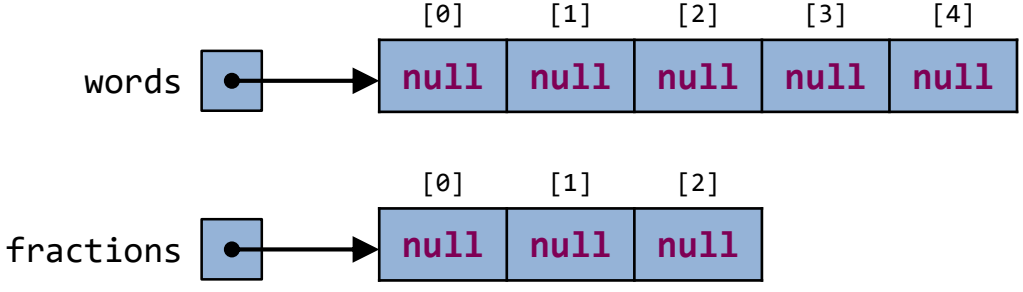

```
public class FractionMain {
    public static void main(String[] args) {
        Fraction[] fractions = new Fraction[2];
        fractions[0].numerator = 3;
        fractions[0].denominator = 4;

        fractions[1].numerator = 15;
        fractions[1].denominator = 16;

        for (int i = 0; i < fractions.length; i++) {
            System.out.println(fractions[i].numerator + "/"
                    + fractions[i].denominator);
        }
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at FractionMain.main(FractionMain.java:6)
```

`null`: a value that does not refer to any object

the elements of an array of objects are initialized to `null`

```
String[] words = new String[5];
Fraction[] fractions = new Fraction[3];
```



things you can do w/ **null**

store **null** in a variable or an array element

```
String s = null;
words[2] = null;
```

print a **null** reference

```
System.out.println(s);      // null
```

ask whether a variable or array element is **null**

```
if (words[2] == null) {
    // ...
}
```

things you can do w/ `null`

pass `null` as a parameter to a method

```
int i = s.indexOf(null);
```

return `null` from a method (often to indicate failure)

```
return null;
```

NullPointerException

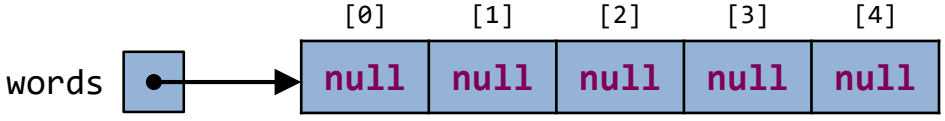
dereference: to access data or methods of an object with the dot notation, such as `s.length()`

it is illegal to dereference **`null`** (causes an exception)

null is not any object, so it has no methods or data

NullPointerException

```
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase(); // ERROR
```

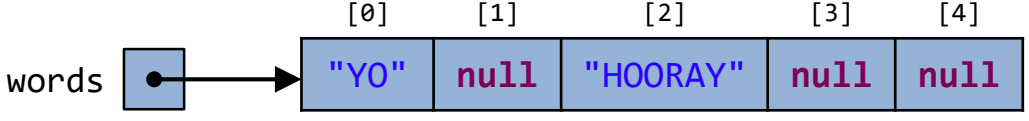


output:

```
word is: null
Exception in thread "main" java.lang.NullPointerException
   at FractionMain.main(FractionMain.java:7)
```

you can check for **null** before calling an object's methods

```
String[] words = new String[5];
words[0] = "yo";
words[2] = "hooray";   // words[1], [3], [4] are null

for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        words[i] = words[i].toUpperCase();
    }
}
```

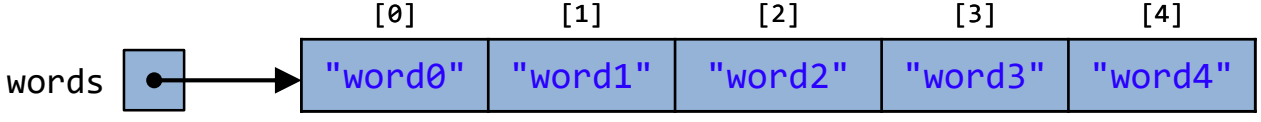


two-phase initialization

phase 1: initialize the array itself (each element is initially **null**)

phase 2: initialize each element of the array to be a new object

```
String[] words = new String[5];              // phase 1
for (int i = 0; i < words.length; i++) {
    words[i] = "word" + i;                   // phase 2
}
```



arrays of objects summary

classes are just types of objects, so you can make arrays of these (e.g., `String[]`, `Fraction[]`, `Scanner[]`, etc.)

by default, all values are **`null`**, so you need to initialize arrays of objects in two phases

- phase 1: initialize the array (and its size)
- phase 2: initialize each element (usually in a **`for`** loop)

given a file of fractions (`fractions.txt`), begins with # fractions, each line has numerator then denominator:

```
5
3 4
5 8
-13 16
10 19
5 3
```

write a program that gets the sum of all of the fractions

```
public class FractionClient {

    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("fractions.txt"));
        Fraction[] fractions = readFractions(input);
        Fraction sum = sum(fractions);

        System.out.println("The sum of fractions is "
                + sum.numerator + "/" + sum.denominator);
    }

    //...
```

```
//...

public static Fraction[] readFractions(Scanner input) {
    int fractionCount = input.nextInt();
    Fraction[] fractions = new Fraction[fractionCount];
    for (int i = 0; i < fractions.length; i++) {
        fractions[i] = new Fraction();
        fractions[i].numerator = input.nextInt();
        fractions[i].denominator = input.nextInt();
    }
    return fractions;
}
```

```
    public static Fraction sum(Fraction[] fractions) {
        Fraction result = new Fraction();
        result.numerator = 0;
        result.denominator = 1;

        // add each fraction to the current result
        for (int i = 0; i < fractions.length; i++) {
            result.numerator = result.numerator
                    * fractions[i].denominator
                    + result.denominator * fractions[i].numerator;
            result.denominator = result.denominator
                    * fractions[i].denominator;
        }

        return result;
    }
} // end FractionClient class
```

what is wrong with this code?

object behaviour: methods

client code redundancy

our client program wants to set `Fraction` object numerators and denominators:

```
for (int i = 0; i < fractions.length; i++) {
    fractions[i] = new Fraction();
    fractions[i].numerator = input.nextInt();
    fractions[i].denominator = input.nextInt();
}
```

to initialize them in other places, the code must be repeated

we can remove this redundancy using a method

eliminating redundancy, v1

we can eliminate the redundancy with a **`static`** method:

```
// Sets the value of a fraction
public static void setFraction(Fraction f, int n, int d) {
    f.numerator = n;
    f.denominator = d;
}
```

`main` (or `readFractions` or `sum`) would call the method:

```
for (int i = 0; i < fractions.length; i++) {
    fractions[i] = new Fraction();
    setFraction(fractions[i], input.nextInt(), input.nextInt());
}
```

problem with `static` method

we are missing a major benefit of objects: code reuse

- each program using `Fraction` needs a `setFraction` method

the syntax doesn't match how we're used to using objects

```
setFraction(fractions[i], 1, 4);  // static (bad)
```

the point of classes is to combine state and behaviour

- `setFraction` behaviour is closely related to a `Fraction`'s data
- the method belongs inside each `Fraction` object

```
fractions[i].setFraction(1, 4);  // inside object (better)
```

instance method (or object method): exists inside each object of a class and gives behaviour to each object

```
public type name(parameters) {
    statements;
}
```

same syntax as static methods, but without **`static`** keyword

example:

```
public void shout() {
    System.out.println("HELLO THERE!");
}
```

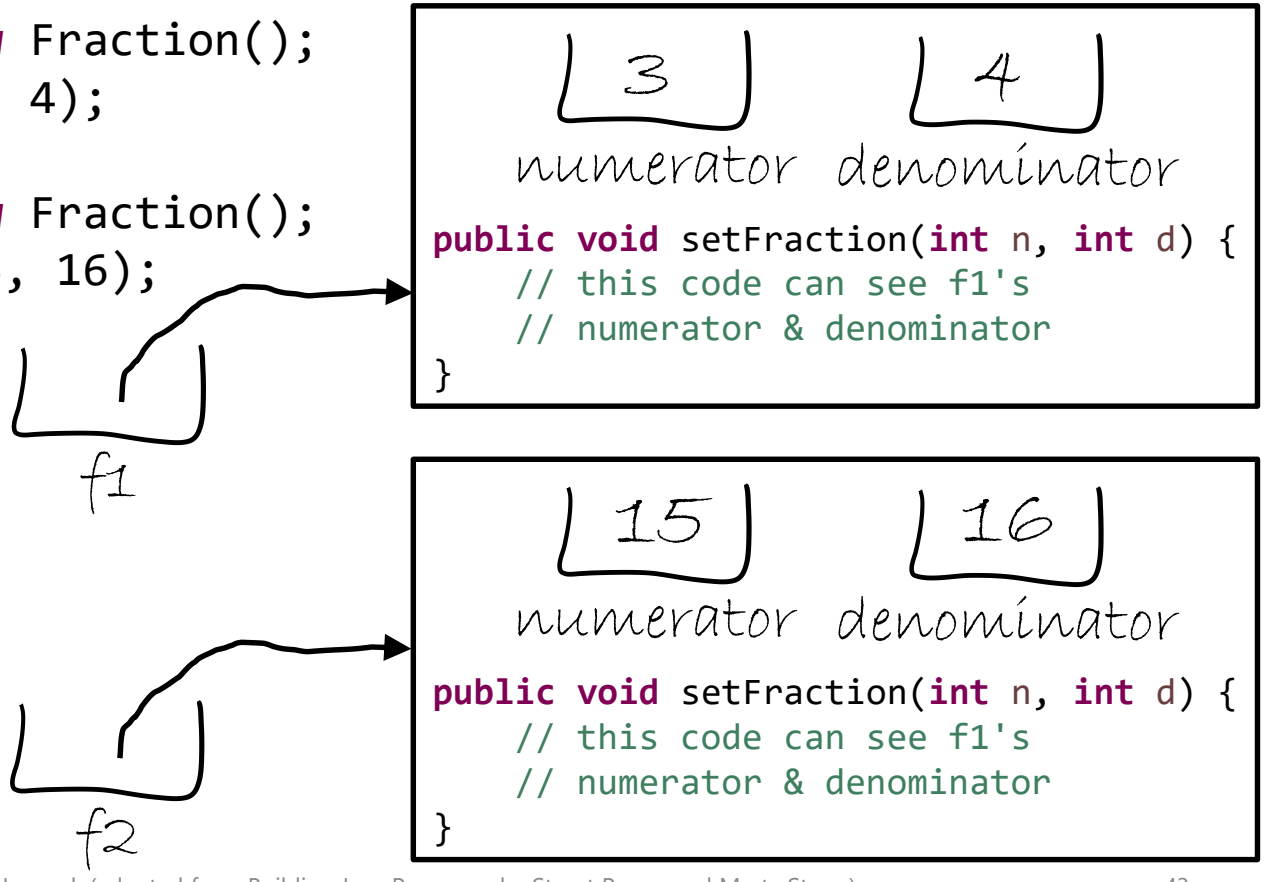
```
public class Fraction {
    int numerator;
    int denominator;

    public void setFraction(int n, int d) {
        numerator = n;
        denominator = d;
    }
}
```

the `setFraction` method no longer has a `Fraction f` parameter

- how will the method know which fraction to set?
- how will the method access that fraction's numerator/denominator?

each `Fraction` object has its own copy of the `setFraction` method, which operates on that object's state:

```
Fraction f1 = new Fraction();
f1.setFraction(3, 4);

Fraction f2 = new Fraction();
f2.setFraction(15, 16);
```



the implicit parameter (`this`)

the object an instance method is called on (from the client)

```
f1.setFraction(3, 5);  // f1 is the implicit parameter
f2.setFraction(1, 18); // f2 is the implicit parameter
```

the instance method (`setFraction`, in this case) can refer to that object's fields (in the `Fraction` class)

`this.numerator` and `this.denominator`

OR just `numerator` and `denominator`

we say that it executes in the context of a particular object

kinds of methods

accessor: a method that lets clients examine object state

examples: `getNumerator`, `asDouble`

often has a non-void return type

mutator: a method that modifies an object's state

examples: `setFraction`, `simplify`

object behaviour summary

instance methods act on an instance of an object and have an implicit **`this`** parameter (the object before the dot)

accessor methods let you examine the object's state

mutator methods let you modify the object's state

object initialization: constructors

currently it takes 3 lines to create a `Fraction` and initialize it (or 2 with the `setFraction` method):

```
Fraction f = new Fraction();
f.numerator = 3;
f.denominator = 8; // tedious ヽ(ʘ_ʘ)ノ
```

we'd rather specify the fields' initial values at the start:

```
Fraction f = new Fraction(3, 8);   // better! ヽ(•‿•)ノ
```

(we are able to do this with most types of objects in Java)

constructor: initializes the state of new objects

```
public Type() {
    statements;
}
```

runs when the client uses the **`new`** keyword

no return type is specified: implicitly "returns" the new object being created

if a class has no constructor, Java gives it a default constructor with no parameters that sets all fields to zero (`0`, `0.0`, **`null`**)

constructor example

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

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

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

multiple constructors

a class can have multiple constructors

each one must accept a unique set of parameters

default constructor: a constructor with no parameters

this default constructor initializes a `Fraction` to $^0/_1$

```
// Constructs a new fraction 0/1
public Fraction() {
    this.numerator = 0;
    this.denominator = 1;
}
```

constructor summary

constructors are used to initialize the state of new objects and are called when you use the **`new`** keyword

constructors do not have a return type (not even **`void`**)

you can have multiple constructors, as long as the parameters are different (type or number)

next class:

encapsulation