# Object-Oriented Languages

- Chapter 12

# Classes

- A **class** is a blueprint for creating objects

```
class Rectangle {
    public double height, width;
    public double area() {
        return height * width;
    }
}
```

  - This is Java code; the equivalent C++ code is very similar
- **Class members**: methods and fields

# Objects

- The central concept of "object-oriented" programming
- In C++ and Java, they are **instances** of classes, created through **new**
  - e.g. when expression **new Rectangle()** is evaluated, a new object is created
  - "instance" = "object"
  - "class X is instantiated" = "an instance of X is created"

# References (Pointers) to Objects

- Objects are manipulated indirectly through object references (pointers)
- **Java: Rectangle x; x = new Rectangle();**
  - **x** is a variable of type "reference to Rectangle objects"
- **C++: Rectangle* x; x = new Rectangle();**
  - **x** is a variable of type "pointer to Rectangle objects"

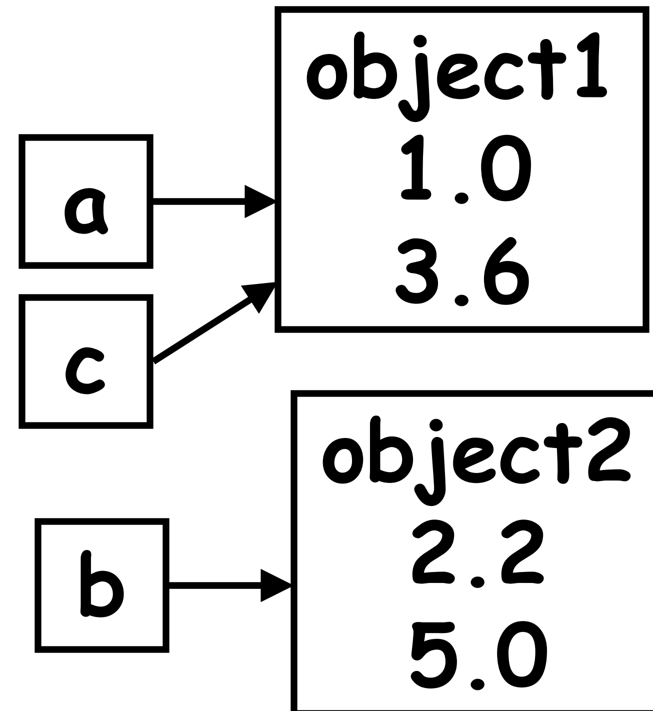
# Creation of Objects

- During the evaluation of **new Rectangle()**
  - A new instance of Rectangle is created
  - A reference (pointer) to this instance is "returned"
  - ***x*** is assigned this reference (pointer) value
    - e.g. the value may be the address of the first byte of the object's memory
    - or the value may be some internal "handle" to the actual object (e.g., index in some internal table, which itself contains the address of the first byte)

# Members: Methods and Fields

- Two separate kinds: **instance members** and **static members**
  - Instance members: each instance of the class has a separate copy

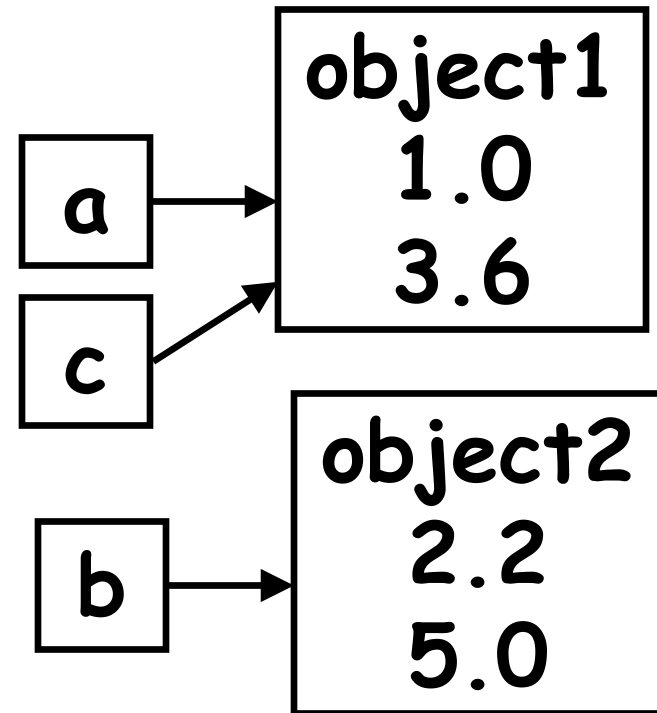
```
Rectangle a, b, c;
a = new Rectangle();
b = new Rectangle();
a.height = 1.0; a.width = 3.6;
b.height = 2.2; b.width = 5.0;
c = a;
```



# Members: Methods and Fields

- Same code in C++
  - x->field vs. x.field in Java

```
Rectangle  *a, *b, *c;
a = new Rectangle();
b = new Rectangle();
a->height = 1.0;
a->width = 3.6;
b->height = 2.2;
b->width = 5.0;
c = a;
```



# Instance Methods

- An instance method operates on objects
  - "method m is invoked on the object"

```
double area() { return height*width;}
```

in reality, this is syntactic sugar over

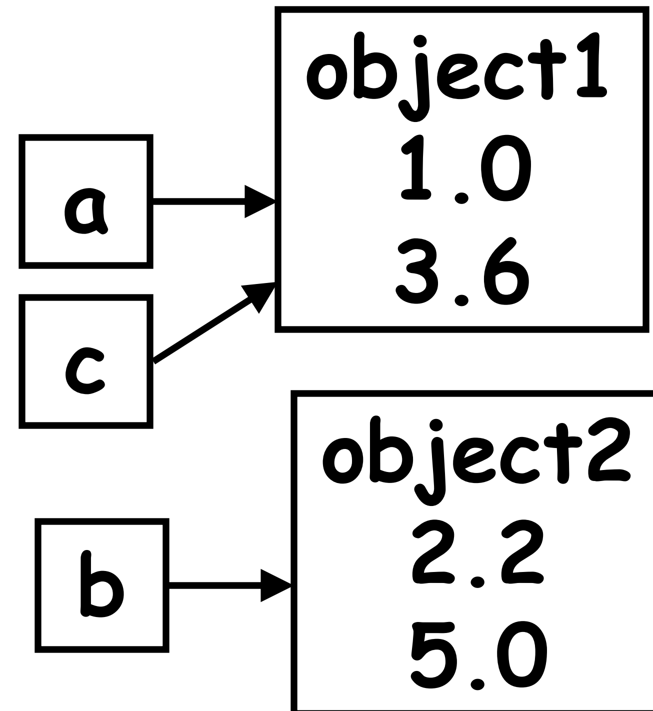
```
double area(Rectangle this) { // Java
      return this.height * this.width; }
double area(Rectangle* this) { // C++
      return this->height * this->width; }
```

- There is an implicit formal parameter this which is **a reference to the object on which the method was invoked**

# Members: Methods and Fields

- Calling an instance method: there is an object on which we are calling it
  - x.m() in Java, x->m() in C++

```
Rectangle  *a, *b, *c;
a = new Rectangle();
a->height = 1.0;
a->width = 3.6;
c = a;
double result = c->area();
```



# Method Overloading (Java)

- A class may have more than one method with the same name
  - the name is "overloaded"
- All overloaded methods must have different signatures
  - Signature: method name + types of formal parameters
- double area() { … } → signature **area()**
- double area(int precision_level) { … } → signature **area(int)**

# Constructors

- Constructors are used to set up the initial state of new objects

```
public Rectangle(double height,double width) {
    this.height = height; this.width=width; }
```

- **x = new Rectangle(1.1, 2.3);**
  - A new object is created: with default values 0.0 in Java, and undefined values in C++
    - The constructor is invoked on this object; the fields are initialized with 1.1 and 2.3
  - A reference to the object is assigned to x

# Compilation Model of Java (start detour)

- The compiler takes as input source code
  - Sun provides a standard compiler; others can build their own compilers if they want
  - Typically, class *A* is stored in file *A*.java
    - Exception: nested classes
- Compiler output: Java bytecode
  - *A*.java -> *A*.class
  - A standardized platform-independent representation of Java code
  - Essentially, a programming language that is understood by the Java Virtual Machine

# Rectangle.class

```
class Rectangle extends java.lang.Object {
    public double h;  public double w;
    Rectangle();
    public double area();
}
Rectangle()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
double area()
   0 aload_0
   1 getfield #4 <Field double h>
   4 aload_0
   5 getfield #5 <Field double w>
   8 dmul
   9 dreturn
```

# Execution Model

- Java bytecode is executed by a Java Virtual Machine (JVM)
  - Sun provides several kinds of JVMs for various platforms (e.g. Solaris, Wintel, etc.)
  - Several other vendors for JVMs
    - e.g. IBM sells a JVM that is performance-tuned for enterprise server applications
- Platform independence: as long as there are JVMs available, the exact same Java bytecode can be executed anywhere

# JVM

- There are two ways to execute the bytecode
- Interpretation: the VM just executes each bytecode instruction itself
  - Initial JVMs used this model
- Compilation: the VM uses its own internal compiler to translate bytecode to native code for the platform
  - The native code is executed by the platform
  - Faster than interpretation

# Compilation inside a VM (end detour)

- Just-in-time: the first time some bytecode needs to be executed, it is compiled to native code on the fly
  - Typically done at method level: the first time a method is invoked, the compiler kicks in
  - Problems: compilation has overhead, and the overall running time may actually increase
- Profile-driven compilation
  - Start executing through interpretation, but track "hot spots" (e.g. frequently executed methods), and at some point compile them

# Inheritance

- class B extends A { … }
  - Single inheritance: only one superclass (Java)
- class B : public A, Foo, Bar { … }
  - Multiple inheritance: several superclasses (C++)
- Every member of A is inherited by B
  - if a field **f** is defined in A, every object of class B has an **f** field
  - if a method **m** is defined in A, this method can be invoked on an object of class B
- B may declare new members

# Example

```
class Rectangle {
   private double h,w;
   public Rectangle(double h,double w) { … }
   public double getHeight() { return h; }
   public double getWidth() { return w; }
   public double area() { … } }

class SwissRectangle externs Rectangle {
   private int hole_size;
   public SwissRectangle(double h,double w,
                                  int hs) { … }
   public void shrinkHole() { hole_size--; }
   public double area() { … } // overridden }
```

# Constructors and Inheritance

- Constructors are not inherited
- A constructor in a subclass Y must invoke a constructor in the superclass X
  - (this is a bit of an oversimplification)
- The constructor of superclass X initializes the part of the "object state" that is declared in X
  - i.e., does something with the fields declared in X and inherited by the subclasses

# Inheritance of Methods

- If a subclass declares a method with the same name but a different signature, we have overloading
  - Either method can be invoked on an instance of the subclass
- If a subclass declares a method with the same signature, we have overriding
  - Only the new method applies to instances of the subclass

# Polymorphism of References

- Reference variables for A objects also may points to B objects
  - **A v = new B()** in Java; **A* v = new B()** in C++
- Earlier we said that the type of v is pointer (reference) to instances of A
- Correct definition: pointer to instances of A or instances of any subclass of A
  - If C is a subclass of B, variable v can point to instances of C
  - Poly (many) morph (form) ism

# Method Invocation – Compile Time

- What happens when we have a method invocation of the form **x.m(a,b)**?
- Two very different things are done
  - At compile time, by the Java compiler (javac)
  - At run time, by the Java Virtual Machine
- At compile time, a target method is associated with the invocation expression
  - "compile-time target", "static target"
  - The static target is based on the declared type of x

# Method Invocation – Compile Time

```
class A { void m(int p, int q) {…} … }
class B extends A { void m(int r, int s) {…} … }
A x;
x = new B();
x.m(1,2);
```

- Since x has declared type *A*, the compile-time target is method m in class *A*
- javac encodes this in the bytecode (foo.class)
  - **virtualinvoke x,<A: void m(int,int)>**

# Method Invocation - Run Time

- The Java virtual machine loads the bytecode and starts executing it
- When it tries to execute instruction **virtualinvoke x,<A: void m(int,int)>**
  - Looks at <u>the class Z of the object that x refers to</u> at that particular moment
  - Searches Z for a method with signature **m(int,int)** and return type **void**
  - If Z doesn't have it, goes to Z's superclass, and so on upwards, until a match is found

# Method Invocation – Run Time

- The run-time (dynamic) target: "lowest" method that matches the signature and the return type of the static target
  - "Lowest" with respect to the inheritance chain from Z to java.lang.Object
- Once the JVM determines the run-time target method, it invokes it on the object that is pointed-to by x
- "virtual dispatch", "method lookup"

# Virtual Methods in C++

```
class A { virtual void m(int p, int q) {…} … }
class B : public A
        { virtual void m(int r, int s) {…} … }
A* x;
x = new B();
x->m(1,2);
```

- Since x has declared type A*, the compile-time target is method m in class A
- The run-time target is m in B
  - Without the keyword "virtual", the run-time target will be the same as the compile-time target

# Terminology

- Invocation x.m(a,b) sends a message m to the object referred to by x
  - This object is the receiver object, and its class is the receiver class
  - The method that contains x.m(a,b) belongs to the sender object
- Dynamic binding (virtual dispatch): mapping the message to a method
- Polymorphic call: more than one possible run-time target

# Abstract Classes and Methods

- Abstract class: instances of it cannot be created
  - Only instances of its subclasses
- Abstract methods
  - No code: just name, parameter types, and return type
  - Abstract methods must be overridden in subclasses, by non-abstract methods

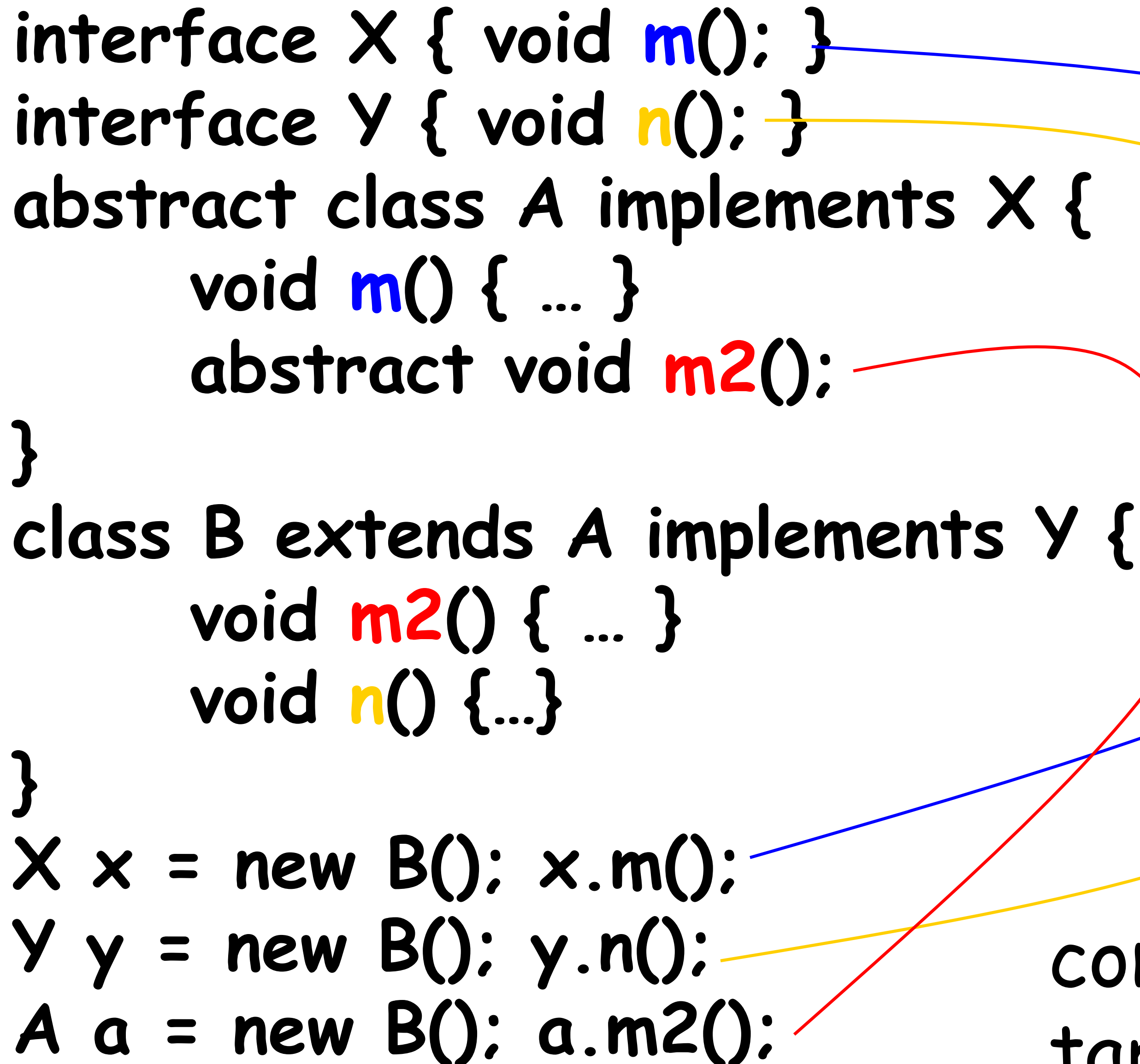
# Abstract Classes

- Abstract class: class that contains abstract methods
  - abstract void m(int x); // Java
  - virtual void m(int x) = 0; // C++
- Why can't we say **new X()** if X is abstract?
- An abstract method can be the compile-time target of a method call
  - But not the run-time target, obviously

# Interfaces in Java

- Very similar to abstract classes in which all methods are abstract
- A Java class has only one superclass, but can implement many interfaces
  - **class Y extends X implements A, B { … }**
- A reference variable can be of interface type, and can refer to any instance of a class that implements the interface
- An interface method can be the compile-time target of a method call

# Example




compile-time targets

# Static Methods and Fields

- Static field: a single copy for the entire class
- Static method: **not** invoked on an object
  - Just like a regular procedure (function) in a procedural language (e.g. C, Pascal, etc.)
- Terminology
  - "static method/field" = "class method/field"
  - "instance method/field" = "non-static method/field"

# Classic Example (Java)

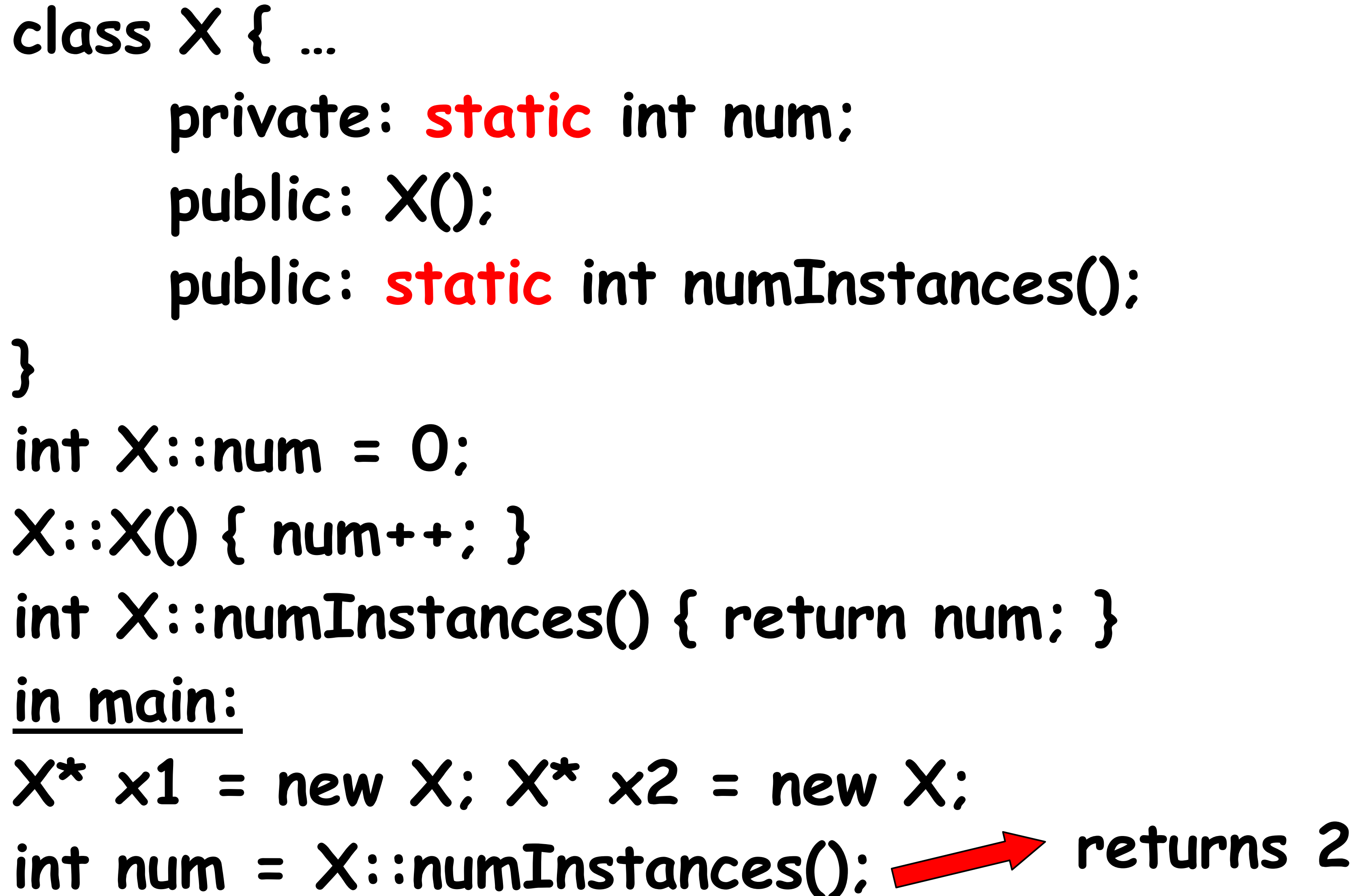
```
class X { …
      private static int num = 0;
      // constructor
      public X() { num++; }
      public static int numInstances()
            { return num; }
}
```

in main:

```
X x1 = new X(); X x2 = new X();
int n = X.numInstances();
```

returns 2

# Classic Example (C++)



# Example: Singleton Pattern (Java)

```
class Logger {
  private Logger() { }
  private static Logger instance = null;
  public static Logger getInstance() {
      if (instance == null)
            instance = new Logger();
      return instance;
  }
}
client code: Logger.getInstance().writeLog(…)
```