



Introduction to Classes and Objects

David Greenstein
Monta Vista High School

Client Class

- A client class is one that constructs and uses objects of another class.

B is a “client” of A

```
public class A {  
    private int field1;  
  
    public A() {}  
    ...  
    public void aMethod1() {}  
    ...  
    private void aMethod2() {}  
    ...  
}
```

```
public class B {  
    ...  
    public void bMethod1() {  
        A s = new A();  
        ...  
        s.aMethod1();  
    }  
    ...  
}
```

**B only has access to A's
public constructors and methods**

Public vs. Private

- **Public constructors** and **methods** of a class are its **interface** with classes that use it (e.g. its clients).
- All fields are usually declared **private** and **hidden** from clients.
- **Constants** in a class are designated ***private final***.
- In some rare cases, a constant is universal and it is made ***public static final***. (e.g. Math.PI, Math.E)
- “Helper” methods that are needed only inside the class are declared **private**.

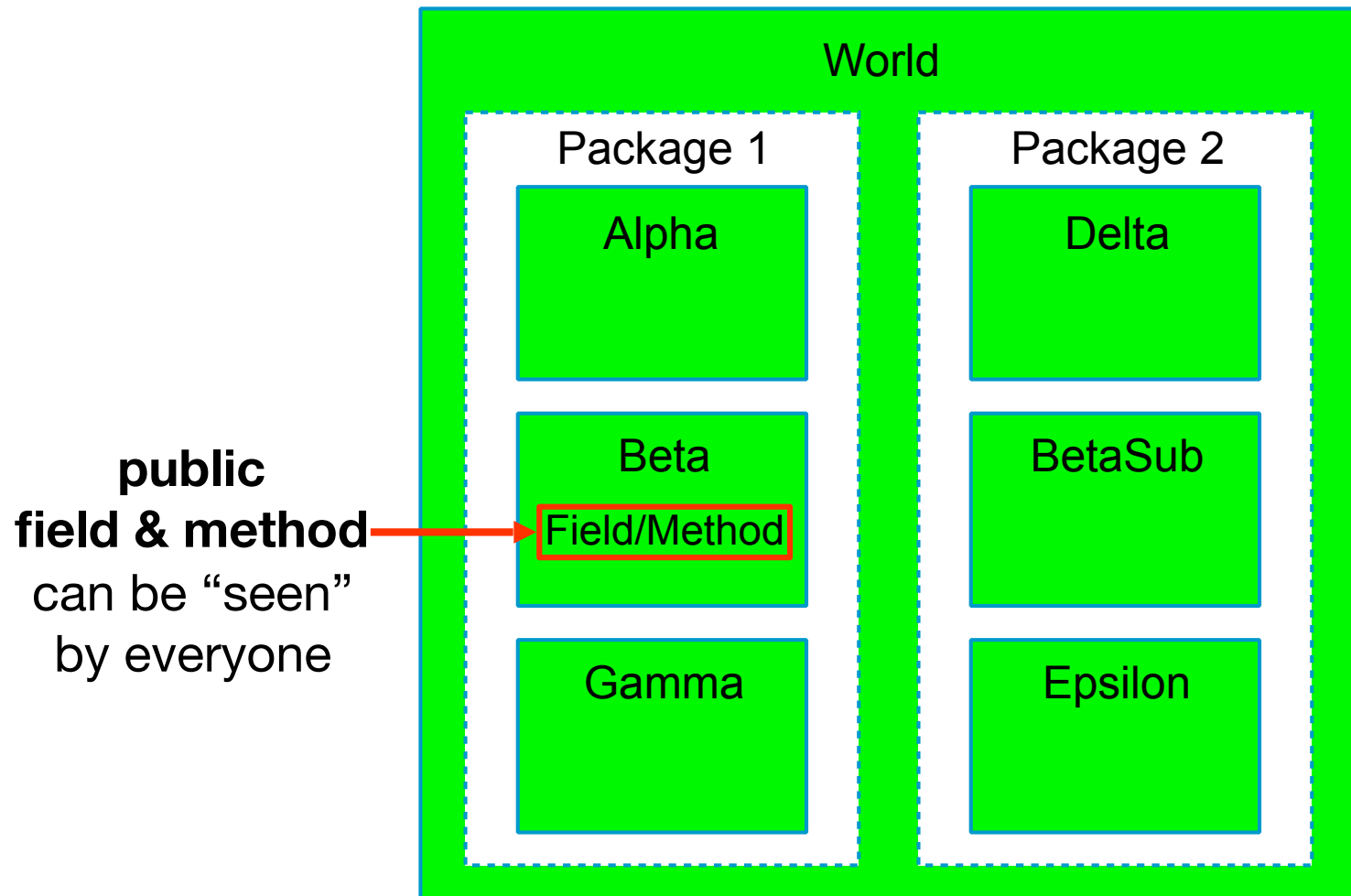
Public vs. Private (cont)

- Private **constructors** are used when the only client is the class itself.
- A **private field** is accessible anywhere within the class's 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;
        ...
    }
}
```

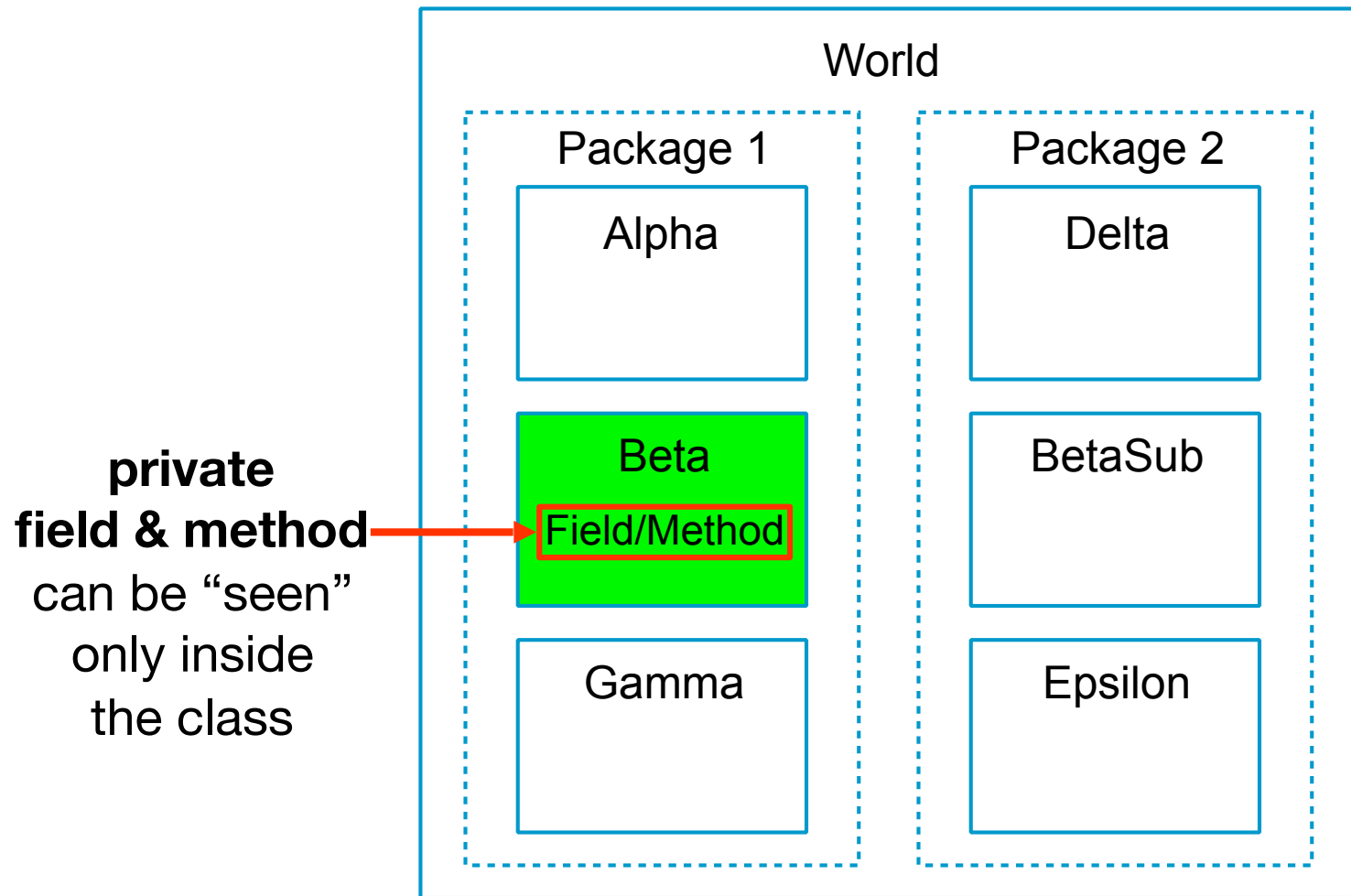
Controlling Access in Java

- **public** modifier



Controlling Access in Java

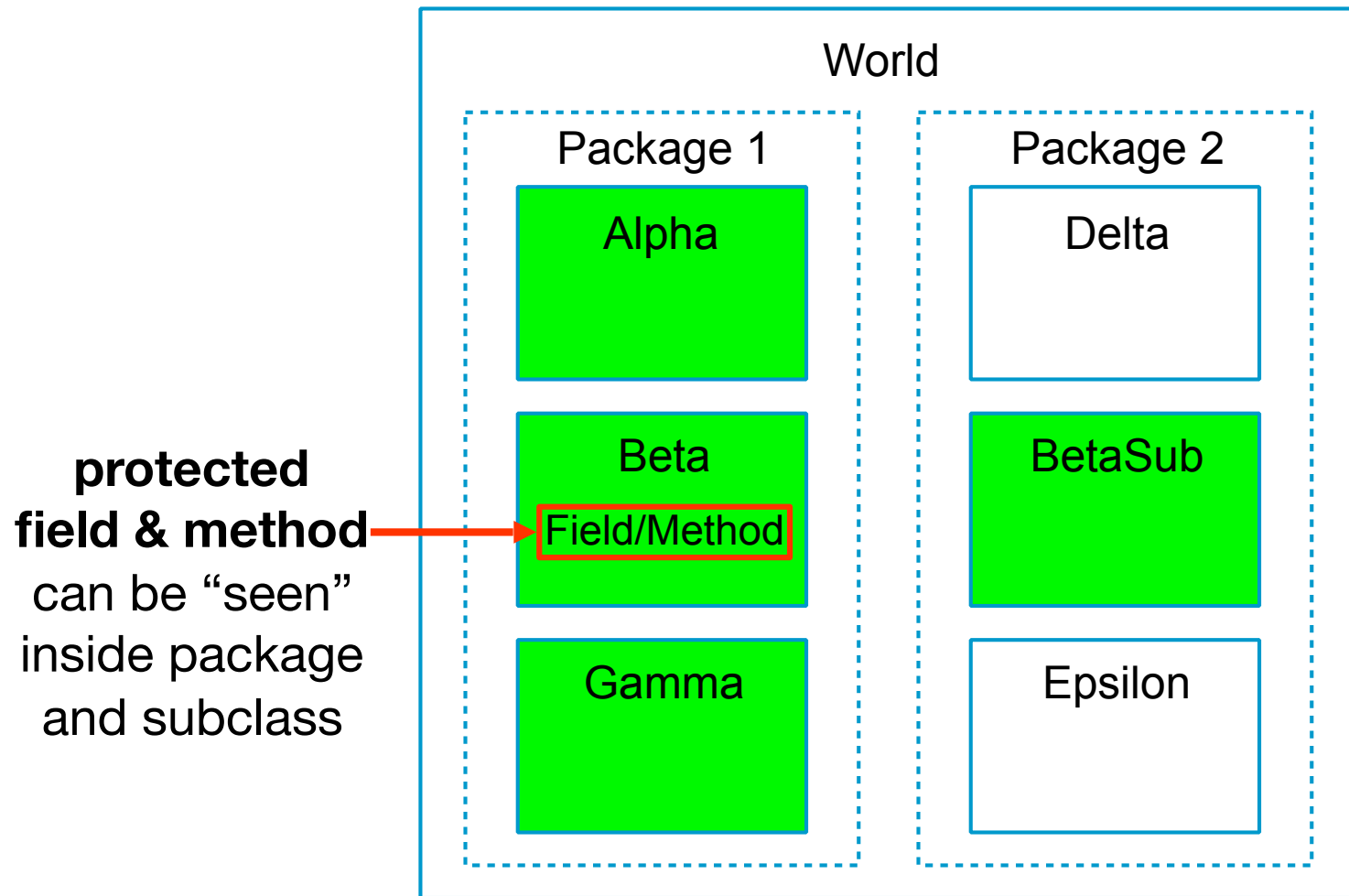
- **private** modifier



private
field & method
can be "seen"
only inside
the class

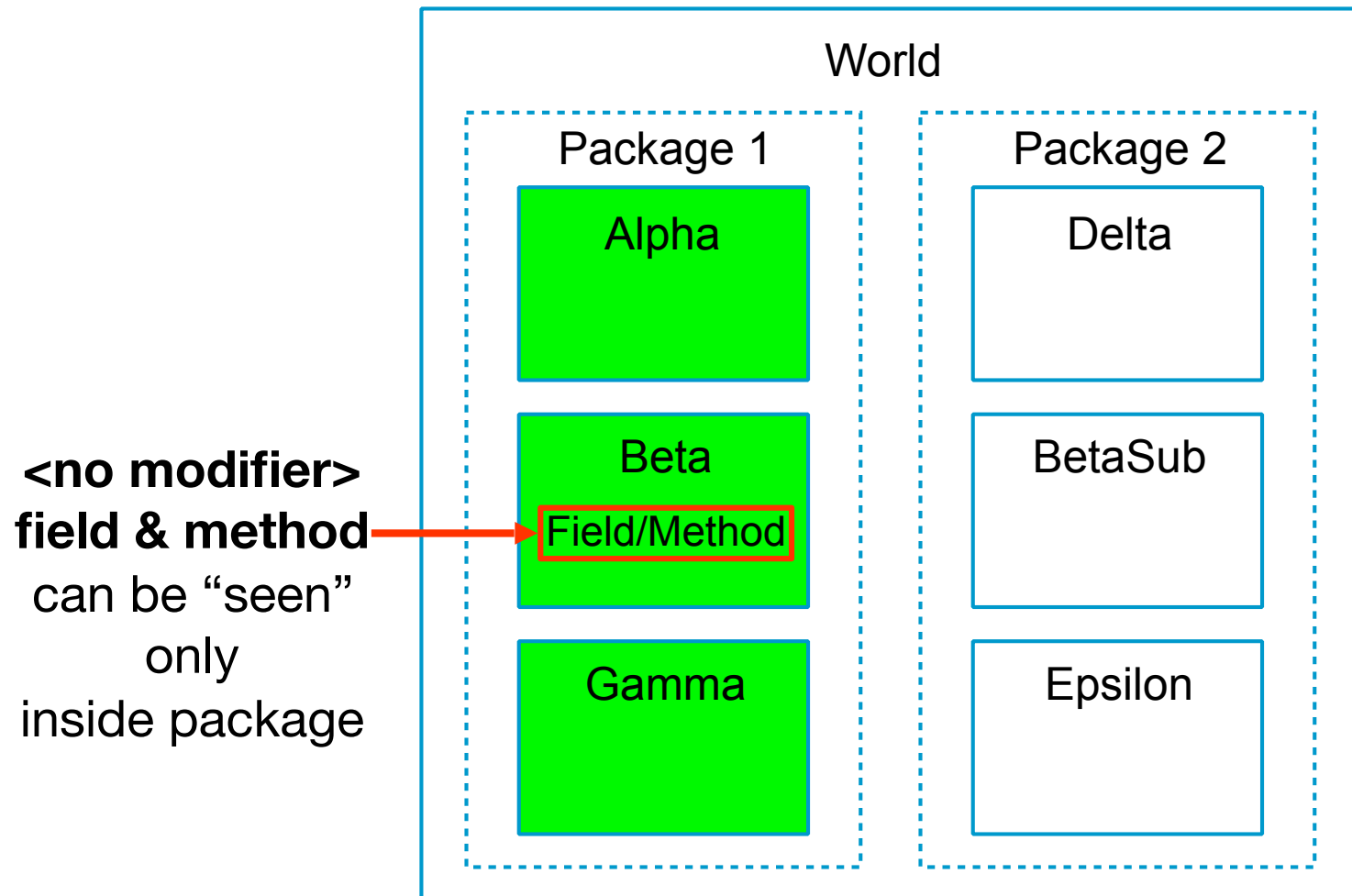
Controlling Access in Java

- **protected** modifier



Controlling Access in Java

- no modifier



Controlling Access in Java

- Field/Method Access Summary

| <i>Modifier</i> | <i>Class</i> | <i>Package</i> | <i>Subclass</i> | <i>World</i> |
|--------------------|--------------|----------------|-----------------|--------------|
| <i>public</i> | <i>Y</i> | <i>Y</i> | <i>Y</i> | <i>Y</i> |
| <i>protected</i> | <i>Y</i> | <i>Y</i> | <i>Y</i> | <i>N</i> |
| <i>no modifier</i> | <i>Y</i> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <i>private</i> | <i>Y</i> | <i>N</i> | <i>N</i> | <i>N</i> |

- The AP Exam (and this class) only use **public** and **private** modifiers.

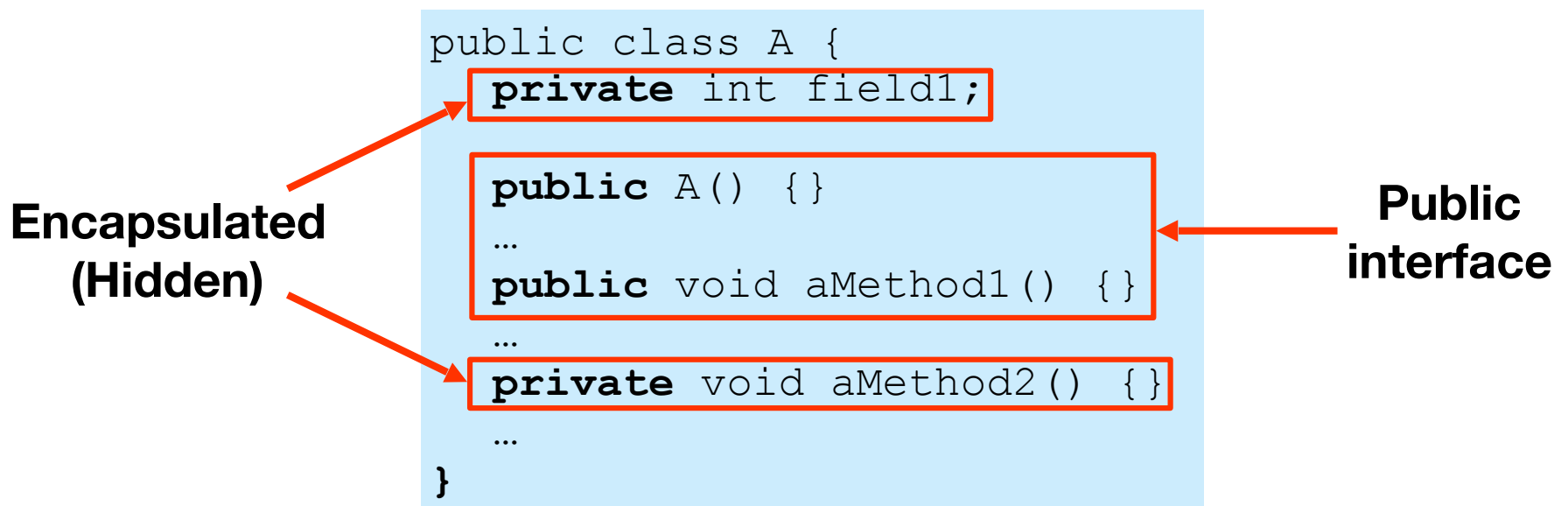
Accessors and Modifiers

- 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**.
- **Modifiers'** names often start with **set**.

```
public class Fraction
{
    private int num, denom;
    ...
    public int getNum() { return num; }
    public void setNum(int n) { num = n; }
    ...
}
```

Encapsulation

- **Hiding** the implementation details of a class is called **encapsulation**. (e.g. making all fields and helper methods **private**)
- 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**.



Constructors

- A **constructor** is a procedure for creating objects of the class. It is different than a method.
- Most constructors are **public**.
- 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 **name of the class**.
- Constructors may take **parameters**.

```
public class A {  
  
    public A() { ... }  
    public A(int a, String b)  
    { ... }  
    ...  
}
```

Constructors (cont)

- If a class has more than one constructor, they must have a different *signature*.
- Programmers often provide a “**no-args**” constructor that takes no parameters.
- If a programmer does not define any constructors, Java provides one default no-args constructor, which **allocates memory** and **sets fields to the default values**: numbers to zero, objects to null, boolean to false, and char to null (0) character.

```
public class A {  
  
    public A() { ... }  
    public A(int a, String b)  
    { ... }  
    ...  
}
```

Constructors (cont)

```
public class Fraction
{
    private int num, denom;
    public Fraction ( )
    {
        num = 0;
        denom = 1;
    }
    public Fraction (int n)
    {
        num = n;
        denom = 1;
    }
    // Continued ->>
```

**no-args
constructor**

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

copy constructor

Constructors (cont)

- A nasty bug!!!

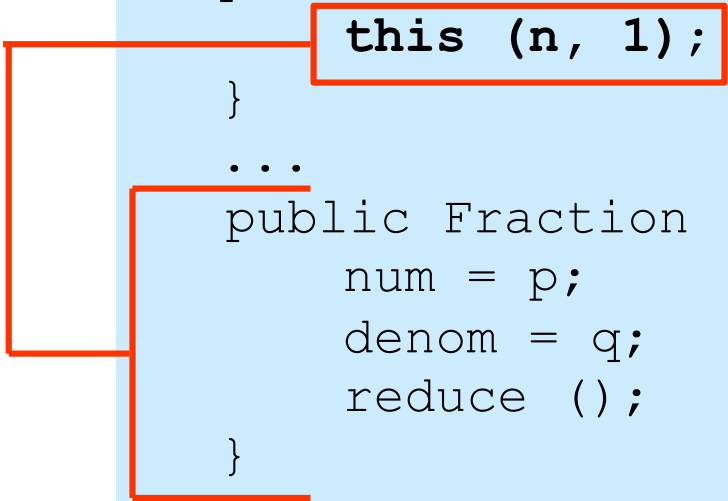
```
public class MyWindow extends JFrame
{
    ...
    // Constructor:
    public void MyWindow ( )
    {
        ...
    }
    ...
}
```

Compiles fine, but the compiler thinks this is a method and uses **MyWindow's** default ***no-args*** constructor instead!!

Constructors (cont)

- Constructors of a class can call each other using the keyword **this**.
- Using **this** is a good way to avoid duplicating code, and it makes it easier to maintain. You only need to change one constructor so both are changed.

```
public class Fraction {  
    ...  
    public Fraction (int n) {  
        this (n, 1);  
    }  
    ...  
    public Fraction (int p, int q) {  
        num = p;  
        denom = q;  
        reduce ();  
    }  
    ...  
}
```

A diagram consisting of red lines. A box highlights the line `this (n, 1);` in the first constructor. A line extends from the box to the left, then turns down and then right to point to the opening curly brace of the second constructor, `public Fraction (int p, int q) {`. This indicates that the first constructor calls the second one.

new Operator

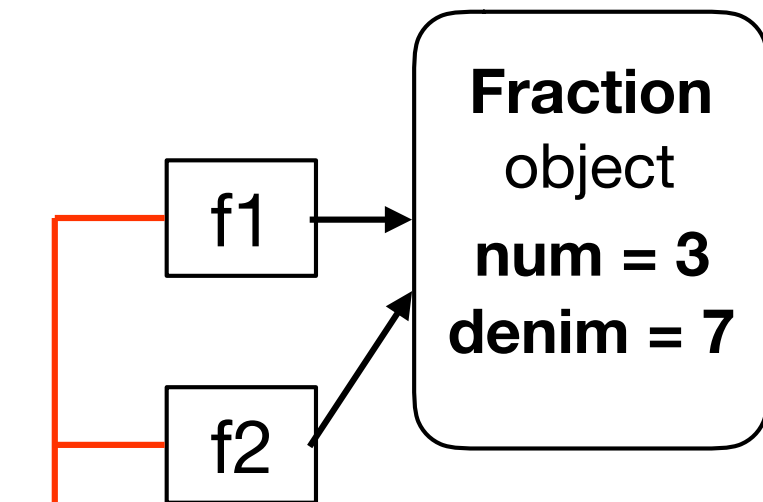
- Constructors are invoked when using the **new** operator.
- Parameters passed by the **new** operator must match the number, types, and order of parameters expected by one of the constructors.

```
Fraction f1 = new Fraction(2);  
...  
Fraction f2 = new Fraction(3,6);
```

```
public class Fraction {  
    ...  
    public Fraction (int n)  
    {  
        this(n, 1);  
    }  
    ..  
    public Fraction (int n, int d)  
    {  
        num = n;  
        denom = d;  
    }  
    ...  
}
```

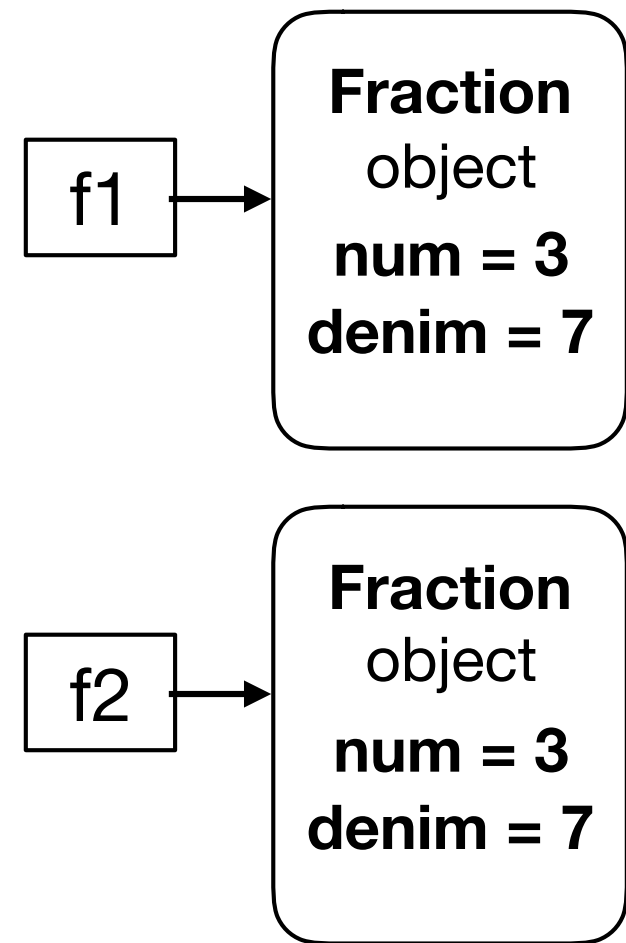
References to Objects

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

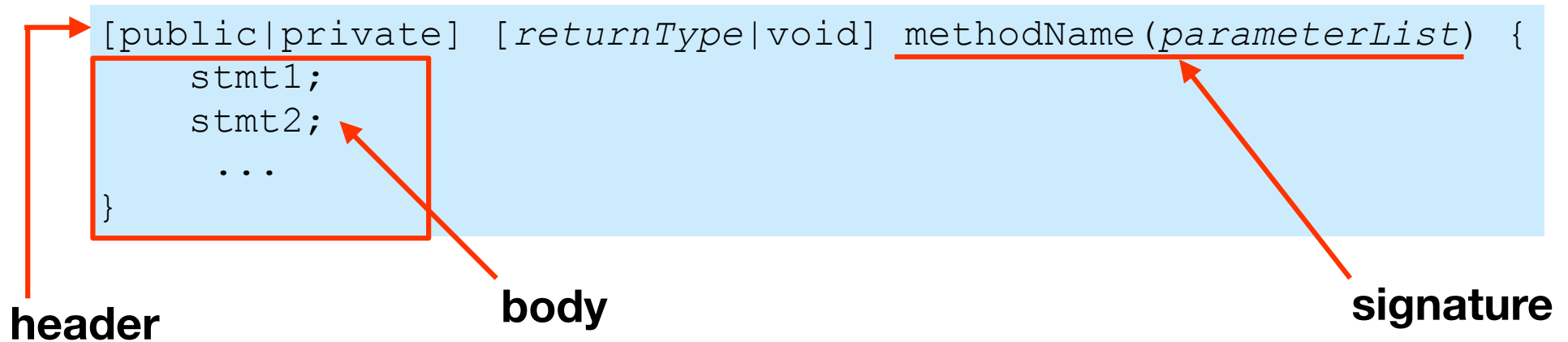


Refer to the same object

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



Methods



- A method is always defined inside a class.
- Methods used by client classes are **public**.
- “Helper” methods only used inside the class are **private**.
- Style:
 - Method names start with a lowercase letter.
 - Method names are “verb-like”.

Passing Parameters

- A parameter is something passed with a method call.
- Any expression that has an appropriate data type can serve as a parameter.
- Methods can return one primitive or object.
- A “smaller” type can be promoted to a “larger” type.

Calling a method

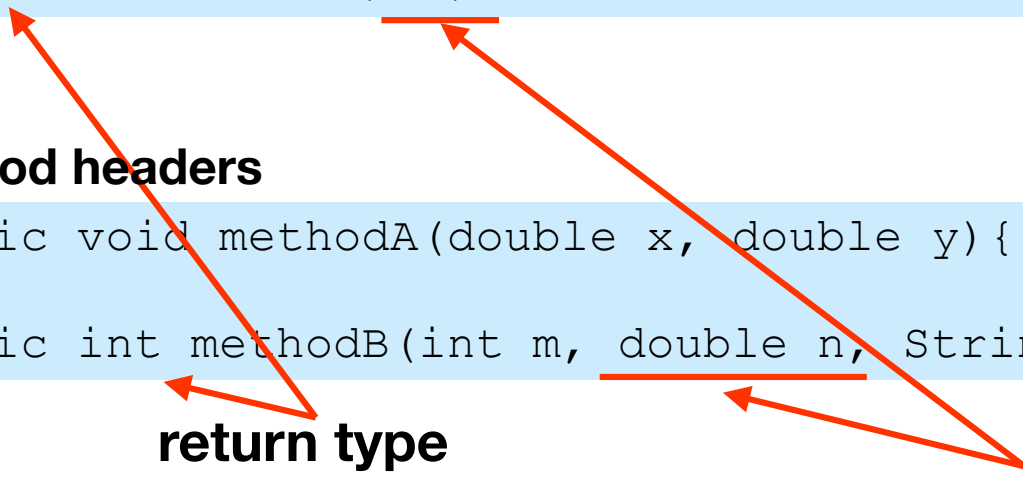
```
methodA(3.2, Math.PI);  
int a = methodB(2, 3, "hello");
```

Method headers

```
public void methodA(double x, double y) { ... }  
public int methodB(int m, double n, String s) { ... }
```

return type

promoted type



Pass by Value

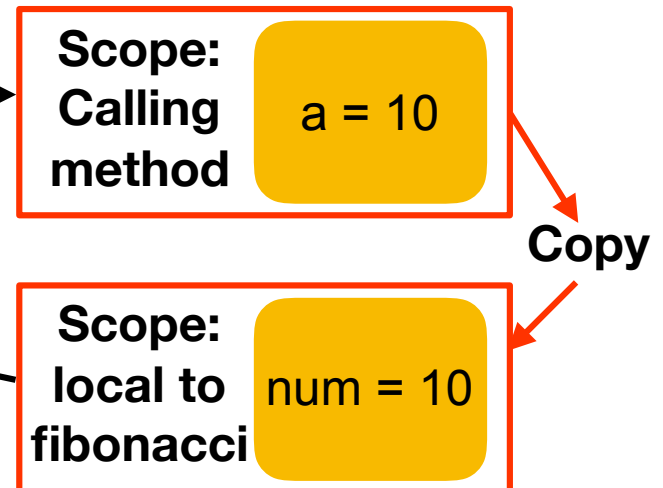
- Primitive data type parameters are always “pass by value”. A copy of the value is made of the parameter.
- In this example, **num** changes in the **fibonacci()** method but the original variable **a** does not change.

Client method

```
int a = 10;  
int b = fibonacci(a);
```

Method

```
public int fibonacci(int num) {  
    int start = num;  
    ...  
    return start;  
}
```



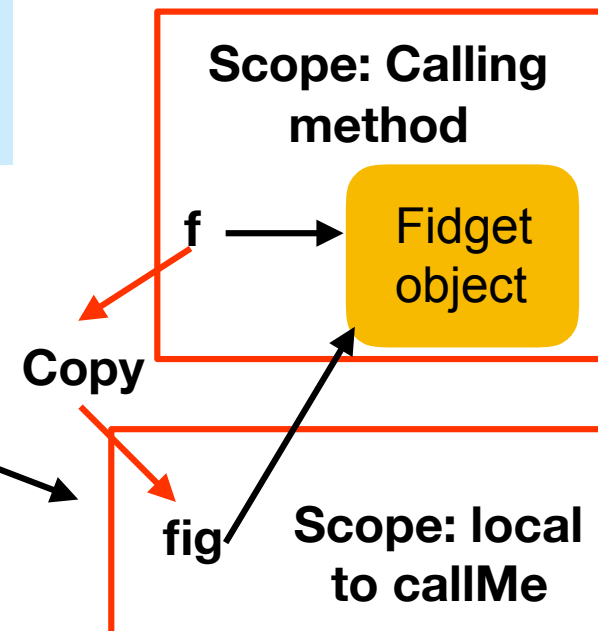
Pass by “Reference” (Value)

- Objects are always passed as references: the reference (object address) is copied, not the object.
- In this example, any changes to the **fig** object is reflected in the original **f** object.

Client method

```
Fidget f = new Fidget();  
AnotherClass ac = new AnotherClass();  
ac.callMe(f);
```

```
public class AnotherClass {  
    ...  
    public void callMe(Fidget fig) {  
        ...  
    }  
}
```



“this” Object

- 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 Yahtzee {  
    ...  
    Player p1 = new Player(this);  
    ...  
}
```

Refers to
this Yahtzee
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, if not **void**, return a value or **null**.

```
public Dice getDice() {  
    ...  
    return myDice;  
    ...  
}
```

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

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.

Math class - different parameter types

```
public static int abs(int num) { ... }  
public static double abs(double num) { ... }
```

String class - different number of parameters

```
public static String substring(int num) { ... }  
public static String substring(int start, int stop) { ... }
```

Overloaded Methods (cont)

- 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)

- The return type alone is not sufficient for making a distinction between overloaded methods.

**Syntax
Error**

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

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

Static Fields

- A **static** field (a.k.a. class field or class variable) is shared by all objects of the class.
 - Used for constants across classes.
 - Used to collect statistics or totals of all classes.
- A **non-static** field (a.k.a. instance field or instance variable) belongs to an individual object.
- Public static fields, usually global constants, are referred to in other classes using “dot notation”: `ClassName.constName`

```
public class Die
{
    public static int DEFAULT_SIDES = 6;
    ...
}
```

```
for (int a = 0; a < Die.DEFAULT_SIDES; a++)
    ...
```

Static Fields (cont)

- 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** classes are examples of the above. They have no public constructors and cannot be instantiated.

```
public class Die
{
    public static int DEFAULT_SIDES = 6;
    ...
}
```

Static Methods

- **Static methods** can access and manipulate a class's **static** fields.
- **Static methods** are called using “dot notation”:
ClassName.statMethod(...)

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

Static Methods (cont)

- Static methods cannot access non-static fields or call non-static methods of the class.

```
public class MyClass
{
    public static final int staticConst;
    private static int staticVar;

    private int instanceVar;
    ...
    public static void main(String[] args)
    {
        staticVar = staticConst;
        staticMethod2(...);

        instanceVar = ...;
        instanceMethod(...);
    }
    public void instanceMethod() { ... }
}
```

Access
applies to main
and all static
methods

Okay

ERROR!

Non-Static Methods

- A **non-static method** is called for a particular object using “dot notation” bound to an instance.

```
Die d1 = new Die();  
d1.roll();  
int a = d1.getValue();
```

- **Non-static methods** can access all fields and call all methods of their class, both static and non-static!

```
public class Die {  
    private int value;  
    private static final int DEFAULT_SIDES = 6;  
    public void roll() { ... }  
    ...  
    public static void staticMethod(int num) { ... }  
    ...  
    public int rollAndGetValue() {  
        roll();  
        staticMethod(DEFAULT_SIDES);  
        return value;  
    }  
}
```

All Okay!!



Questions?