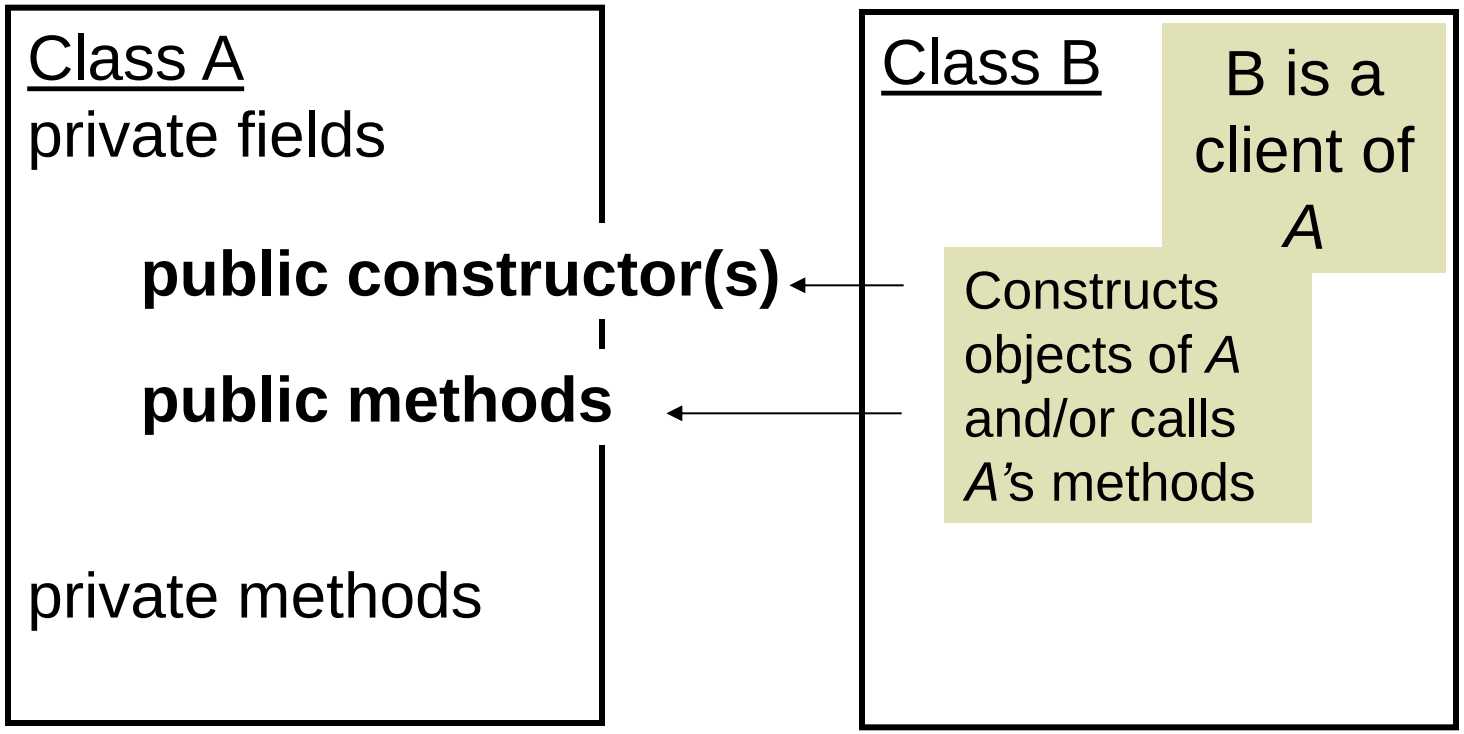
OOP Classes and Objects

Class's Client

- Any class that uses class *A* is called a *client* of *A*



Classes vs Objects

- Classes are used to define objects and provide methods to act on the objects
- Classes are also programs that declare these objects and process them to solve a problem
- Objects are variables that are named instances of a class
  - the class is their type
- Objects have both data and methods
- Both the data items and methods of a class are *members* of the object
- Data items are also called *fields* or *instance variables*
- *Invoking* a method means to *call* the method, i.e. execute the method
  - Syntax for invoking an object's method: the dot operator `object_Variable_Name.method()`
  - `object_Variable_Name` is the *calling object*

Public vs. Private

- Public constructors and methods of a class constitute its **interface** with classes that use it — its clients.
- All fields are usually declared private — they are hidden from clients.
- Static constants can occasionally be public.
- “Helper” methods that are needed only inside the class are declared private.

Public vs. Private (cont'd)

- A private field is accessible anywhere within the **class'** source code.
- Any object can access and modify a private field of another object of the same class.

```
public class Fraction
{
    private int num, denom;
    ...
    public multiply (Fraction other)
    {
        int newNum = num * other.num;
        ...
```

Accessors and Modifiers

- A programmer often provides methods, called *accessors*, that return values of private fields; Methods that set values of private fields are called *modifiers* or ***mutators***.
- Accessors' names often start with **get**, and modifiers' names often start with ***set.***
- These are not precise categories: the same method can modify several fields or modify a field and also return its old or new value.

Accessor and mutator methods

- Some authors:
  - *accessor methods* are the ones that read values of instance variables (read data from an object)
  - *mutator methods* are ones that change the values of instance variables (write data to an object)

- These authors:
  - *accessor methods* are more generic: they are methods that either read data from or write data to an object
  - the term *mutator method* is not used

Encapsulation

- Hiding the implementation details of a class (making all fields and helper methods private) is called *encapsulation*.
- Encapsulation helps in program maintenance: a change in one class does not affect other classes.
- A client of a class interacts with the class only through well-documented public constructors and methods; this facilitates team development.

Encapsulation (cont'd)

```
// Private fields:
private  <sometype>  myField;
...

// Constructors:
public MyClass (...)   { ... }
...
// Public methods:
public  <sometype>  myMethod (...)   { ... }
...
// Private methods:
private  <sometype>  myMethod (...)  { ... }
...
```

Public interface: public constructors and methods

ABSTRACT DATA TYPE

- **An Object-Oriented approach used by several languages**
- **A term for *class* implementation**
  - **a container for both data items and methods to act on the data**
- **Implements information hiding and encapsulation**
- **Provides a public *user interface* so the user knows how to use the class**
  - **descriptions, parameters, and names of its methods**
- **Implementation:**
  - **private instance variables**
  - **method definitions are usually public but always hidden from the user**
  - **the user cannot see or change the implementation**
  - **the user only sees the interface**

Constructors

- A constructor is a procedure for creating objects of the class.
- A constructor often initializes an object's fields.
- Constructors do not have a return type (not even void) and they do not return a value.
- All constructors in a class have the same name — the name of the class.
- Constructors may take parameters.

Constructors (cont'd)

- If a class has more than one constructor, they must have different numbers and/or types of parameters.
- Programmers often provide a "no-arguments" constructor that takes no parameters.
- If a programmer does not define any constructors, Java provides one default no-arguments constructor, which allocates memory and sets fields to the default values.

Constructors (cont'd)

```
public class Fraction
{
   private int num, denom;

   public Fraction ( )
   {
      num = 0;
      denom = 1;
   }

   public Fraction (int n)
   {
      num = n;
      denom = 1;
   }

   public Fraction (int n, int d)
   {
      num = n;
      denom = d;
      reduce ();
   }

   public Fraction (Fraction other)
   {
      num = other.num;
      denom = other.denom;
   }

}
```

"No-args" constructor

Copy constructor

Constructors (cont'd)

- Constructors of a class can call each other using the keyword **this** — a good way to avoid duplicating code:

```
public class Fraction
{

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

```
public Fraction (int p, int q)
{
    num = p;
    denom = q;
    reduce ();
}
```

Operator new

- Constructors are invoked using the operator new.
- Parameters passed to new must match the number, types, and order of parameters expected by one of the constructors.

```
Fraction  f1 = new Fraction ( );
Fraction  f2 = new Fraction (5);
Fraction  f3 = new Fraction (4, 6);
Fraction  f4 = new Fraction (f3);
```

```
public class Fraction
{
   public Fraction (int n)
   {
      num = n;
      denom = 1;
   }
```

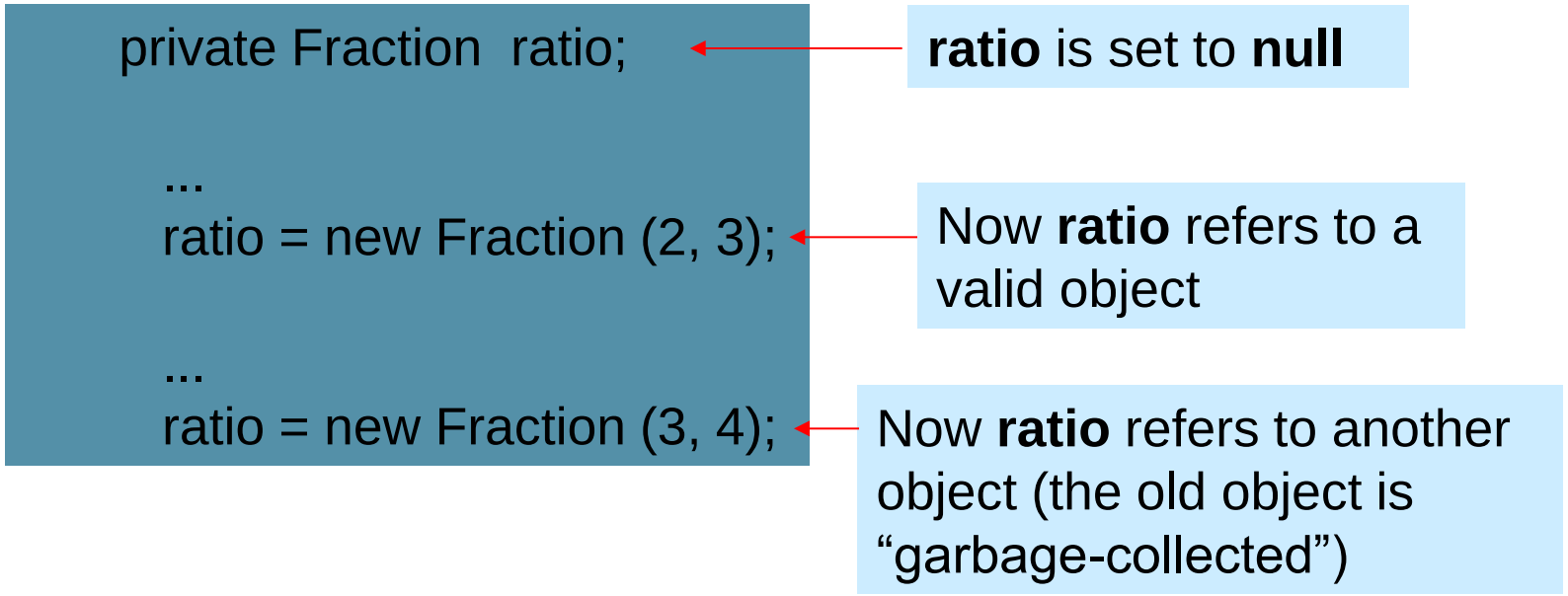
5 / 1

The reserved word ***this***

- The word ***`this`*** has a special meaning for objects
- It is a *reserved* word, which means you should not use it as an identifier for a variable, class or method
- ***`this`*** stands for the name of the calling object
- Java allows you to omit ***`this`*** .
  - It is automatically understood that an instance variable name without the keyword `this` refers to the calling object

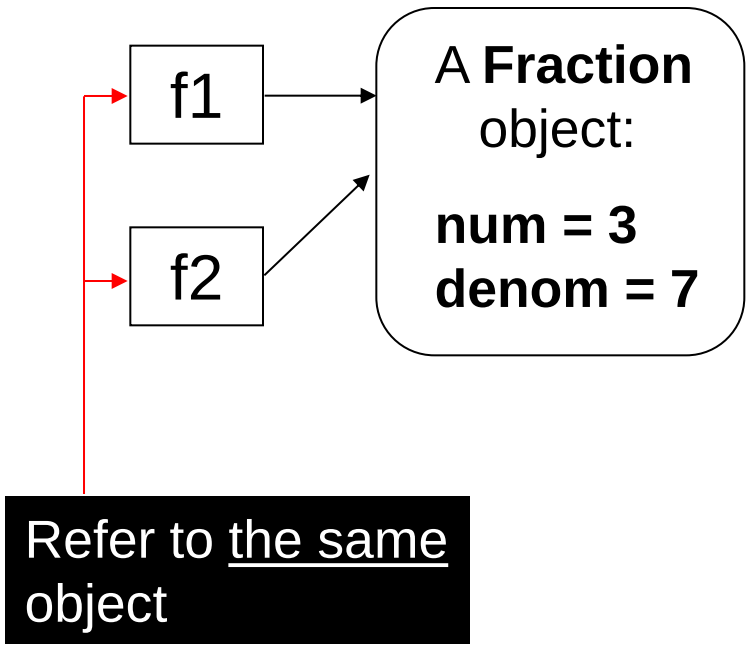
Operator new (cont'd)

- You must create an object before you can use it; the new operator is needed.

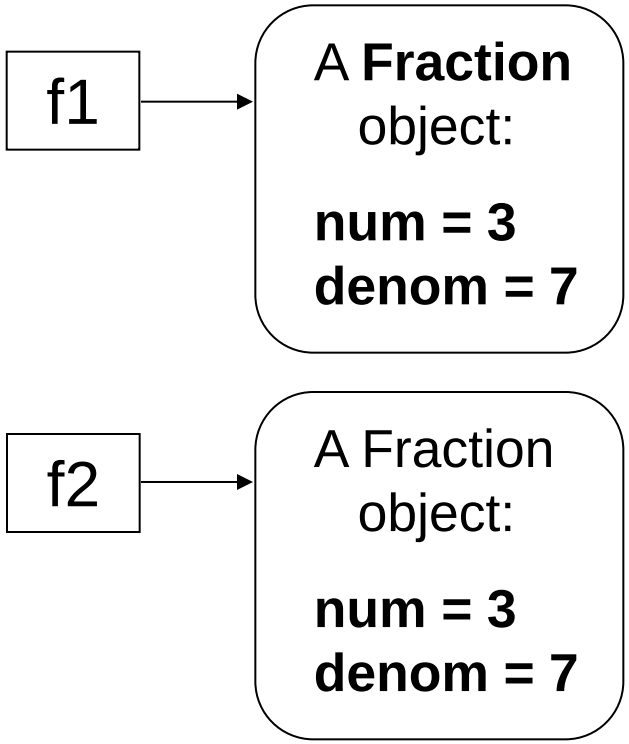


References to Objects

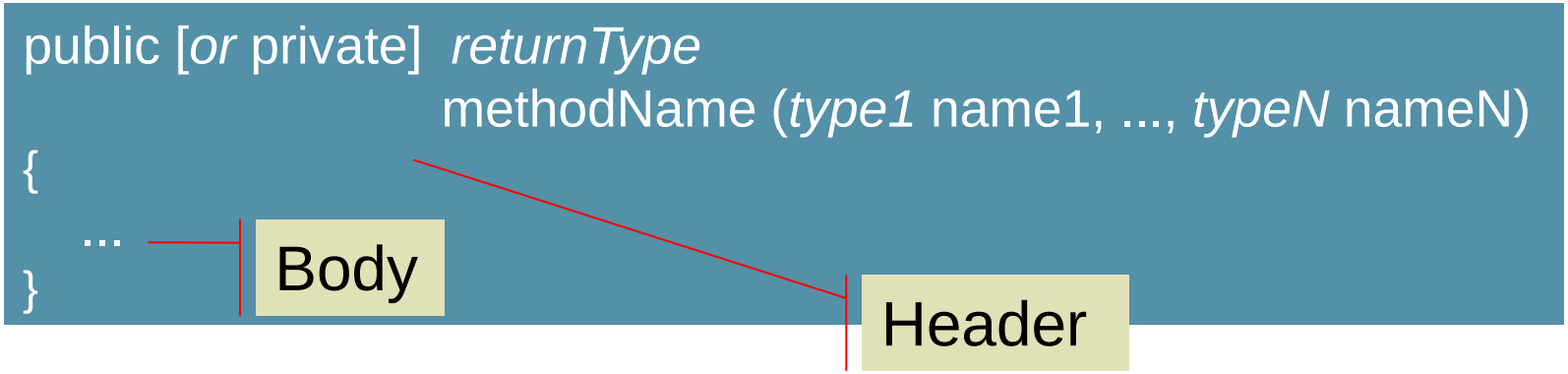
```
Fraction  f1 = new Fraction(3,7);
Fraction  f2 = f1;
```




```
Fraction  f1 = new Fraction(3,7);
Fraction  f2 = new Fraction(3,7);
```



Methods



To define a method:

- decide between public and private (usually public)
- give it a name
- specify the types of parameters and give them names
- specify the method's return type or chose void
- write the method's code

Methods (cont'd)

- A method is always defined inside a class.
- A method returns a value of the specified type unless it is declared void; the return type can be any primitive data type or a class type.
- *A method's parameters can be of any primitive data types or class types*.

Empty parentheses indicate that a method takes no parameters

Methods: Java Style

- A method name starts with a lowercase letter.
- Method names usually sound like verbs.
- The name of a method that returns the value of a field often starts with get:

  getWidth, getX
- The name of a method that sets the value of a field often starts with set:

  setLocation, setText

Passing Parameters to Constructors and Methods

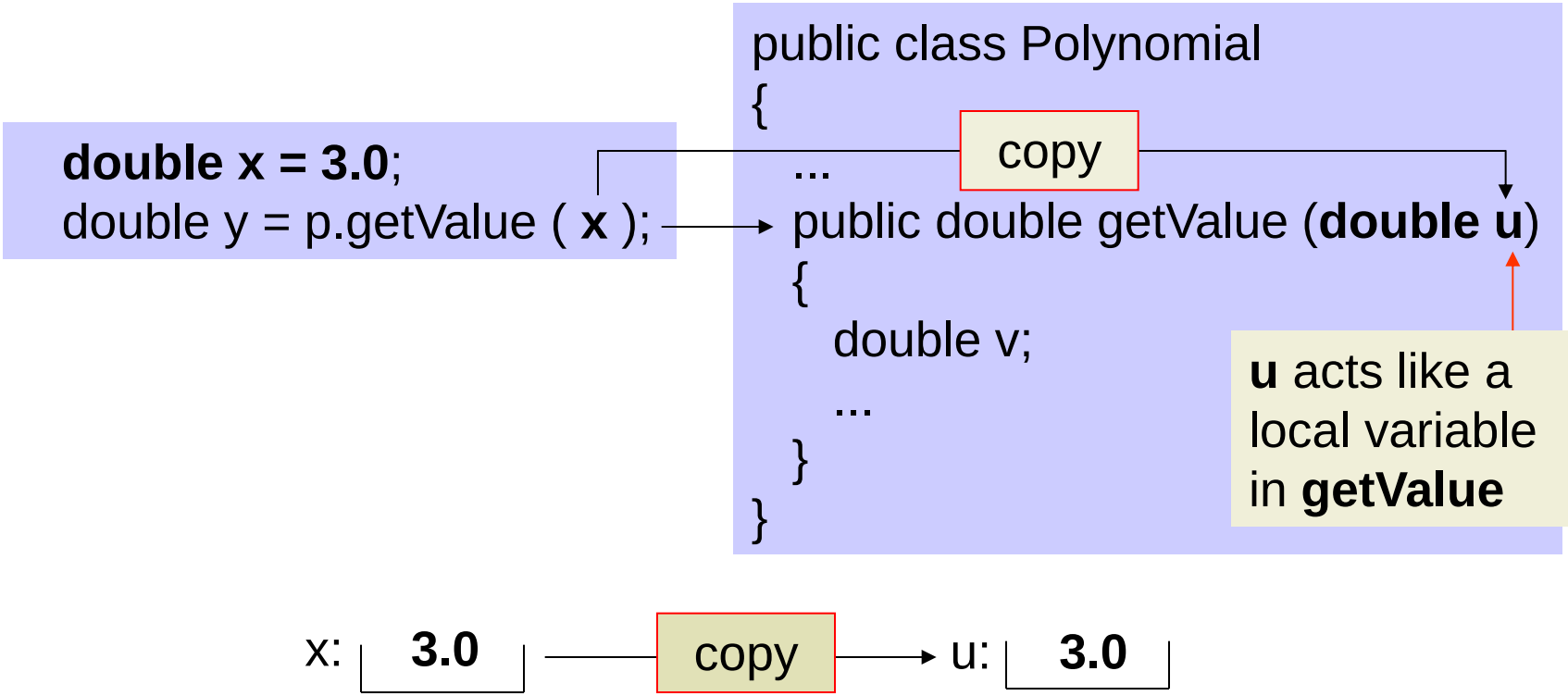
- Any expression that has an appropriate data type can serve as a parameter:

```
double u = 3, v = –4;
...
Polynomial p = new Polynomial (1.0,  –(u + v),  u ∗ v);
double y = p.getValue (2 ∗ v – u);
```

```
public class Polynomial
{
   public Polynomial (double a, double b, double c) { ... }
   public double getValue (double x) { ... }
```

Passing Parameters (cont'd)

- Primitive data types are always passed "by value": the value is copied into the parameter.



Passing Parameters (cont'd)

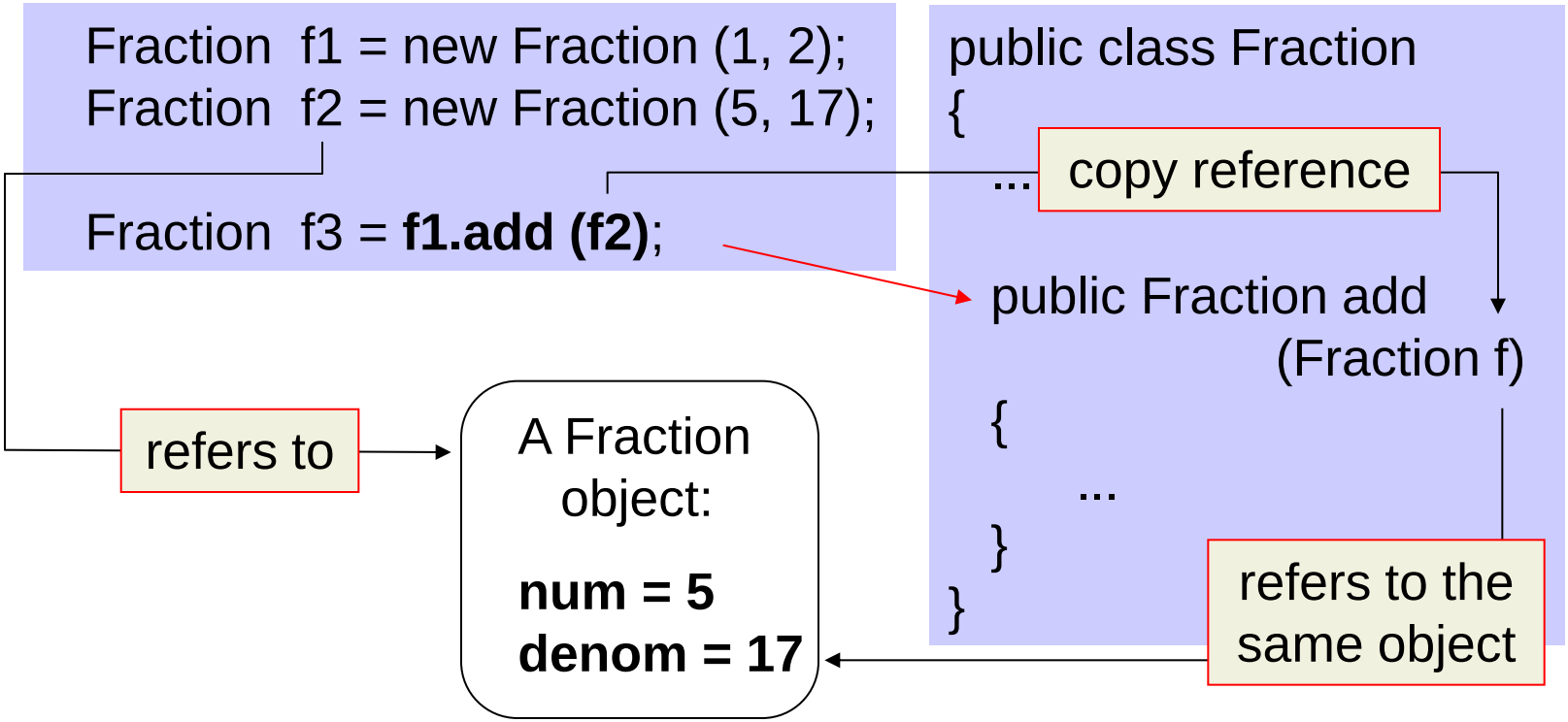
```
public class Test
{
   public double square (double x)
   {
      x *= x;
      return x;
   }

   public static void main(String[ ] args)
   {
      Test calc = new Test ();
      double x = 3.0;
      double y = calc.square (x);
      System.out.println (x + "  " + y);
   }
}
```

x here is a copy of the parameter passed to **square**. The copy is changed, but...

... the original **x** is unchanged.
Output: **3 9**

Passing Parameters (cont'd)

- Objects are always passed as references: the reference is copied, not the object.



Passing Parameters (cont'd)

- A method can change an object passed to it as a parameter (because the method gets a reference to the original object).
- A method can change the object for which it was called (this object acts like an implicit parameter):

```
panel.setBackround(Color.BLUE);
```

Passing Parameters (cont'd)

- Inside a method, this refers to the object for which the method was called. this can be passed to other constructors and methods as a parameter:

```
public class ChessGame
{
    ...
    Player player1 = new Player (this);
    ...
```

A reference to this **ChessGame** object

return Statement

- A method, unless void, returns a value of the specified type to the calling method.
- The return statement is used to immediately quit the method and return a value:

return *expression*;

The type of the return value or expression must match the method's declared return type.

return Statement (cont'd)

- A method can have several return statements; then all but one of them must be inside an if or else (or in a switch):

```
public someType myMethod (...)
{
    ...
    if (...)
        return <expression1>;
    else if (...)
        return <expression2>;
    ...
    return <expression3>;
}
```

return Statement (cont'd)

- A boolean method can return true, false, or the result of a boolean expression:

```
public boolean myMethod (...)
{
   ...
   if (...)
      return true;
   ...
   return n % 2 == 0;
}
```

return Statement (cont'd)

- A void method can use a return statement to quit the method early:

```
public void myMethod (...)
{
   ...
   if (...)
      return;
   ...
}
```

No need for a redundant **return** at the end

return Statement (cont'd)

- If its return type is a class, the method returns a reference to an object (or null).
- Often the returned object is created in the method using new. For example:

```
public Fraction inverse ()
{
    if (num == 0)
        return null;
    return new Fraction (denom, num);
}
```

- The returned object can also come from a parameter or from a call to another method.

Overloaded Methods

- Methods of the same class that have the same name but different numbers or types of parameters are called *overloaded* methods.
- Use overloaded methods when they perform similar tasks:

```
public void move (int x, int y)   { ... }
public void move (double x, double y)   { ... }
public void move (Point p)   { ... }

public Fraction add (int n)   { ... }
public Fraction add (Fraction other)   { ... }
```

Overloaded Methods (cont'd)

- The compiler treats overloaded methods as completely different methods.
- The compiler knows which one to call based on the number and the types of the parameters passed to the method.

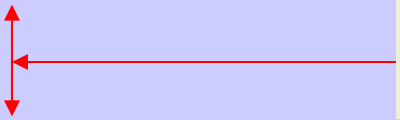
```
Circle circle =  new Circle(5);
circle.move (50, 100);
Point center =
            new Point(50, 100);
circle.move (center);
```

```
public class Circle
{
    public void move (int x, int y)
     { ... }

    public void move (Point p)
     { ... }
    ...
```

Overloaded Methods (cont'd)

- The return type alone is not sufficient for distinguishing between overloaded methods.

```
public class Circle
{
    public void move (int x, int y)
     { ... }

    public Point move (int x, int y)
     { ... }
    ...
```



Syntax error

Static Fields

- A *static field* (*class field* or *class variable*) is shared by all objects of the class.
- A non-static field (*instance field* or *instance variable*) belongs to an individual object.

Static Fields (cont'd)

- A static field can hold a constant shared by all objects of the class:

```
public class RollingDie
{
   private static final double slowDown = 0.97;
   private static final double speedFactor = 0.04;
   ...
```

Reserved words:
static
final

- A static field can be used to collect statistics or totals for all objects of the class.

Static Fields (cont'd)

- Static fields are stored with the class code, separately from instance variables that describe an individual object.
- Public static fields, usually global constants, are referred to in other classes using "dot notation": ClassName.constName

```
double area = Math.PI * r * r;
setBackground(Color.BLUE);
c.add(btn, BorderLayout.NORTH);
System.out.println(area);
```

Static Fields (cont'd)

- Usually static fields are NOT initialized in constructors (they are initialized either in declarations or in public static methods).
- If a class has only static fields, there is no point in creating objects of that class (all of them would be identical).
- Math and System are examples of the above. They have no public constructors and cannot be instantiated.

Static Methods

- Static methods can access and manipulate a class's static fields.
- Static methods cannot access non-static fields or call non-static methods of the class.
- Static methods are called using "dot notation": ClassName.statMethod(...)

```
double x = Math.random();
double y = Math.sqrt (x);
System.exit();
```

Static Methods (cont'd)

```
public class MyClass
{
    public static final int statConst;
    private static int statVar;

    private int instVar;
    ...
    public static int statMethod(...)
    {
        statVar = statConst;
        statMethod2(...);

        instVar = ...;
        instMethod(...);
    }
```

Static method

OK

Errors!

Static Methods (cont'd)

- main is static and therefore cannot access non-static fields or call non-static methods of its class:

```
public class Hello
{
   private int test () { ... }

   public static void main (String[ ] args)
   {
      System.out.println (test ());
   }
}
```

Error: non-static method **test** is called from static context (**main**)

Non-Static Methods

- A non-static method is called for a particular object using “dot notation”:

```
vendor.addMoney(25);
die1.roll();
```

- Non-static methods can access all fields and call all methods of their class — both static and non-static.

Review:

- Are you allowed to define a class with no constructors?
- Can a class have two different no-args constructors?
- What is the common Java style for method names?

Review (cont'd):

- Are parameters of primitive data types passed to methods "by value" or "by reference"?
- Can a method have more than one return statement?
- Can a method have no return statements?

Review (cont'd):

- When is it appropriate to define overloaded methods?
- Describe the difference between static and instance variables.
- What is the syntax for referring to a public static field outside the class?

Review (cont'd):

- Can non-static methods access static fields?
- Can static methods access instance variables?
- Can static methods have local variables?

```
public class Fraction
{
   int numerator;
   int denominator;
   public Fraction (int nume, int denom)
   {
      numerator = nume;
      denominator = denom;
   }
   public int getNumerator ()
   {
      return numerator;
   }

   public int getDenumerator ()

   {
      return denominator;
   }
   public String toString ()
   {
      return (numerator + " / " +
denominator);
   }


}
```

```
public class FractionTester {

public static void main (String[] args)
   {
      Fraction myFraction;
      myFraction = new Fraction (3, 4);
      int numerator = myFraction.getNumerator ();
      int denumerator=myFraction.getDenumerator();
      System.out.println ("Numenator Value: " + numerator);
      System.out.println("Denumerator value: "+denumerator);
      System.out.println ("Fraction: " + myFraction);
   }

}
```

```
public class Students
{
    static int count = 0;
    public Students ()  // constructor
    {
        count = count + 1;
    }

    public static int numberOfStudents ()
    {
        return count;
    }

    public static void main (String[] args)
    {
        Student Ali = new Student ();
        Student Jesi = new Student ();
        Student Fatima = new Student ();
        int totalStudents;
        totalStudents = student.numberOfStudents ();
        System.out.println (totalStudents);
    }
} // class students
```