CMPSC 100

Computational Expression

Classes vs. Objects

A class == the code underlying the functionality

e.g. public class Die (Die.java)

An object == the class initialized in code

e.g. Die d6 = new Die(6);

Classes

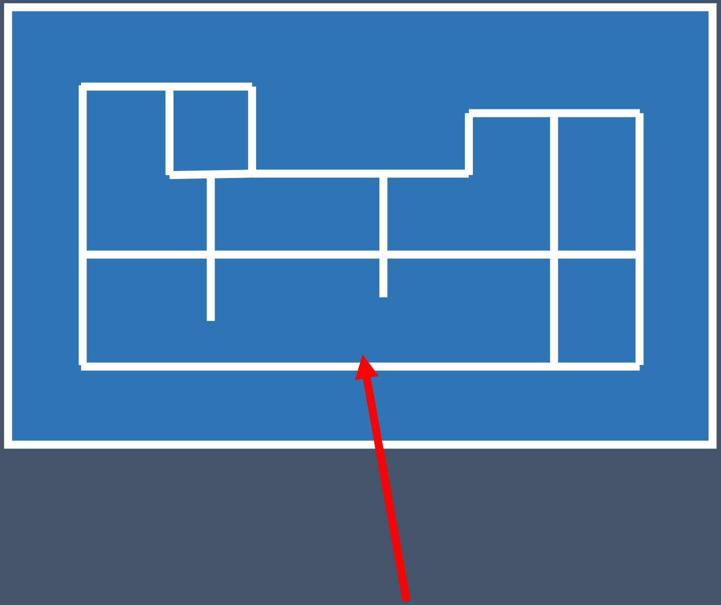
$$\frac{3}{5}$$

¯_(ツ)_/¯

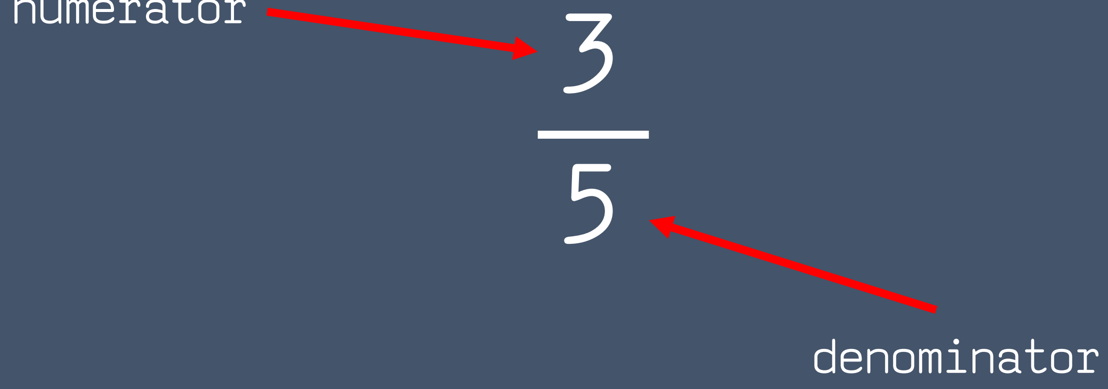
.6, LOLZ

Classes

For example, what is the *minimum amount* of information we need to create a fraction?



Like a “blueprint” for functionality

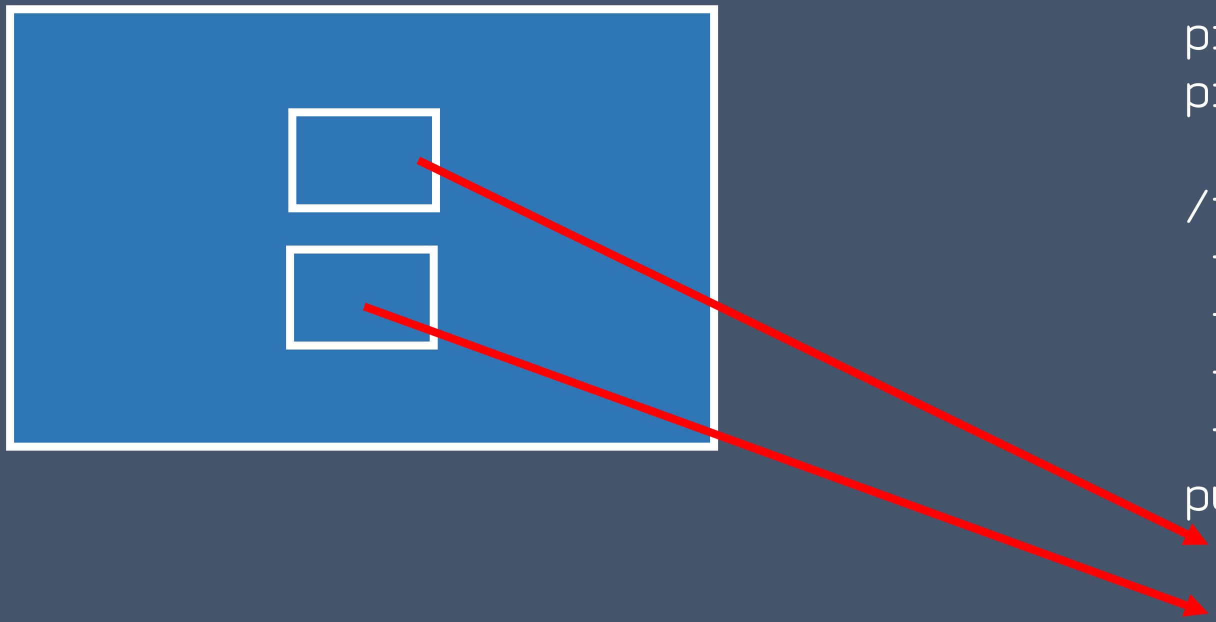


numerator

$$\frac{3}{5}$$

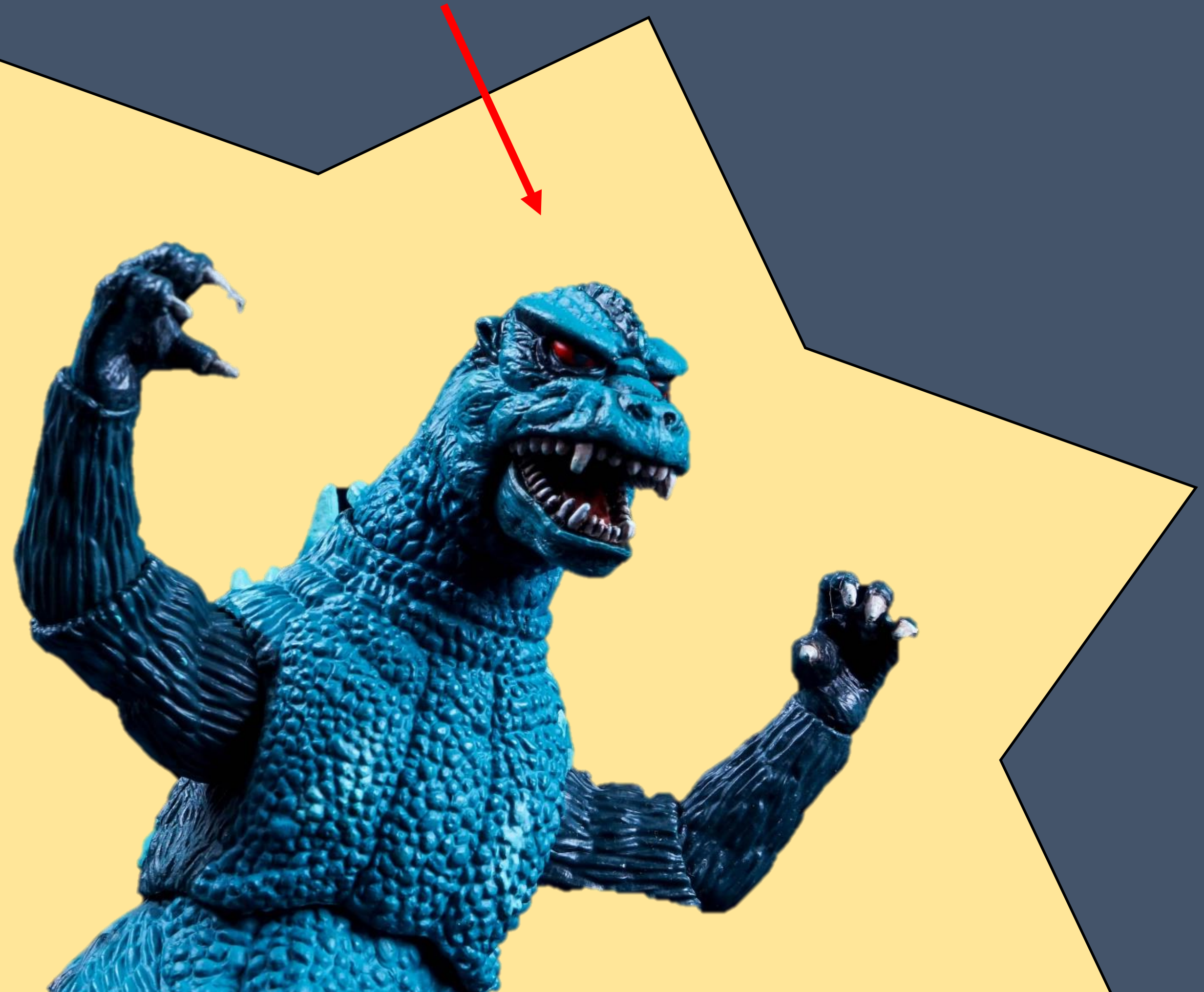
denominator

Classes

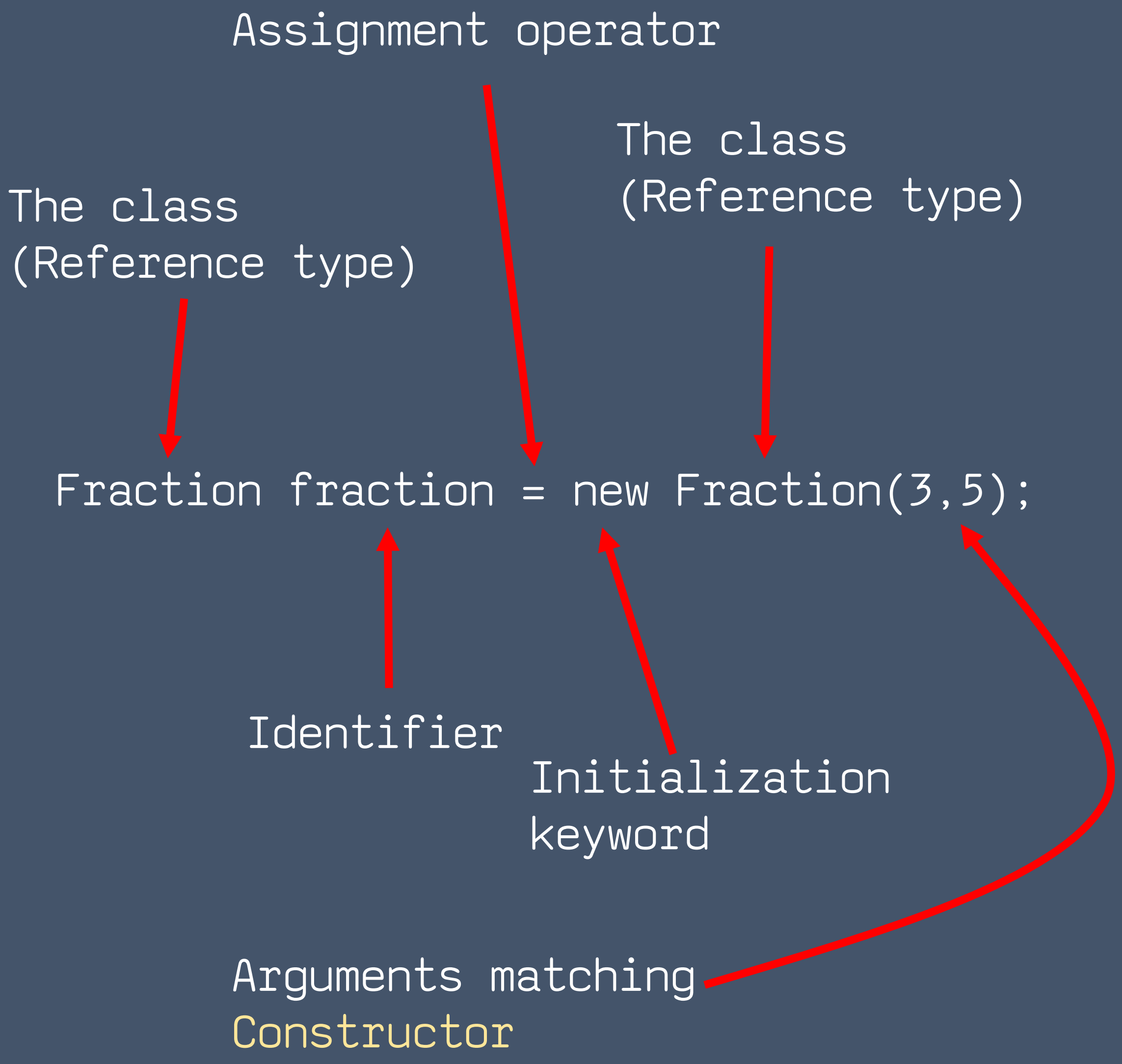


```
/** Makes a fraction.
 *
 * @author The Professor
 */
public class Fraction {

  private int numer;
  private int denom;

  /** Constructor.
   *
   * @param numer Numerator
   * @param denom Denominator
   */
  public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }
}
```

Objects



The living, rampaging, OBJECTZILLA



Assignment operator

The class
(Reference type)

The class
(Reference type)

```
Fraction fraction = new Fraction(3,5);
```

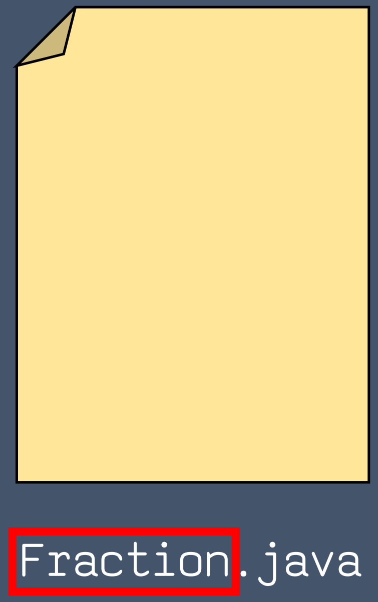
Identifier

Initialization
keyword

Arguments matching
Constructor

Classes vs. Objects

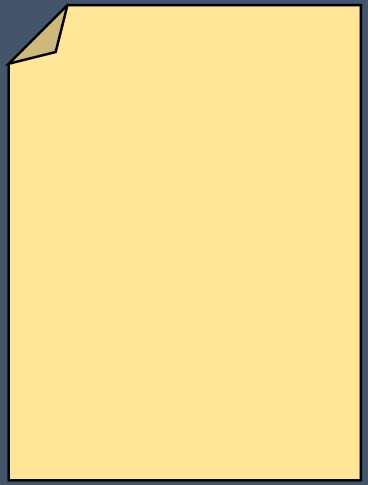
```
/** Makes a fraction.
 *
 * @author The Professor
 */
public class Fraction {

  private int numer;
  private int denom;

  /** Constructor.
   *
   * @param numer Numerator
   * @param denom Denominator
   */
  public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }
}
```



Fraction.java

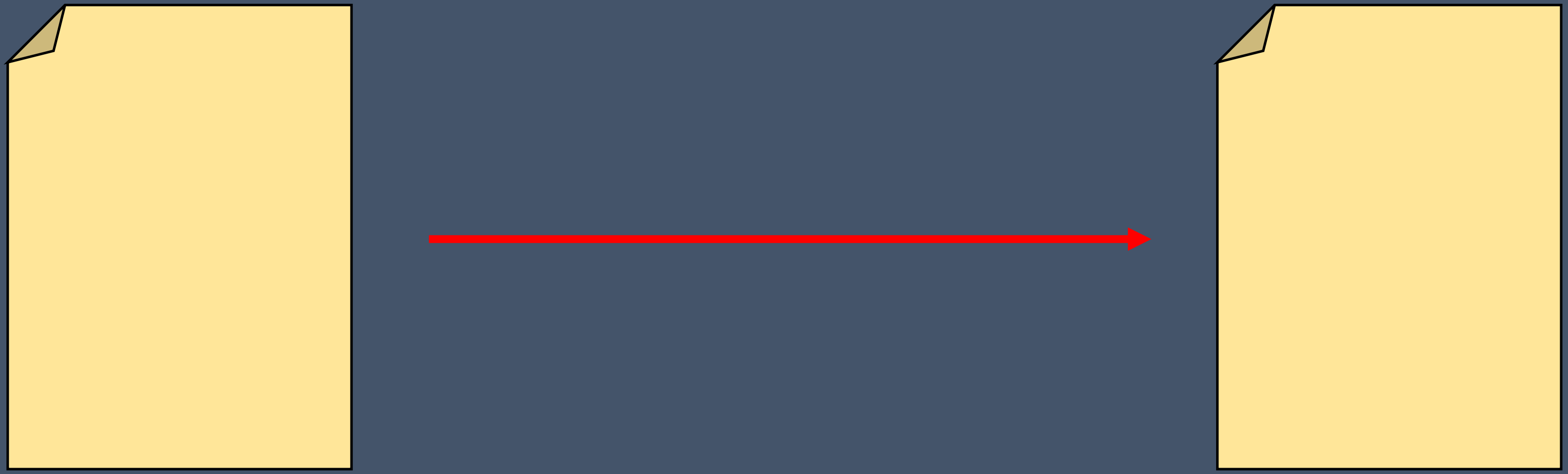
Classes vs. Objects

```
public class FractionJackson {

  public static void main(String[] args) {
    Fraction fraction = new Fraction(3,5);
  }
}
```



FractionJackson.java

Classes vs. Objects



Fraction.java

FractionJackson.java

Classes and Methods

```
/** Makes a fraction.
 *
 * @author The Professor
 */
public class Fraction {

  private int numer;
  private int denom;

  /** Constructor.
   *
   * @param numer Numerator
   * @param denom Denominator
   */
  public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }
}
```

Constructor method

Called *immediately* when an object is **initialized**

Requests/requires the *minimum* amount of data required to create the object. In this case, all we need is a **numerator** (numer) and a **denominator** denom.

Classes and “instances”

```
/** Makes a fraction.
 *
 * @author The Professor
 */
public class Fraction {

  private int numer;
  private int denom;

  /** Constructor.
   *
   * @param numer Numerator
   * @param denom Denominator
   */
  public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }
}
```

These are “instance” variables. We can also refer to them as “global” variables for Fraction.java.

These apply *everywhere* in Fraction.java, but can’t be modified outside of it because they are **encapsulated**.

Notice that it refers to:

this.numer	->	**numer**
This.denom	->	**denom**

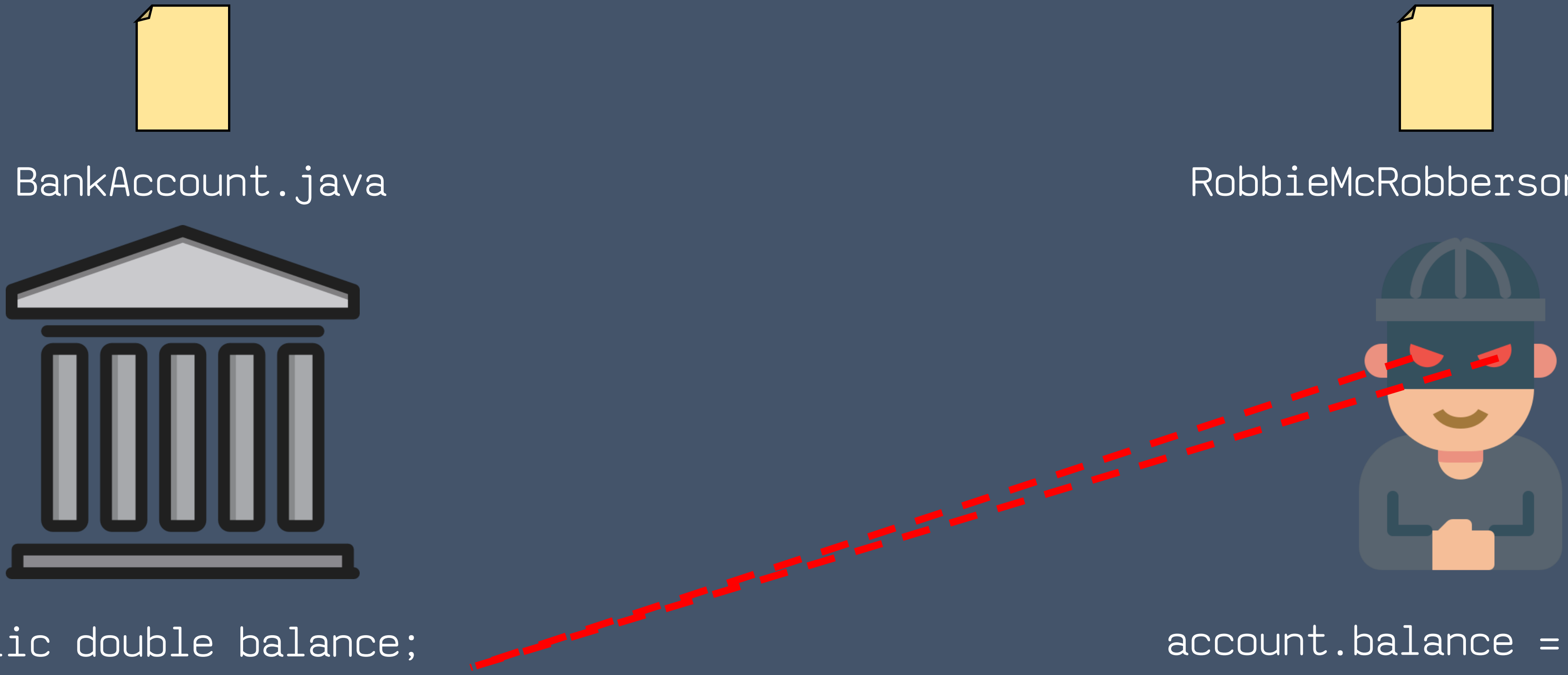
Detour: encapsulation

A quick definition of **encapsulation**:

An object’s properties and attributes should only be modifiable by methods contained ***within that object***

Detour: encapsulation

Imagine the following scenario:



BankAccount.java

```
public double balance;
public int accountNum;
```

RobbieMcRobberson.java

```
account.balance = 0;
print(account.accountNum)
```

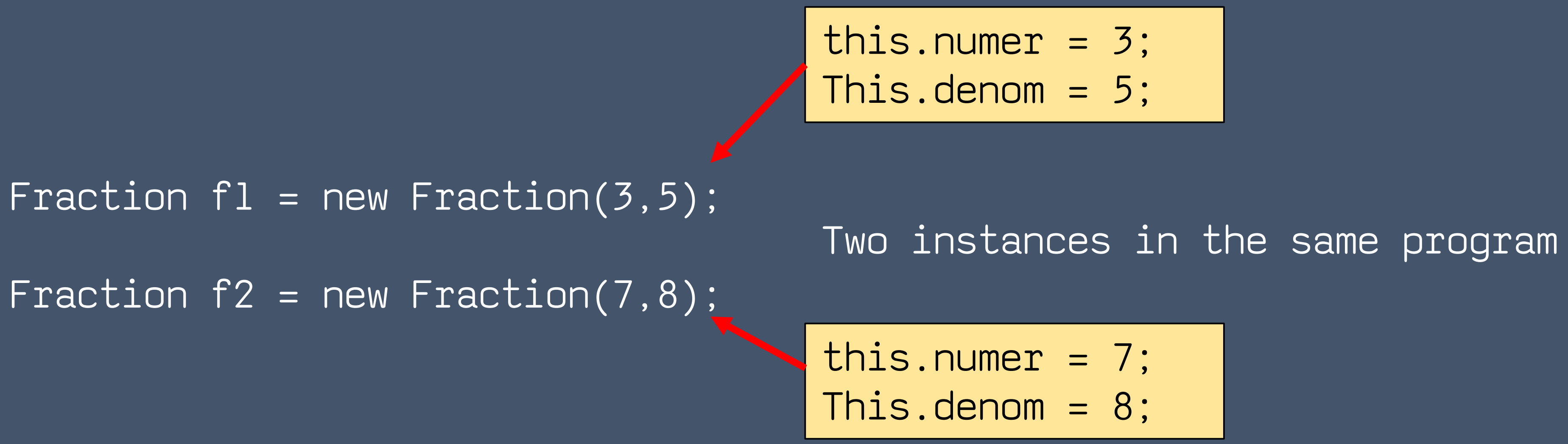
The this keyword

```
public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
 }
```

We refer to these variables as being **local** to the Constructor **method**

We refer to these as **instance** variables global to a **given instance** of a Fraction **object**

Instances and instance variables



```
this.numer = 3;
This.denom = 5;
```

```
Fraction f1 = new Fraction(3,5);

Fraction f2 = new Fraction(7,8);
```

Two instances in the same program

```
this.numer = 7;
This.denom = 8;
```

Each **instance** creates a new, living copy of Fraction

Classes and methods

```
/** Makes a fraction.
 *
 * @author The Professor
 */
public class Fraction {

  private int numer;
  private int denom;

  /** Constructor.
   *
   * @param numer Numerator
   * @param denom Denominator
   */
  public Fraction(int numer, int denom) {
    this.numer = numer;
    this.denom = denom;
  }
}
```

Here, we have a basic Fraction **class**, but no real services or **methods** with which to do anything using our Fraction **objects**

Classes and methods

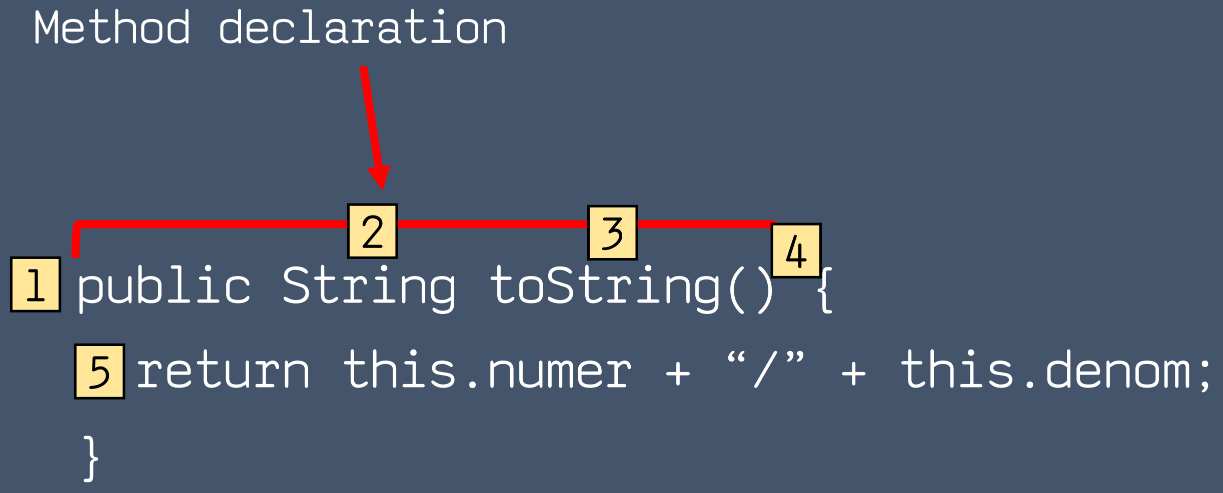
```
/** Constructor.
 *
 * @param numer Numerator
 * @param denom Denominator
 */
public Fraction(int numer, int denom) {
  this.numer = numer;
  this.denom = denom;
}

public String toString() {
  return this.numer + “/” + this.denom;
}
```

We’ll call this the toString **method**.

String is the *return type* of this method, because we want a **String** back when we call it.

public indicates that other files can see and read these

A detour into methods



```
public String toString() {
  return this.numer + “/” + this.denom;
}
```

Necessary parts of a **method declaration**:

1 Visibility keyword

2 Return type

3 Identifier

4 Parameters

5 Return statement
(for any non-**void** return type)

A detour into methods: the `void` return type

We’ve seen **void** before:

```
    public static void main(String[] args)
```

void means to return *nothing*

(e.g. the method does something that doesn’t require us to see or use it directly)

Activity

cd to your Activities repository

Perform a git pull download master

cd to the activity-08 folder and open the Java files in the src directory

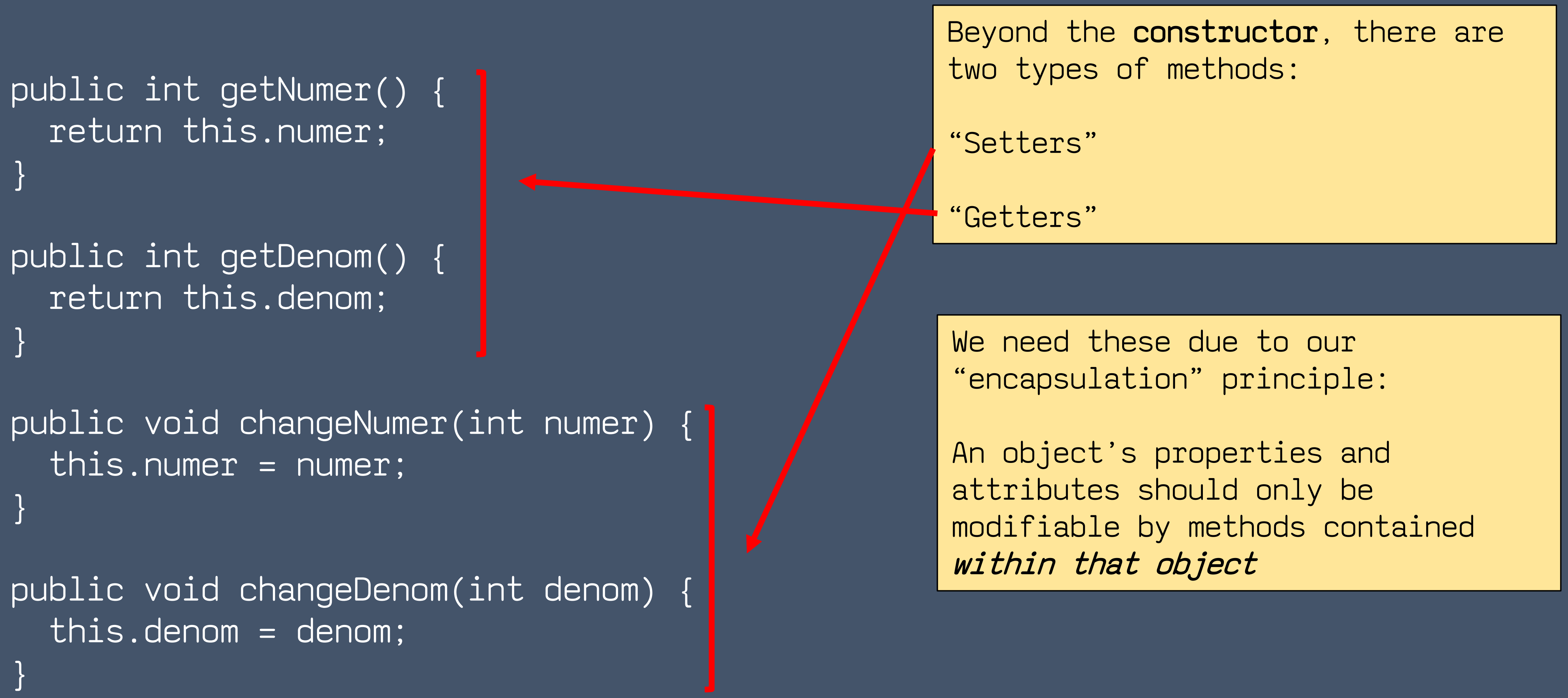
Activity

We’ll pick up where we left off on Monday:

`cd` to your Activities repository

`cd` to the activity-08 folder and open the Java files in the `src` directory

More practice with objects



```
public int getNumer() {
  return this.numer;
}

public int getDenom() {
  return this.denom;
}

public void changeNumer(int numer) {
  this.numer = numer;
}

public void changeDenom(int denom) {
  this.denom = denom;
}
```

Beyond the **constructor**, there are two types of methods:

“Setters”

“Getters”

We need these due to our “encapsulation” principle:

An object’s properties and attributes should only be modifiable by methods contained ***within that object***

A bit of mind-bending

```
public Fraction inverse() {
  return new Fraction(this.denom, this.numer);
}
```

Fraction.java

```
Fraction f3 = f1.inverse();
Fraction f4 = f2.inverse();
```

MakeFraction.java

A bit of mind-bending

```
public Fraction inverse() {
  return new Fraction(this.denom, this.numer);
}
```

This works because (by the time we *can* call this) Java has compiled the **class**, and we are able to use the class as a **return type.**

Even reference types can be return types (**String**, anyone?)

A bit of mind-bending

Our challenge

To implement a method in Fraction.java which:

- Is visible to other Java files
- Returns an object of the Fraction type
- Which multiplies two Fractions together