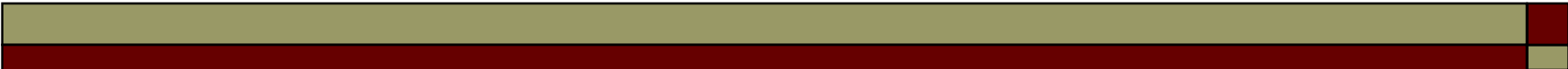


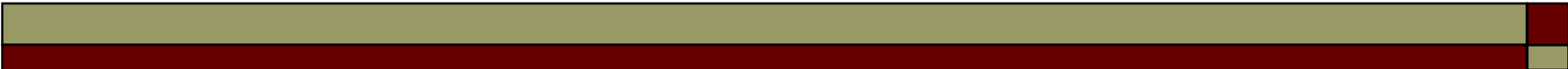
Access Control & Reuse

Slides adapted from Prof. Steven Roehrig
and Sakir YUCEL (CMU)

- 
-
- ❑ Implementation hiding with packages and access specifiers.
 - ❑ Composition
 - ❑ Inheritance
 - ❑ More on constructors
 - ❑ Finals
 - ❑ Class loading

Access Specifiers

- ❑ **public, protected, private** and “friendly”
- ❑ We haven’t used these yet, except for **main()** and **toString()**.
- ❑ **main()** needs to be **public**, so the runtime system can call it.
- ❑ **toString()** needs to be **public** since it is **public** in **Object**, and we are “overriding” it.



The “Need To Know” Principle

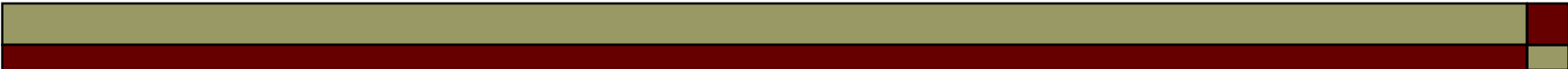
- Like military secrets, portions of your classes are best kept private.
- The public interface of a class should provide everything users need, but nothing they don't.
- Access specifiers allow you to enforce the “need to know” principle.

Access Specifiers

- ❑ **public** members (variables and methods) are freely available for anyone's use.
- ❑ **private** members can be accessed (used) only by the methods of the containing class.
- ❑ **protected** members are public to subclasses, private to others.
- ❑ “Friendly” members have *package access*:
 - no access specifier
 - public within the containing package

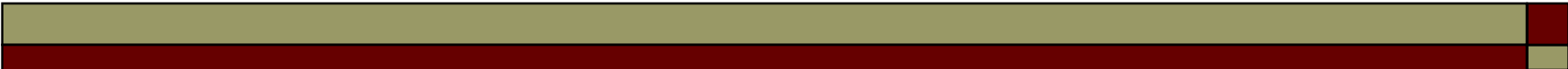
Packages

- ❑ Java's concept of "module".
- ❑ A group of related classes.
- ❑ A package can have a name, but there is the unnamed package, which holds all the classes in your program that you haven't put into a named package.



How Does It All Work?

- So far, we haven't used packages and access specifiers. Why has this worked?
 - We kept all our **.class** files in the same folder; they are in the unnamed package.
 - All of our members were therefore friendly.
 - Only methods that are called from another package need access specifiers.
 - Make sure you have the current directory ('.') in your classpath.



The Basic Rules

- ❑ Class members should be **private** unless there is a need to know or use.
- ❑ Think carefully about the **public** interface.
- ❑ Use accessors/mutators (aka **get** and **set** methods) to control access to private member variables.
- ❑ Often we create methods that are only used internally; make these **private**.
- ❑ We'll worry about **protected** later.

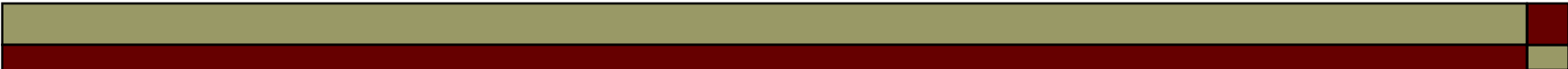
Example

```
public class Fraction {  
    public Fraction()  
    public Fraction(int n, int d)  
    public String toString()  
    public String toDecimal()  
    public Fraction add(Fraction f)  
    private int numerator;  
    private int denominator;  
    private int gcd(int a, int b)  
}
```

How To Change A Fraction?

- ❑ This is a design decision.
- ❑ Some classes are “immutable” for good (or bad!) reasons. **String** is an example.
- ❑ If we want users to change a **Fraction** object’s values, provide a “set” function:

```
public void set(int n, int d) {  
    // test carefully for suitability, then:  
    numerator = n;  
    denominator = d;  
}
```



Interface vs. Implementation

- For flexibility, we want the right to change an implementation if we find a better one.
- But we don't want to break client code.
- Access specifiers restrict what clients can rely on.
- **Everything marked private is subject to change.**

Example: NNCollection

- Our clients want to store last names and associated telephone numbers.
- The list may be large.
- They want
 - a class NameNumber for name & number pairs
 - NNCollection()
 - insert(NameNumber)
 - findNumber(String)

NameNumber

```
public class NameNumber {  
    private String lastName;  
    private String telNumber;  
    public NameNumber() {}  
    public NameNumber(String name, String num) {  
        lastName = name;  
        telNumber = num;  
    }  
    public String getLastName() {  
        return lastName;  
    }  
    public String getTelNumber() {  
        return telNumber;  
    }  
}
```

NNCollection

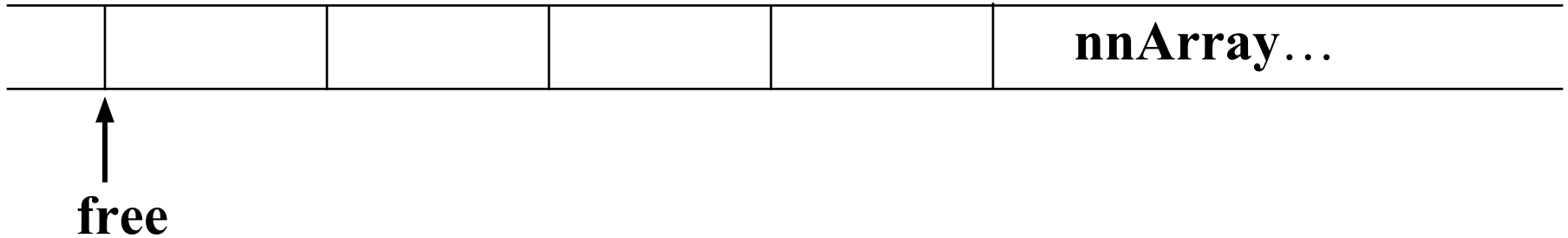
```
public class NNCollection {
    private NameNumber[] nnArray = new NameNumber[100];
    private int free;
    public NNCollection() {free = 0;}
    public void insert(NameNumber n) {
        int index = 0;
        for (int i = free++;
            i!=0 && nnArray[i-1].getLastName().compareTo(n.getLastName())>0;
            i--) {
            nnArray[i] = nnArray[i-1];
            index = i;
        }
        nnArray[index] = n;
    }
}
```

NNCollection (cont.)

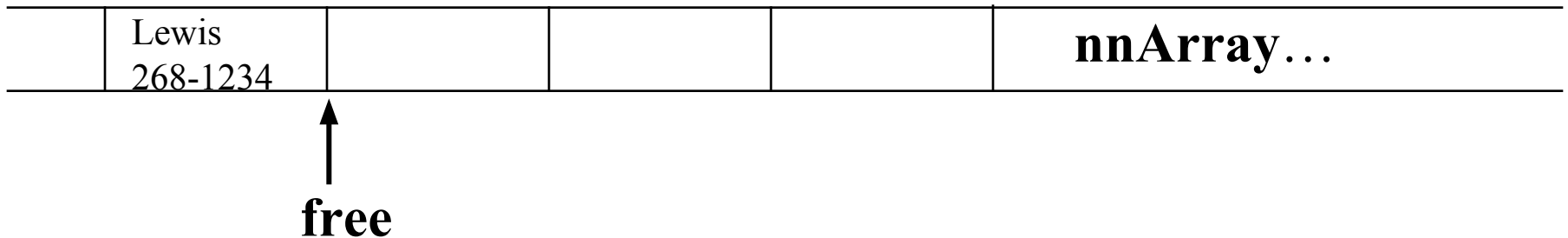
```
public String findNumber(String lName) {  
    for (int i = 0; i != free; i++)  
        if (nnArray[i].getLastName().equals(lName))  
            return nnArray[i].getTelNumber();  
    return new String("Name not found");  
}
```

NNCollection Insertion

Initial Array

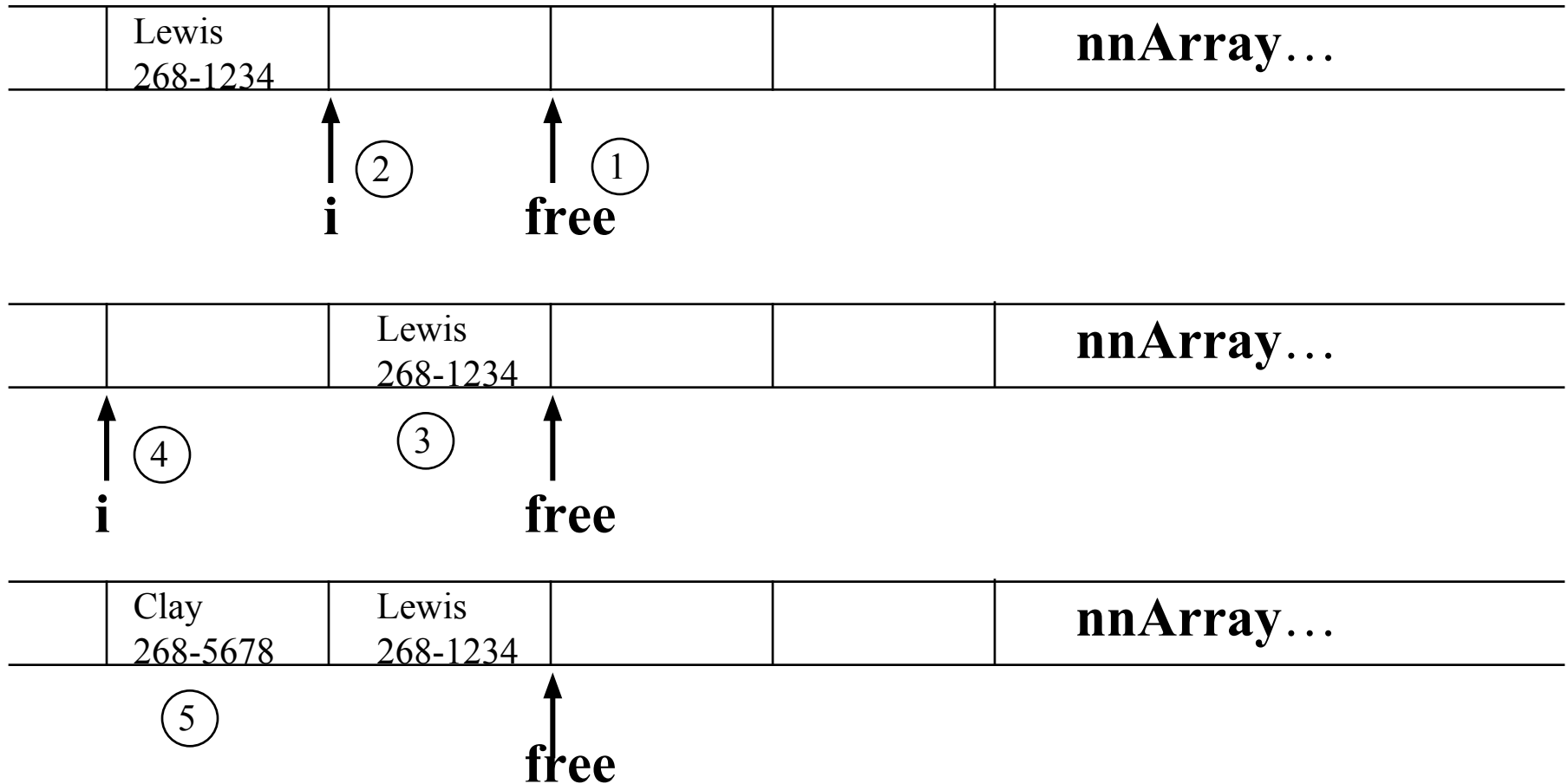


Insert “Lewis”



NNCollection Insertion (cont.)

Insert “Clay”



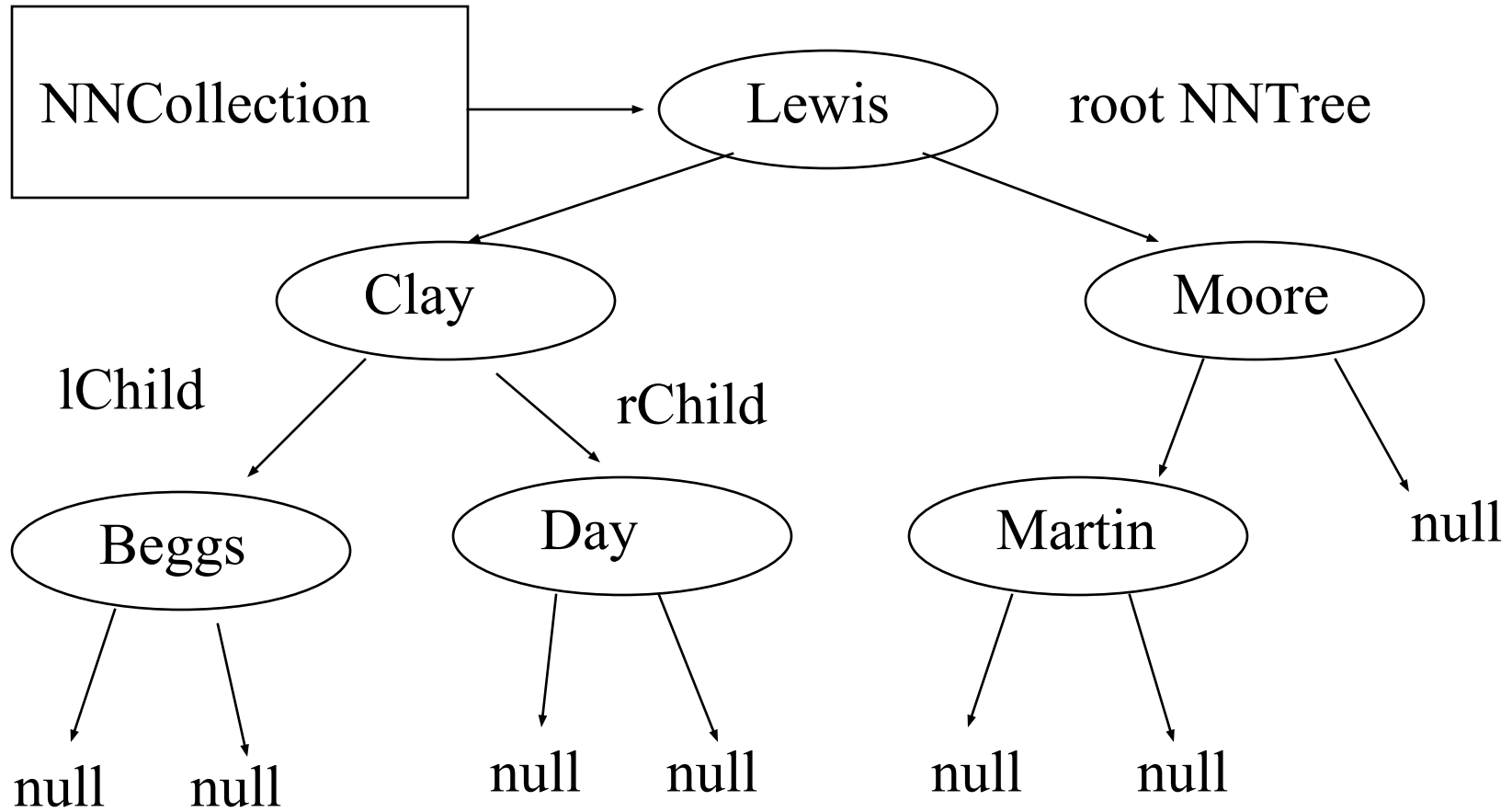
Yes, This Is Rotten

- ❑ It uses a fixed-size array.
- ❑ Array elements are interchanged every time a new name is entered. Slow.
- ❑ The array is searched sequentially. Slow.
- ❑ But, our clients can get started on *their* code.
- ❑ We go back and build a better implementation.

Better NNCollection

- Use a *binary tree*.
- Names “on the left” precede lexicographically.
- Roughly logarithmic insert and retrieve times.
- Very recursive, but not very expensive.

Binary Tree Layout

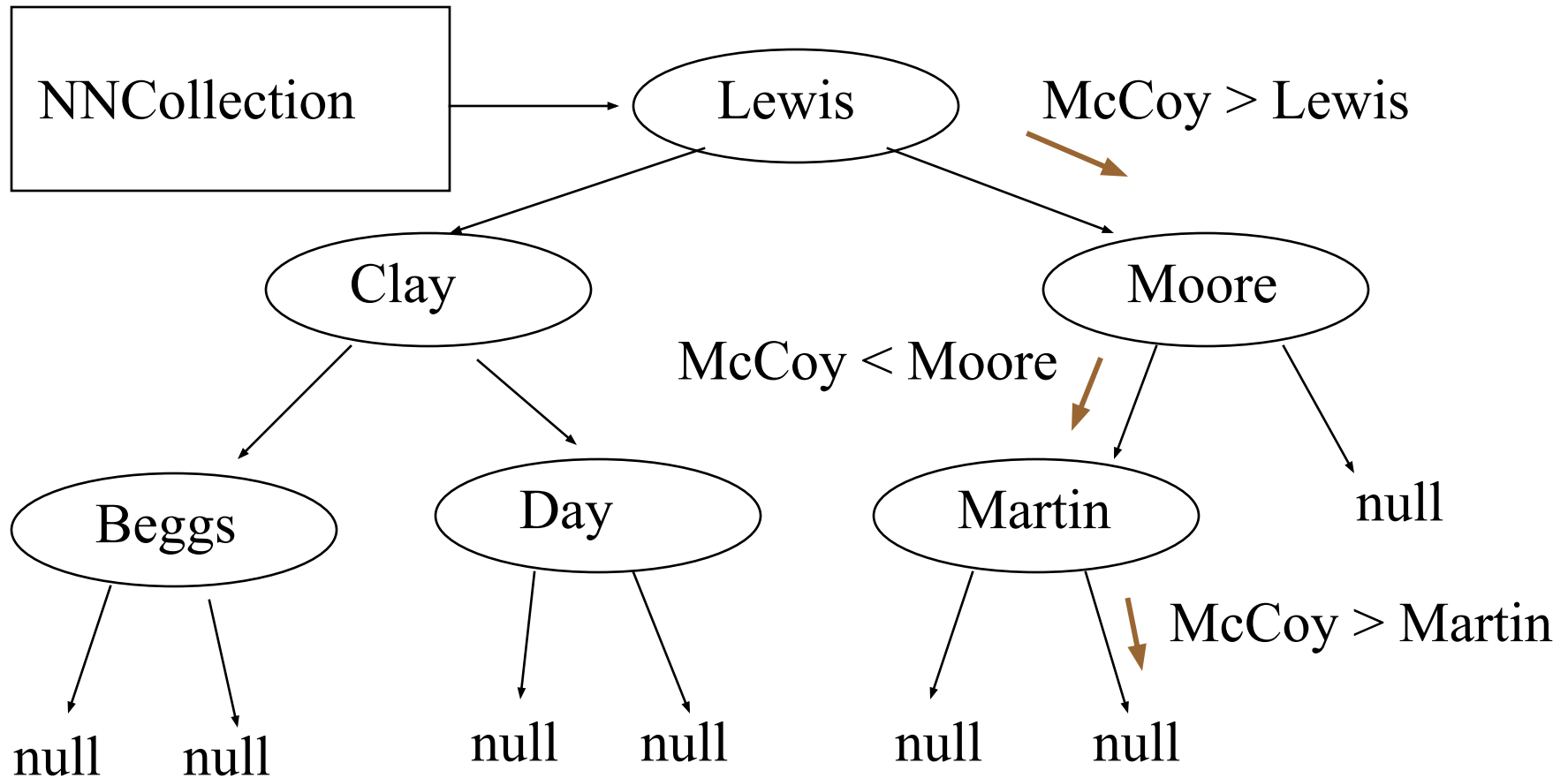


Note: Only the name of the **NameNumber** object is shown

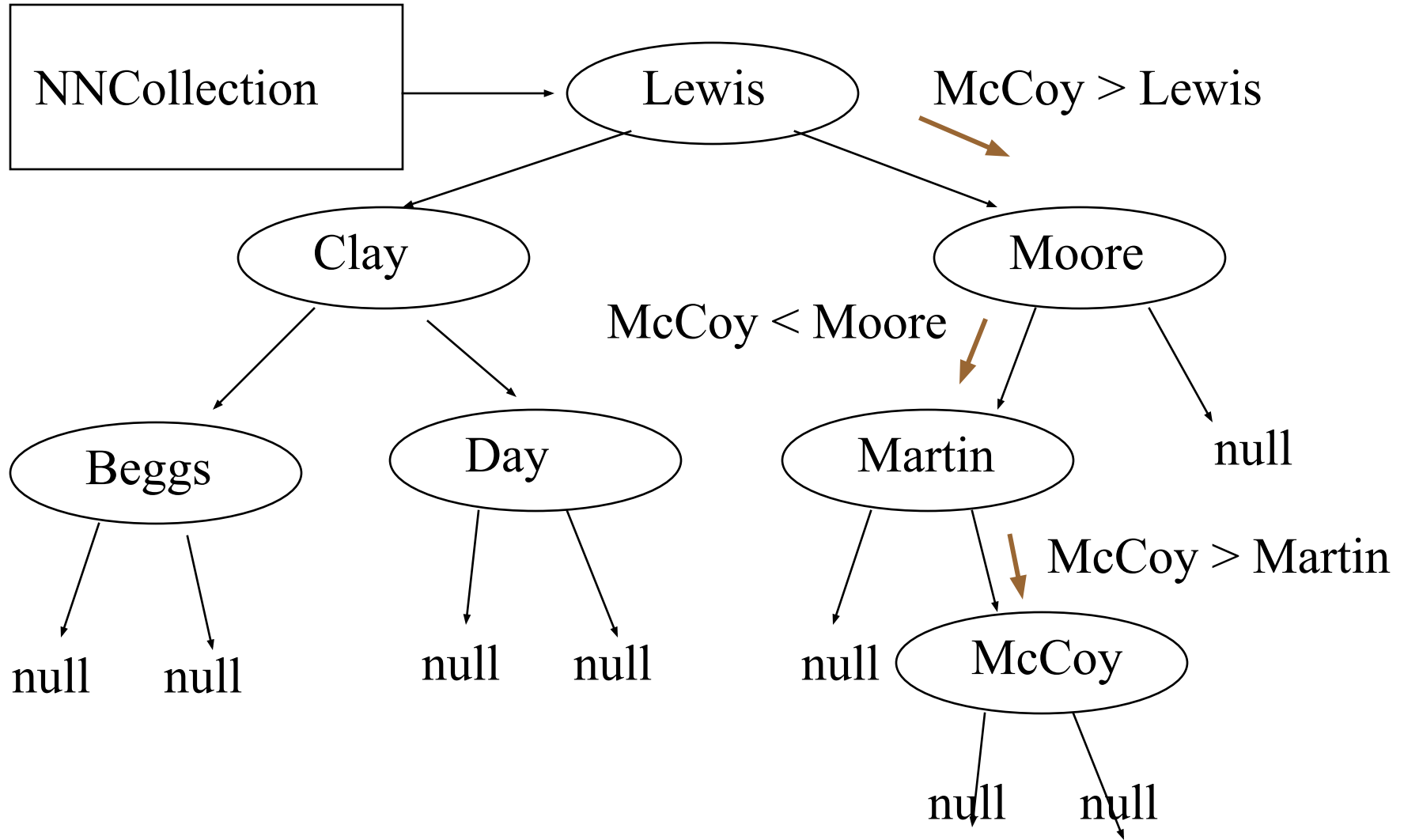
NNTree Class

- Each **NNTree** object
 - is a node, holding a **NameNumber** object.
 - keeps a reference to its left child and right child.
 - knows how to insert a **NameNumber** object.
 - knows how to find a **NameNumber** object.

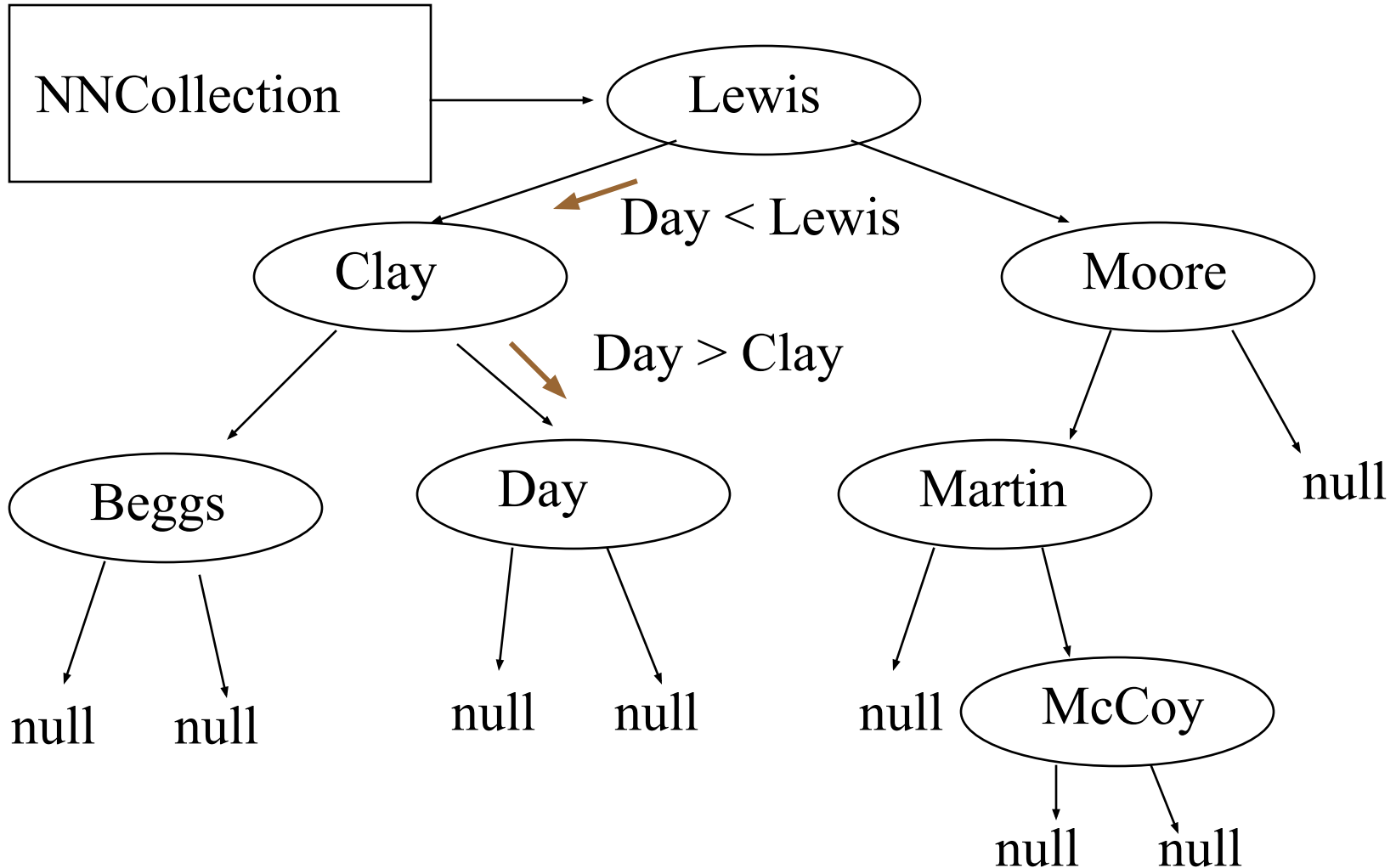
Inserting “McCoy”

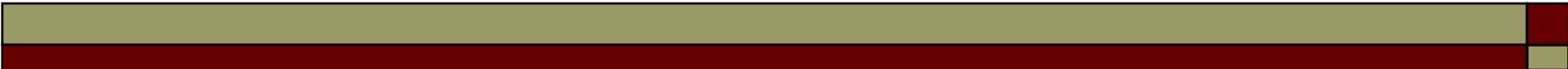


Inserting “McCoy”



Finding “Day”





NNTree Class Definition

```
public class NNTree {  
    private NNTree lChild;  
    private NNTree rChild;  
    private NameNumber contents;  
    public NNTree(NameNumber n) {  
        contents = n;  
    }  
}
```

NNTree Class Definition (cont.)

```
public void insert(NameNumber n) {  
    if (n.getLastName().compareTo(contents.getLastName()) < 0)  
        if (lChild != null)  
            lChild.insert(n);  
        else  
            lChild = new NNTree(n);  
    else  
        if (rChild != null)  
            rChild.insert(n);  
        else  
            rChild = new NNTree(n);  
}
```

NNTree Class Definition (cont.)

```
public String findNumber(String lName) {  
    if (lName.compareTo(contents.getLastName()) < 0)  
        if (lChild != null)  
            return lChild.findNumber(lName);  
        else  
            return new String("Name not found");  
    else if (lName.equals(contents.getLastName()))  
        return contents.getTelNumber();  
    else if (lName.compareTo(contents.getLastName()) > 0)  
        if (rChild != null)  
            return rChild.findNumber(lName);  
        else  
            return new String("Name not found");  
    else  
        return new String("Name not found");  
}
```

NNCollection Again

```
public class NNCollection {  
    private NNTree root;  
    public NNCollection() {}  
    public void insert(NameNumber n) {  
        if (root != null) root.insert(n);  
        else root = new NNTree(n);  
    }  
    String findNumber(String lName) {  
        if (root != null)  
            return root.findNumber(lName);  
        else  
            return new String("Name not found");  
    }  
}
```

More on Packages

- Bringing in a package of classes:
import java.util.*;
- Bringing in a single class:
import java.util.ArrayList;
- The compiler can find these things, through the classpath.
- If we're working from the command line, the classpath must be an environmental variable.

Creating a Package

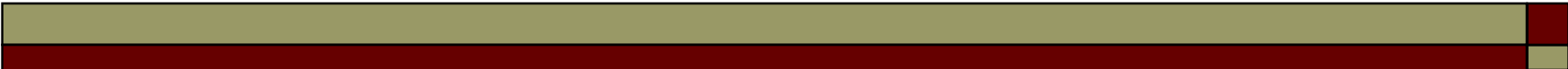
- The very first line in all the files intended for the package named **myPackage**:
package myPackage;
- Put all of the .class files in a directory named **myPackage**.
- Put the **myPackage** directory, as a subdirectory, in a directory given in the classpath.

Class Access

- ❑ Classes can be **public** or not.
- ❑ Non-**public** classes are available only within the package they are defined in.
- ❑ There can be only one **public** class in a “compilation unit” (a .java file).
- ❑ Non-**public** classes are “helper” classes, not for public use.

Class Reuse

- A noble goal, and in Java it's finally happening!
- Basically two ways: *composition* and *inheritance*.
- Composition is called “has-a”.
- Inheritance is called “is-a”.



Composition

- Point, Rectangle, Shape, Triangle
- is-a vs. has-a
- Rectangle has-a point
- Rectangle is-a shape
- Point is-a shape

Inheritance

- An object of a new class that *inherits* from an existing class has all the “powers and abilities” of the parent class:
 - all data members
 - all methods
 - you can add additional data and methods if you wish
 - a derived class object “is-an” object of the parent class type, so can be used in function calls where a parent-class object is specified

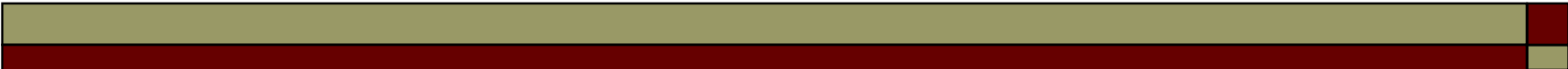
Inheritance Syntax

```
class Cleanser {
    private String activeIngredient;
    public void dilute(int percent) {// water-down}
    public void apply(DirtyThing d) {// pour it on}
    public void scrub(Brush b) {// watch it work}
}

public class Detergent extends Cleanser {
    private String specialIngredient;
    public void scrub(Brush b) {
        // scrub gently, then
        super.scrub(b); // the usual way
    }
    public void foam() { // make bubbles}
}
```

Access Control, Again

- **Detergent** does indeed have an **activeIngredient**, but it's not accessible.
- If **Detergent** needs to access it, it must be either
 - made **protected** (or friendly) in **Cleanser**, or
 - be accessible through get and set methods in **Cleanser**.
- You can't inherit just to get access!



What Is A **Detergent** Object?

- An object of type **Cleanser**, having all the members of **Cleanser**.
- An object of type **Detergent**, having all the additional members of **Detergent**.

Subclasses and Constructors

- Think of a **Detergent** object as containing a **Cleanser** *sub-object*.
- So, that sub-object has to be constructed when you create a **Detergent** object.
- The **Cleanser** object has to be created *first*, since constructing the remaining **Detergent** part might rely on it.
- “Always call the base class constructor first.”

Subclasses and Constructors

```
class Cleanser {
    private String activeIngredient;
    Cleanser() {
        System.out.println("Cleanser constructor");
    }
}

public class Detergent extends Cleanser {
    private String specialIngredient;
    Detergent() {
        System.out.println("Detergent constructor");
    }
    public static void main(String[] args) {
        Detergent d = new Detergent();
    }
}
```

Subclasses and Constructors

```
class Cleanser {
    private String activeIngredient;
    Cleanser(String active) {
        activeIngredient = active;
    }
}

public class Detergent extends Cleanser {
    private String specialIngredient;
    Detergent(String active, String special) {
        super(active); // what if this isn't here?
        specialIngredient = special;
    }
}
```

Composition vs. Inheritance

- Think “has-a” vs. “is-a”.
- Consider the **NNCollection** class. Suppose now we need to store a **String/int** pair (names and ages, perhaps).
- Should we inherit, or compose?
- In either approach, we just need to be able to turn **Strings** into **ints**, and vice versa (not hard).

Composition vs. Inheritance

```
class NACollection {  
    private NNCollection nnc;  
    NACollection() { // ...}  
    public void insert (NameAge n) { uses nnc's insert()  
    public int findAge(String name) { uses nnc's findNumber()  
}
```

or

```
class NACollection extends NNCollection {  
    NACollection() { //...}  
    public void insert(NameAge n) { uses super.insert()  
    public int findAge(String name) { uses super.findNumber()  
}
```

protected Class Members

- ❑ **public** to subclasses.
- ❑ **private** to “the outside world”,
- ❑ except within the package (i.e., they are “friendly”).
- ❑ **private** is usually the best policy, unless you are looking for speed (but then why use Java!?).

Upcasting

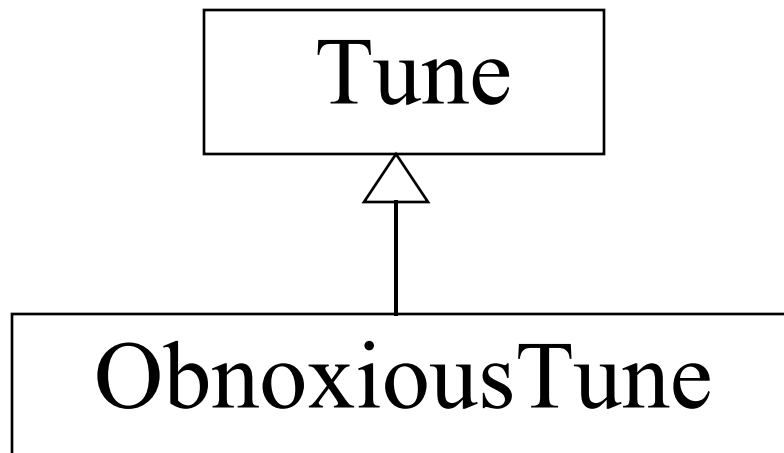
```
class CellPhone {  
    cellPhone() { //...}  
    public void ring(Tune t) { t.play(); }  
}
```

```
class Tune {  
    Tune() { // ...}  
    public void play() { // ...}  
}
```

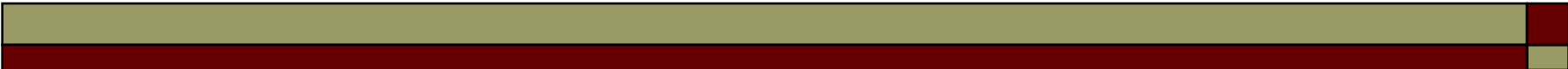
```
class ObnoxiousTune extends Tune{  
    ObnoxiousTune() { // ...}  
    // ...  
}
```

An ObnoxiousTune “is-a” Tune

```
class DisruptLecture {  
    public static void main() {  
        CellPhone noiseMaker = new CellPhone();  
        ObnoxiousTune ot = new ObnoxiousTune();  
        noiseMaker.ring(ot); // ot works though Tune called for  
    }  
}
```



A “UML” diagram



The **final** Keyword

- Vaguely like **const** in C++.
- It says “this is invariant”.
- Can be used for
 - data
 - methods
 - classes
- A kind of protection mechanism.

final Data (Compile-Time)

- For primitive types (**int**, **float**, etc.), the meaning is “this can’t change value”.

```
class Sedan {
```

```
    final int numDoors = 4;
```

- For references, the meaning is “this reference must always refer to the same object”.

```
    final Engine e = new Engine(300);
```

final Data (Run-Time)

- Called a “blank final;” the value is filled in during execution.

```
class Sedan {  
    final int topSpeed;  
    Sedan(int ts) {  
        topSpeed = ts;  
        // ...  
    }  
}  
  
class DragRace {  
    Sedan chevy = new Sedan(120), ford = new Sedan(140);  
    //! chevy.topSpeed = 150;
```

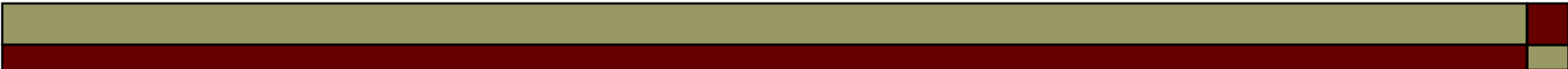
final Method Arguments

```
class Sedan {  
    public void replaceTire(final Tire t) {  
        //! t = new Tire();  
    }  
}
```

- Same idea:
 - a **final** primitive has a constant value
 - a **final** reference always refers to the same object.
- Note well: a **final** reference does *not* say that the object *referred to* can't change (*cf.* C++)

final Methods

- ❑ **final** methods cannot be overridden in subclasses. Maybe a bad idea?
- ❑ **final** methods can be inlined, allowing the compiler to insert the method code where it is called.
- ❑ This may improve execution speed.
- ❑ Only useful for small methods.
- ❑ **private** methods are implicitly **final**.



final Classes

- These can't be inherited from (ummm, “subclassed”?).
- All methods are implicitly **final**, so inlining can be done.

Class Loading

- A **.class** file is loaded when
 - the first object of that type is created, or
 - when a static member is first used.
- When a derived class object is created, the base class file is immediately loaded (before the derived class constructor actually goes to work).

Variable-Length Argument Lists

```
class A { int i; }
public class VarArgs {
    static void f(Object[] x) {
        for (int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
        f(new Object[] {
            new Integer(47), new VarArgs(),
            new Float(3.14), new Double(11.11) } );
        f(new Object[] {"one", "two", "three" } );
        f(new Object[] {new A(), new A(), new A() } );
    }
}
```

Variable-Length Argument Lists

- This prints

47

VarArgs@fee6172e

3.14

11.11

one

two

three

A@fee61874

A@fee61873

A@fee6186a