接口

刘 钦

南京大学软件学院

Outline

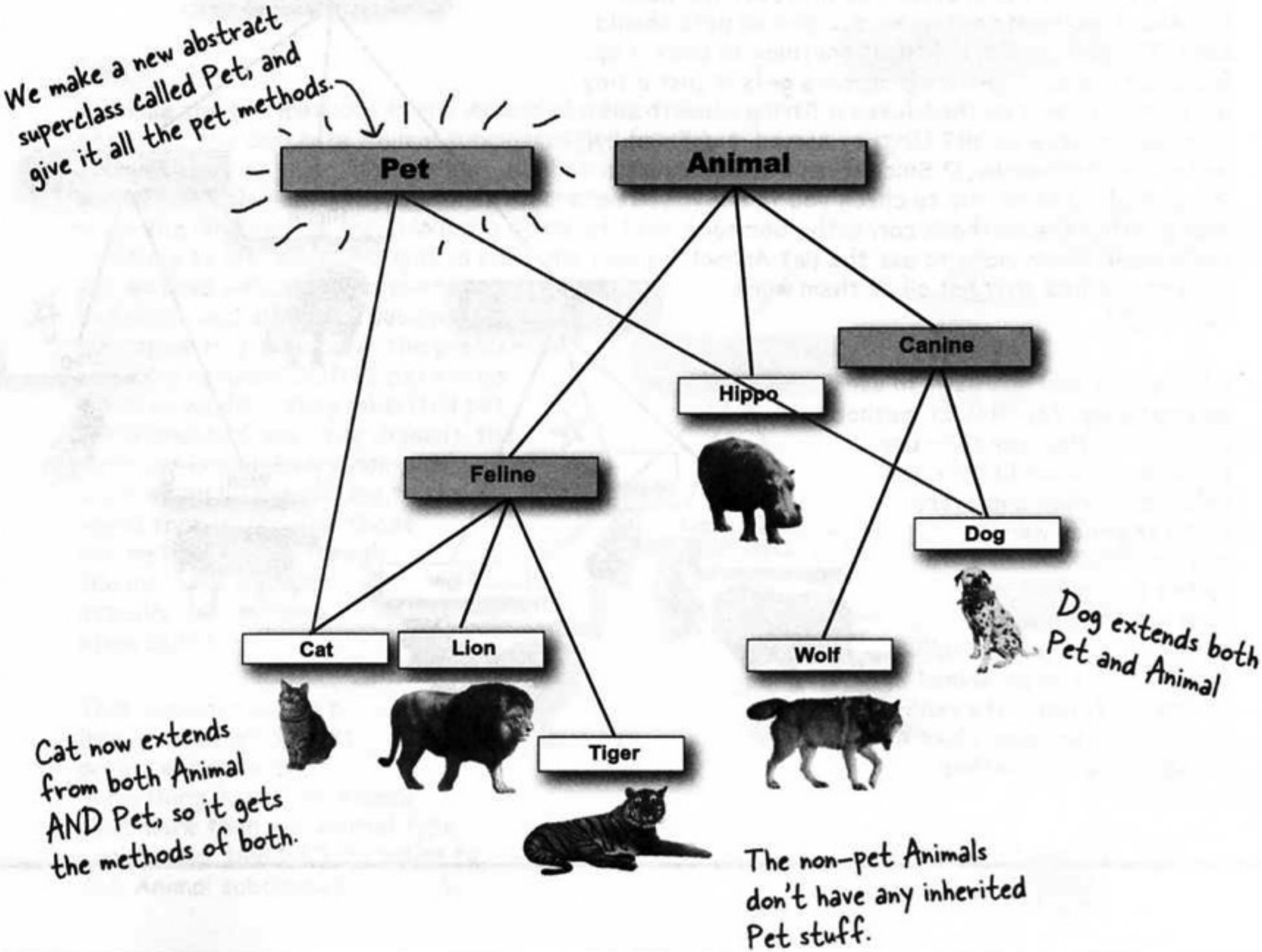
- 接口
- Classes versus Interfaces

- What if later you want to use Dog for a PetShop program?
  - A Pet needs methods like beFriendly() and Play().
- PetShop program may have many things.

So what we REALLY need is:

- A way to have pet behavior in **just** the pet classes
- A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.), without having to cross your fingers and hope all the programmers get it right.
- A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class.

It looks like we need TWO superclasses at the top



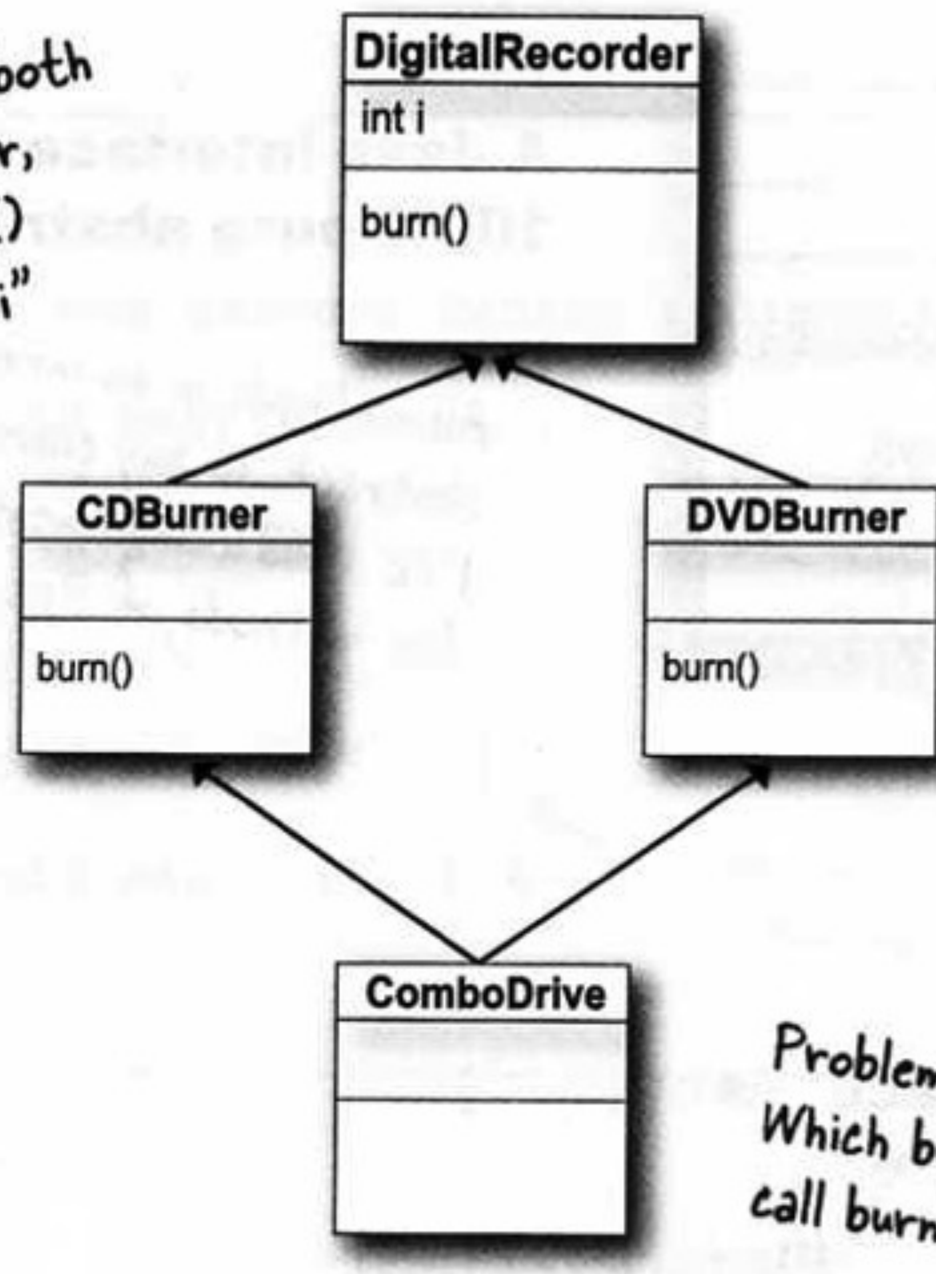
It's called "multiple inheritance" and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

Deadly Diamond of Death

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



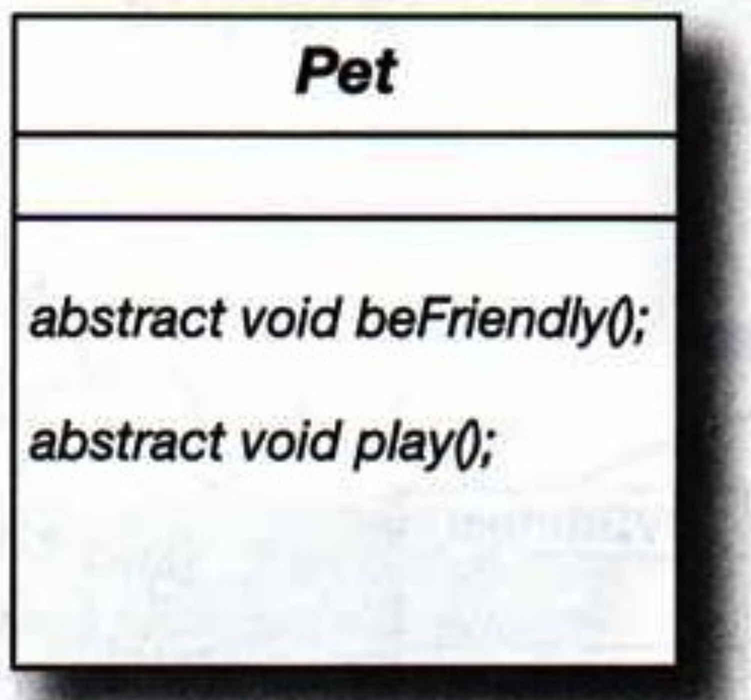
Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

Interface

A Java interface solves your multiple inheritance problem by giving you much of the polymorphic *benefits* of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

The way in which interfaces side-step the DDD is surprisingly simple: ***make all the methods abstract!*** That way, the subclass ***must*** implement the methods (remember, abstract methods *must* be implemented by the first concrete subclass), so at runtime the JVM isn't confused about *which* of the two inherited versions it's supposed to call.



A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e. override) the methods of Pet.

To DEFINE an interface:

```
public interface Pet {...}
```

Use the keyword "interface" instead of "class"

To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet {...}
```

Use the keyword "implements" followed by the interface name. Note that when you implement an interface you still get to extend a class

Making and Implementing the Pet interface

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

You say 'interface' instead of 'class' here

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}
```

All interface methods are abstract, so they MUST end in semicolons. Remember, they have no body!

Dog IS-A Animal and Dog IS-A Pet

You say 'implements' followed by the name of the interface.

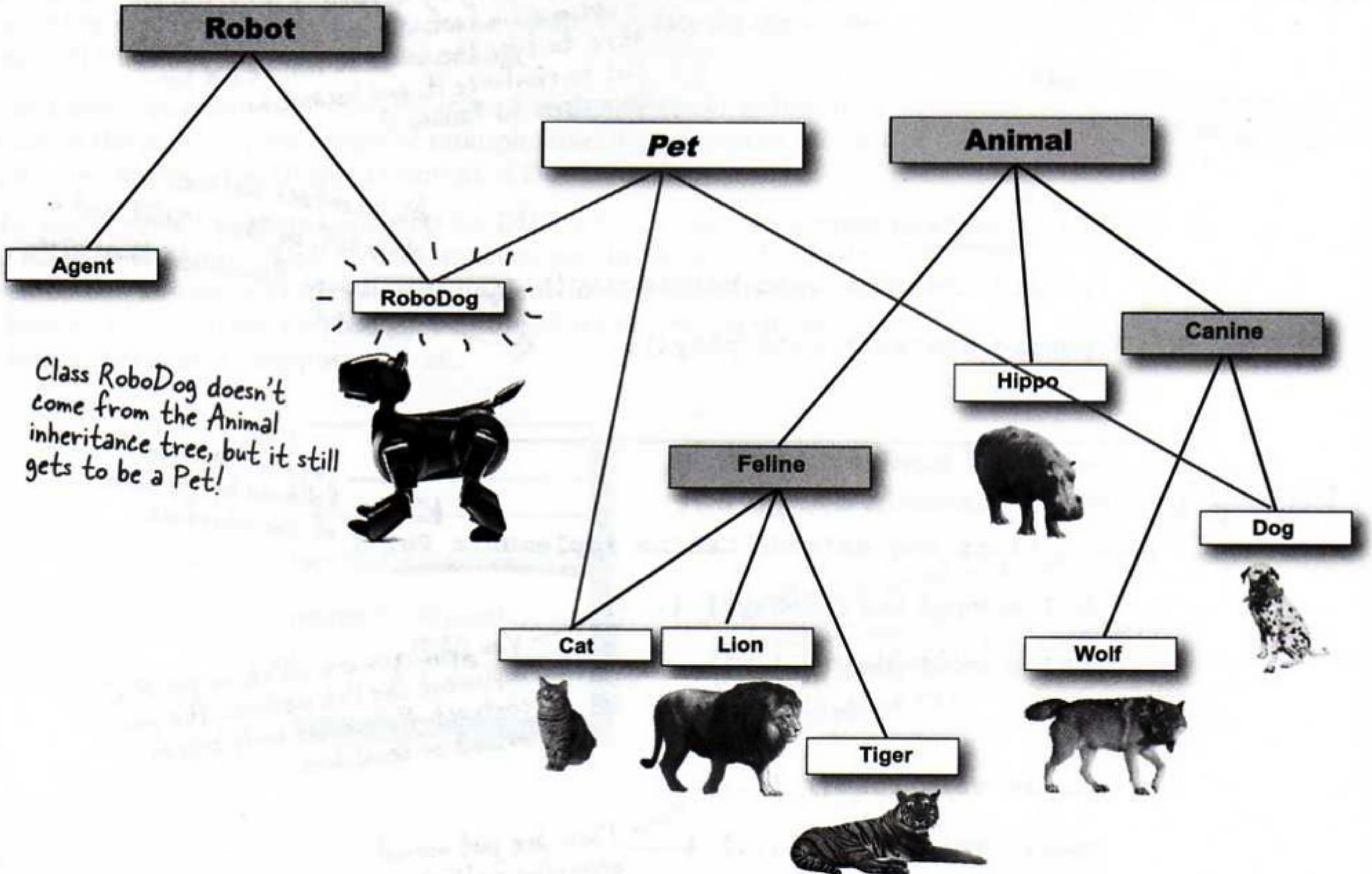
```
public class Dog extends Canine implements Pet {
    public void beFriendly() {...}
    public void play() {..}

    public void roam() {...}
    public void eat() {...}
}
```

You SAID you are a Pet, so you MUST implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

These are just normal overriding methods.

Classes from *different* inheritance trees can implement the *same* interface.

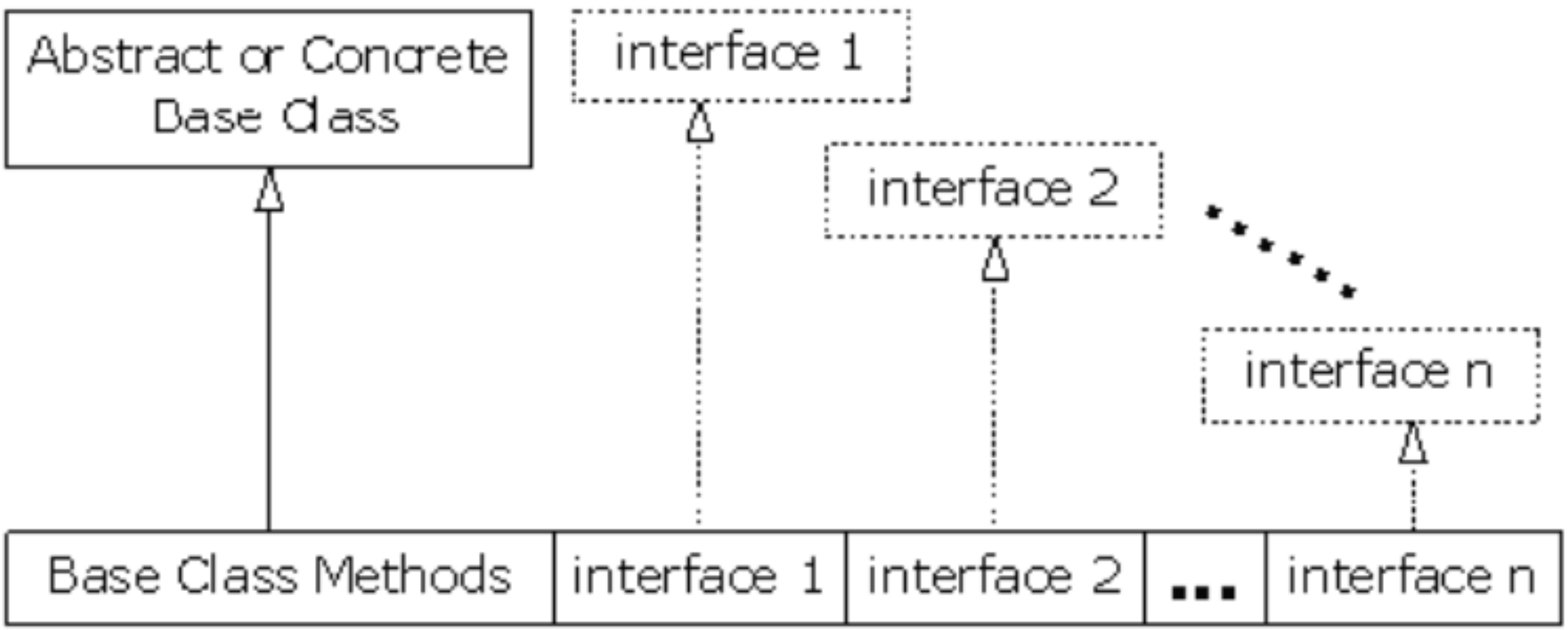


Extends one & Implements more

Better still, a class can implement *multiple* interfaces!

A Dog object IS-A Canine, and IS-A Animal, and IS-A Object, all through inheritance. But a Dog IS-A Pet through interface implementation, and the Dog might implement other interfaces as well. You could say:

```
public class Dog extends Animal implements
Pet, Saveable, Paintable { ... }
```


Abstract or Concrete Base Class
interface 1
interface 2
interface n
Base Class Methods
interface 1
interface 2
...
interface n

How do you know whether to make a class, a subclass, an *abstract* class, or an interface?

- Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.
- Make a subclass (in other words, *extend* a class) only when you need to make a ***more specific*** version of a class and need to override or add new behaviors.
- Use an abstract class when you want to define a ***template*** for a group of subclasses, and you have at least *some* implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- Use an interface when you want to define a ***role*** that other classes can play, regardless of where those classes are in the inheritance tree.

通过继承扩展接口

- //: c08:HorrorShow.java
- // Extending an interface with inheritance.
- interface Monster {
- void menace();
- }
- interface DangerousMonster extends Monster {
- void destroy();
- }

- //: c08:RandVals.java
- // Initializing interface fields with
- // non-constant initializers.
- import java.util.*;
- public interface RandVals {
    - Random rand = new Random();
    - int randomInt = rand.nextInt(10);
    - long randomLong = rand.nextLong() * 10;
    - float randomFloat = rand.nextLong() * 10;
    - double randomDouble = rand.nextDouble() * 10;
- } ///:~
- 在接口中定义的数据成员自动是**static** 和**final** 的。它们不能是“空final”，但是可以被非常量表达式初始化。
- 这些数据成员不是接口的一部分，只是被存储在该接口的静态存储区域内。
- 接口可以嵌套在类或其它接口中

Invoking the superclass version of a method

superclass version of the method does important stuff that subclasses could use

```
abstract class Report {
    void runReport() {
        // set-up report
    }
    void printReport() {
        // generic printing
    }
}

class BuzzwordsReport extends Report {

    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();

    }
    void buzzwordCompliance() {...}
}
```

call superclass version, then come back and do some subclass-specific stuff

Classes versus Interfaces

- Explicit use of concrete class names
  - locks you into specific implementations
  - making down-the-line changes unnecessarily difficult
  - losing flexibility
- Programming to interfaces
  - parallel design and development

Design I

-

```
f()
{   LinkedList list = new LinkedList();
    //...
    g( list );
}
g( LinkedList list )
{
    list.add( ... );
    g2( list )
}
```

Design II

- a new requirement for fast lookup has emerged

```
f()
{   Collection list = new LinkedList();
    //...
    g( list );
}
g( Collection list )
{
    list.add( ... );
    g2( list )
}
```

针对接口编程

As another example, compare this code:

```
f()
{   Collection c = new HashSet();
    //...
    g( c );
}
g( Collection c )
{
    for( Iterator i = c.iterator(); i.hasNext() ;)
        do_something_with( i.next() );
}
```

to this:

```
f2()
{   Collection c = new HashSet();
    //...
    g2( c.iterator() );
}
g2( Iterator i )
{   while( i.hasNext() ;)
        do_something_with( i.next() );
}
```

The g2() method can now traverse Collection derivatives as well as the key and value lists you can get from a Map. In fact, you can write iterators that generate data instead of traversing a collection. You can write iterators that feed information from a test scaffold or a file to the program. There's enormous flexibility here.

Problems of Inheritance

- **Losing flexibility**
- **Coupling**
- **The fragile base-class problem**

Design I

```
class Stack extends ArrayList
{   private int stack_pointer = 0;
    public void push( Object article )
    {   add( stack_pointer++, article );
    }
    public Object pop()
    {   return remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {   for( int i = 0; i < articles.length; ++i )
            push( articles[i] );
    }
}
```

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

Disadvantage

- First, if you override everything, the base class should really be an interface, not a class. There's no point in implementation inheritance if you don't use any of the inherited methods.

- Second, and more importantly, you don't want a stack to support all ArrayList methods.

Design II

```
class Stack
{   private int stack_pointer = 0;
    private ArrayList the_data = new ArrayList();
    public void push( Object article )
    {   the_data.add( stack_pointer++, article );
    }
    public Object pop()
    {   return the_data.remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {   for( int i = 0; i < o.length; ++i )
            push( articles[i] );
    }
}
```

Design III

```
class Monitorable_stack extends Stack
{
    private int high_water_mark = 0;
    private int current_size;
    public void push( Object article )
    {   if( ++current_size > high_water_mark )
            high_water_mark = current_size;
        super.push(article);
    }

    public Object pop()
    {   --current_size;
        return super.pop();
    }
    public int maximum_size_so_far()
    {   return high_water_mark;
    }
}
```

So far so good, but consider the fragile base-class issue. Let's say you want to create a variant on Stack that tracks the maximum stack size over a certain time period. One possible implementation might look like this

- This new class works well, at least for a while. Unfortunately, the code exploits the fact that push_many() does its work by calling push(). At first, this detail doesn't seem like a bad choice. It simplifies the code, and you get the derived class version of push(), even when the Monitorable_stack is accessed through a Stack reference, so the high_water_mark updates correctly.

- One fine day, someone might run a profiler and notice the Stack isn't as fast as it could be and is heavily used. You can rewrite the Stack so it doesn't use an ArrayList and consequently improve the Stack's performance.

Design IV

```
class Stack
{   private int stack_pointer = -1;
    private Object[] stack = new Object[1000];
    public void push( Object article )
    {   assert stack_pointer < stack.length;
        stack[ ++stack_pointer ] = article;
    }
    public Object pop()
    {   assert stack_pointer >= 0;
        return stack[ stack_pointer-- ];
    }
    public void push_many( Object[] articles )
    {   assert (stack_pointer + articles.length) < stack.length;
        System.arraycopy(articles, 0, stack, stack_pointer+1,
                                                    articles.length);
        stack_pointer += articles.length;
    }
}
```

- The new version of Stack works fine; in fact, it's better than the previous version. Unfortunately, the Monitorable_stack derived class doesn't work any more, since it won't correctly track stack usage if push_many() is called (the derived-class version of push() is no longer called by the inherited push_many() method, so push_many() no longer updates the high_water_mark). Stack is a fragile base class.

Summary

- In general, it's best to avoid concrete base classes and extends relationships in favor of interfaces and implements relationships.

Listing 0.1. Eliminate fragile base classes using interfaces

```
import java.util.*;

interface Stack
{
    void push( Object o );
    Object pop();
    void push_many( Object[] source );
}

class Simple_stack implements Stack
{   private int stack_pointer = -1;
    private Object[] stack = new Object[1000];

    public void push( Object o )
    {   assert stack_pointer < stack.length;

        stack[ ++stack_pointer ] = o;
    }

    public Object pop()
    {   assert stack_pointer >= 0;

        return stack[ stack_pointer-- ];
    }

    public void push_many( Object[] source )
    {   assert (stack_pointer + source.length) < stack.length;

        System.arraycopy(source,0,stack,stack_pointer+1,source.length);
        stack_pointer += source.length;
    }
}


class Monitorable_Stack implements Stack
{
    private int high_water_mark = 0;
    private int current_size;
    Simple_stack stack = new Simple_stack();

    public void push( Object o )
    {   if( ++current_size > high_water_mark )
            high_water_mark = current_size;
        stack.push(o);
    }

    public Object pop()
    {   --current_size;
        return stack.pop();
    }

    public void push_many( Object[] source )
    {
        if( current_size + source.length > high_water_mark )
            high_water_mark = current_size + source.length;

        stack.push_many( source );
    }

    public int maximum_size()
    {   return high_water_mark;
    }
}
```