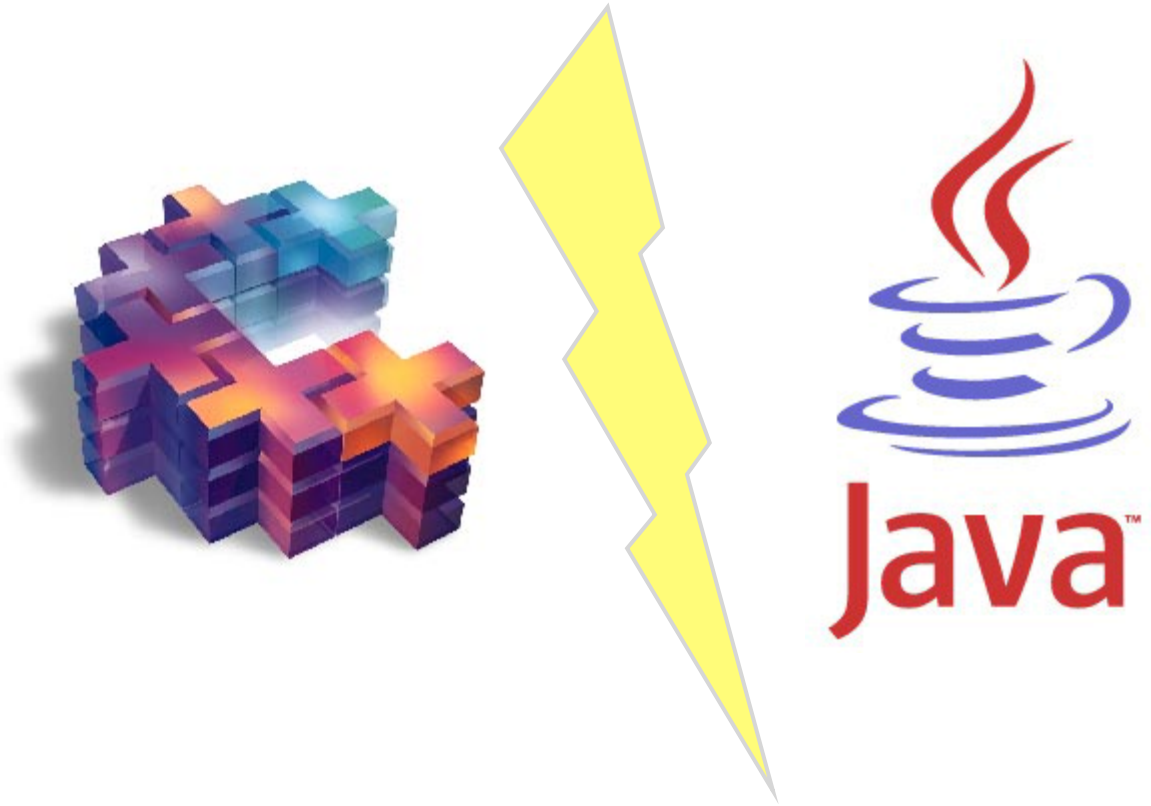
Aris Papadopoulos
ap7@doc.ic.ac.uk

Programming

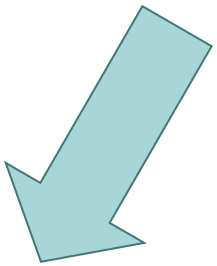
Why Java?

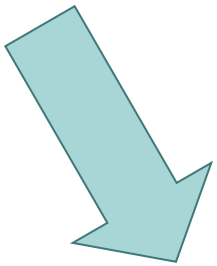


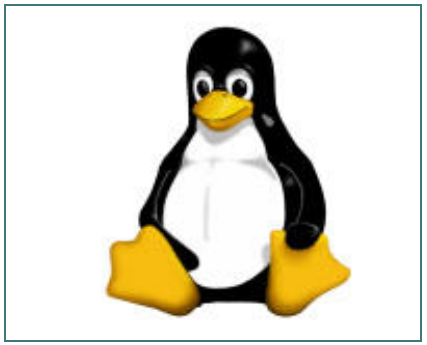
1. Portability

Compiling C++ Programs

```
#include <stdio>
main( int argc, char *argv[]) {
  // do something
}
```







Java Architecture

- Wrapping up the power of:
  - Java language
  - Java Virtual Machine
  - Java file format
  - Java API

Bytecode

Hello.java

```
class Hello {
  public static void main(String[] args) {
    System.out.println( “Hello World” );
}
```

javac

Hello.class

A highly optimised set of portable instructions, executable on a JVM.

```
…
Method Hello()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello World!">
   5 invokevirtual #4 <Method void println(String)>
   8 return
```

The Java Virtual Machine

```
class Hello {
  public static void main() {
  // do something
}
```

Hello.class

Mac JVM

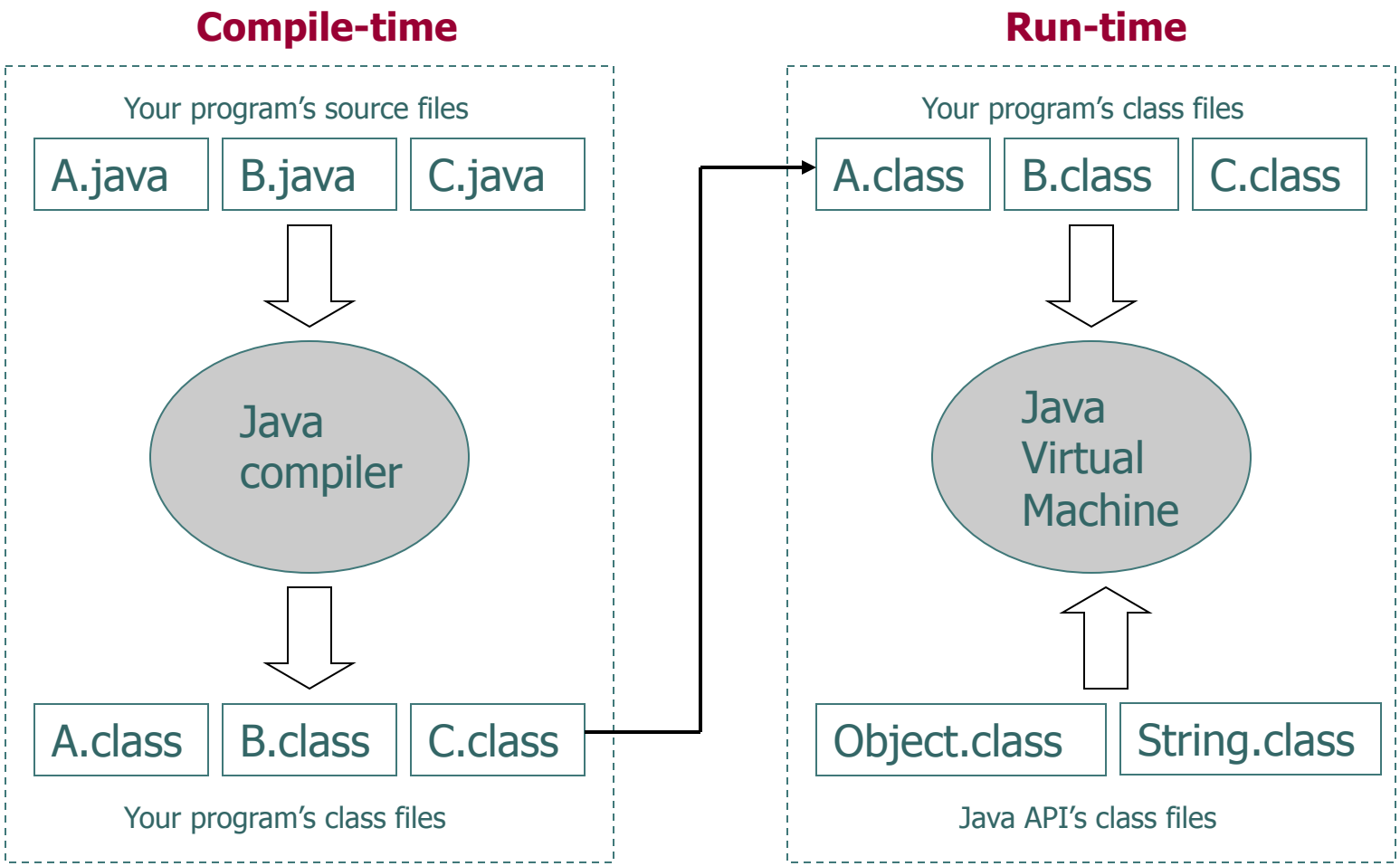
Hello.class

Linux JVM

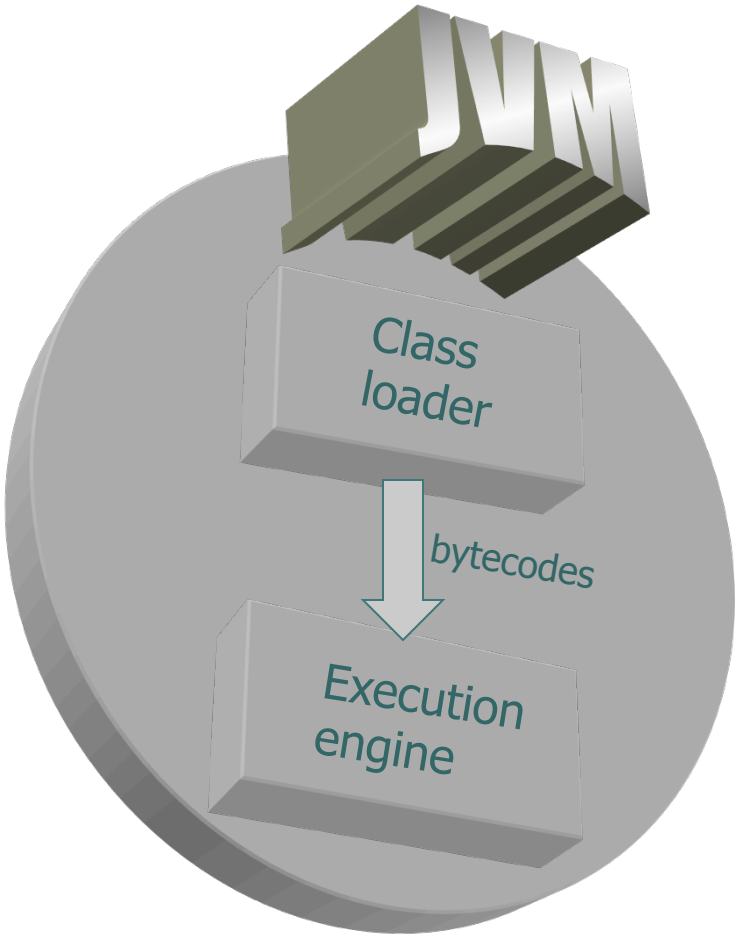
Hello.class

Win JVM

The Java Virtual Machine



The Java Virtual Machine



- One at a time
- Just-in-time compiler
- Adaptive optimizer

Want more reasons?

2. Security:
   - Because execution of the bytecode is controlled by the JVM, harmful actions can be prohibited, which leads us to the…
3. Applets:
   - Dynamically downloaded programs that safely execute in the context of a browser.

4. Memory Management.

- C++ has two ways of creating objects

```
int main( int argc , char *[] argv ) {

  Dog d("rover");
  Cat c("fluffy");

  d.bark();

  Dog *dp = new Dog("sirius");

  dp->bark();

  delete dp;
}
```

On the stack ("automatic")

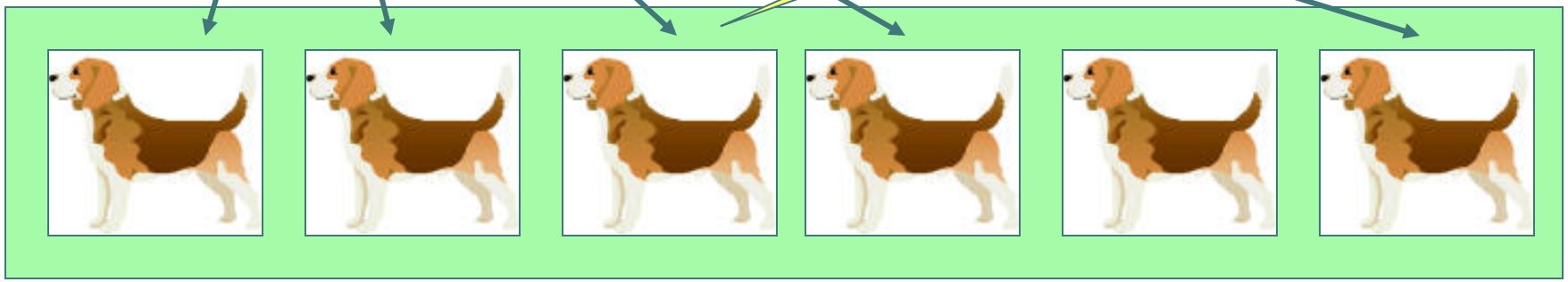
On the heap

Need to delete

Memory Management (C++)

```
int main( int argc , char *[] argv ) {

  for( int i=0; i<100 ; i++ ) {
    Dog *dp = new Dog("sirius");
    dp->bark();
  }
}
```

You need to explicitly delete



Memory Management (Java)

- All Java objects are created on the heap

```
public static void main( String[] args ) {
  Dog d = new Dog("rover");
  Cat c = new Cat("fluffy");
  d.bark();
  Dog dp = new Dog("sirius");
  dp.bark();
}
```

All objects are accessed through references

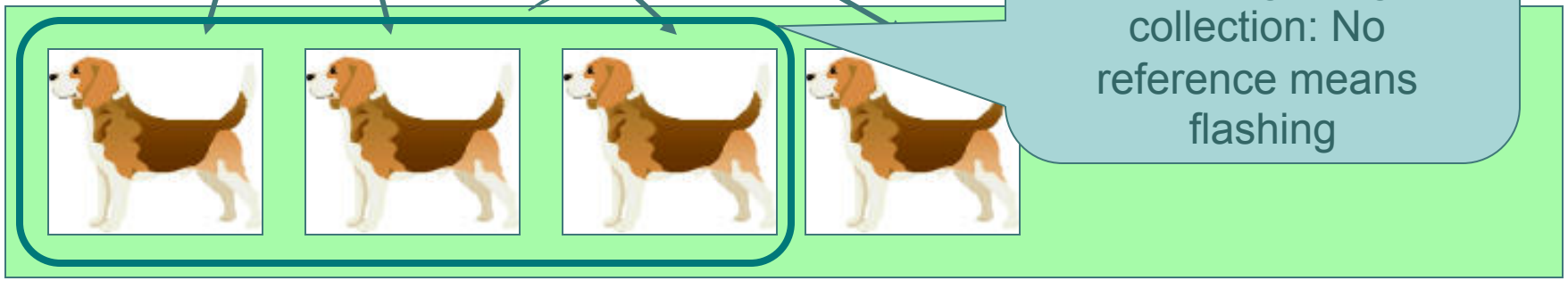
No stars or arrows

Memory Management (Java)

```
public static void main( String[] args ){
   for( int i=0; i<100 ; i++ ) {
     Dog dp = new Dog("sirius");
     dp.bark();
   }
}
```

Java's heap is *garbage collected*

The reference approach to objects enables garbage collection: No reference means flashing



Memory Management (Java)

```
public static void main( String[] args ){

  for( int i=0; i<100 ; i++ ) {
    Dog dp = new Dog("sirius");
    dp.bark();
  }
}
```

Garbage Collection: Reference Counting

- Object creation: Reference count = 1;
- New reference: count++;
- Reference out of scope OR new value: count--;
- If reference count == 0: object can be freed.

A First Program

All programs must have at least one class

At least one class

A standard library function to print text on the console

```
class Hello {

  public static void ma                tring[] args ) {

    System.out.println("Our first Java program");
  }
}
```

O.O. Java

- Programs comprise sets of classes
- Classes can have
  - Fields
  - Methods
  - Constructors

A Dog Class

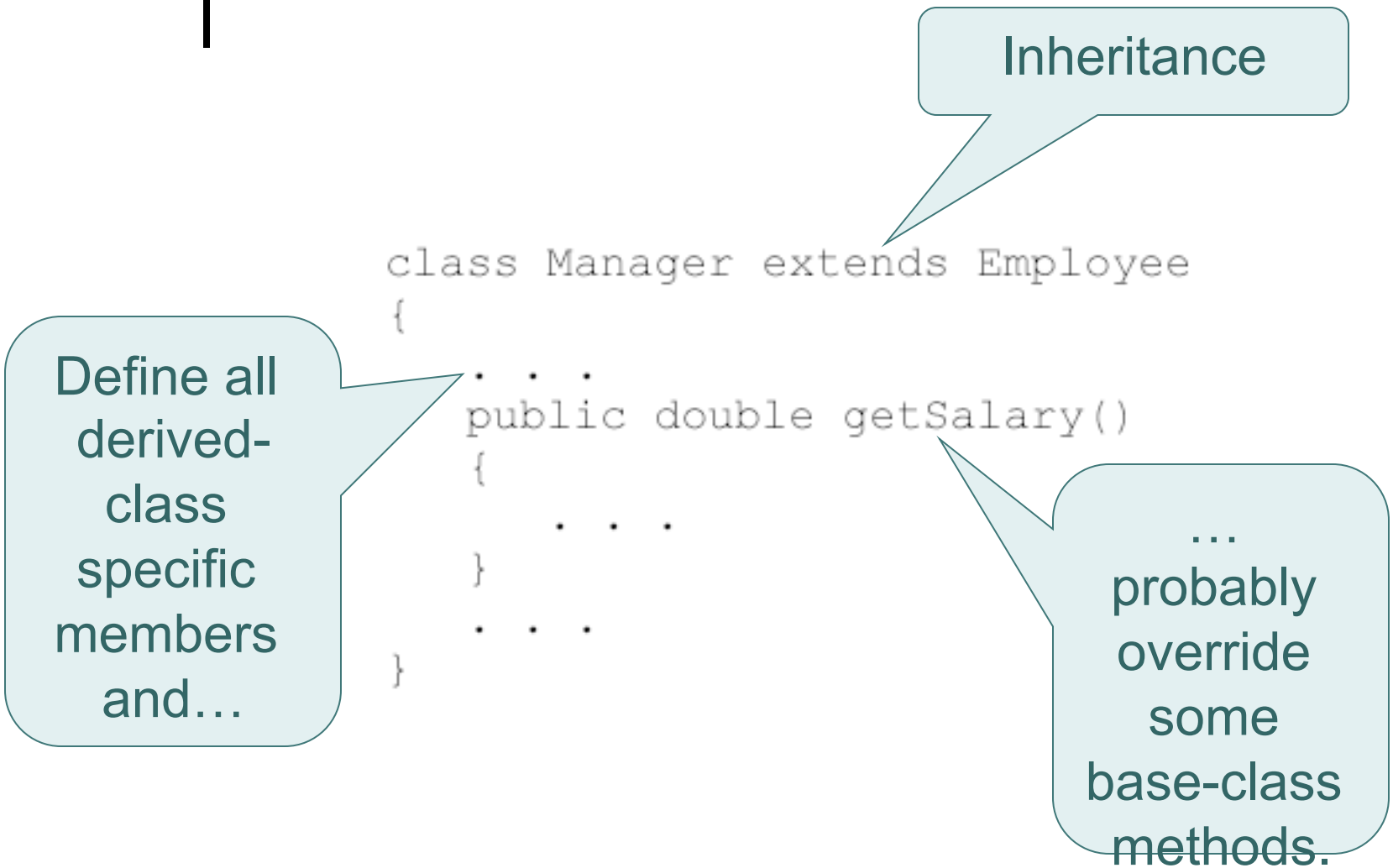
```
class Dog {

  String name;
  int age;

  Dog ( String p_name ) {
    name = p_name;
  }

  void bark() {
    System.out.println("Woof! Woof!");
  }
}
```

Field declarati

A constructor. No destructor as Java has garbage

Methods are defined inside the class

Inheritance



Inheritance

- if Manager's getSalary() needs access to an Employee's private member, e.g. salary, neither of these would work:

```
public double getSalary()
{
    return salary + bonus; // won't work
}
```

```
public double getSalary()
{
   double baseSalary = getSalary(); // still won't work
   return baseSalary + bonus;
}
```

Inheritance

- Use of super:

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

Inheritance

- Use of super in constructor syntax:

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

Call the constructor of Manager's superclass with the given parameters

- Corresponding C++ (initialiser list) syntax:

```
Manager::Manager(String n, double s, int year, int month,
   int day) // C++
: Employee(n, s, year, month, day)
{
   bonus = 0;
}
```

Inheritance

- If you do not want a class to be inherited, you use final:

```
final class Executive extends Manager
{
    . . .
}
```

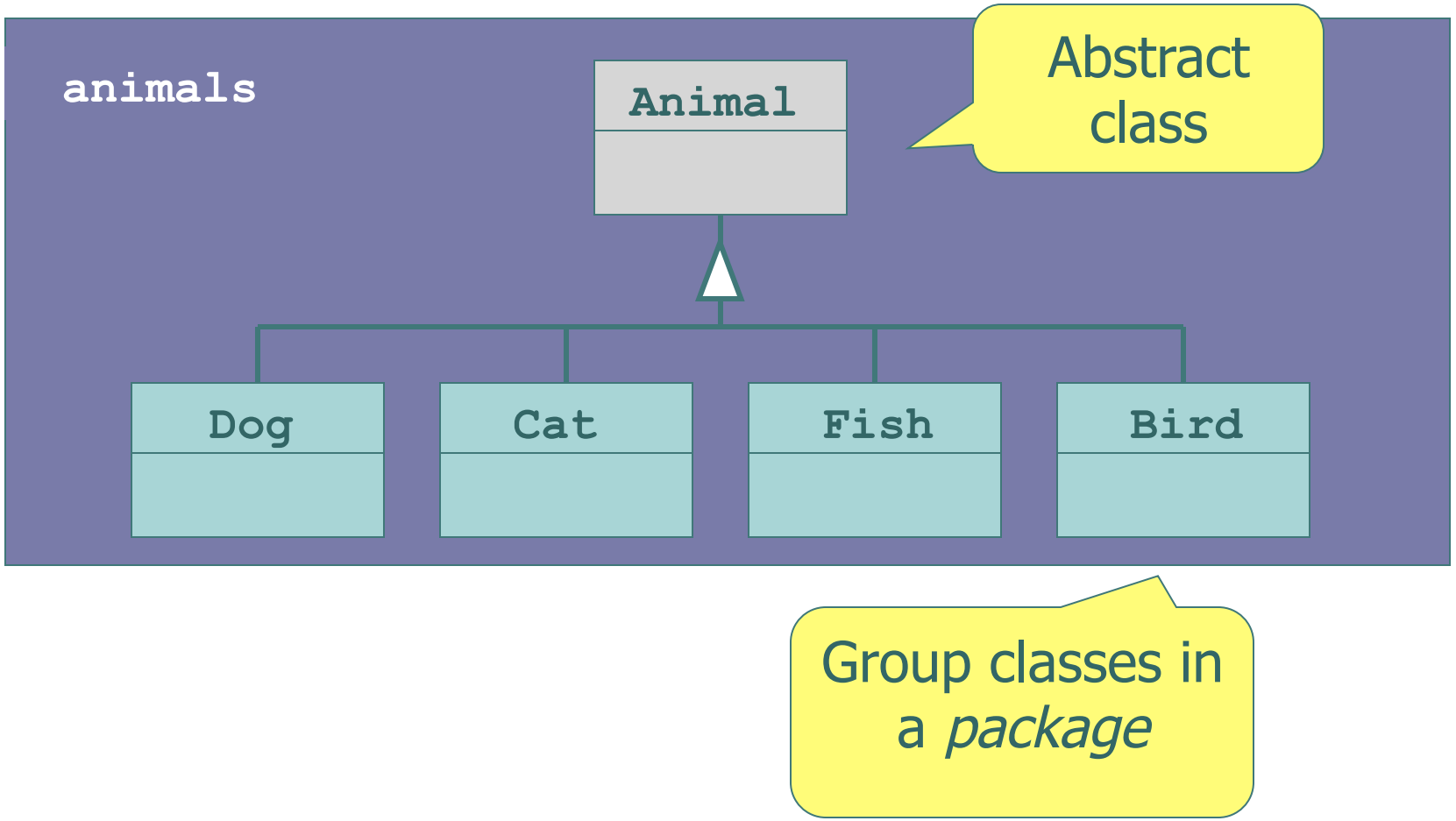
Packages

- Java allows grouping classes in collections.
- Main purpose is guarantying uniqueness of class names.
- Sun recommends the reverse domain name rule, e.g.:
  - uk.ac.ic.doc.project_name

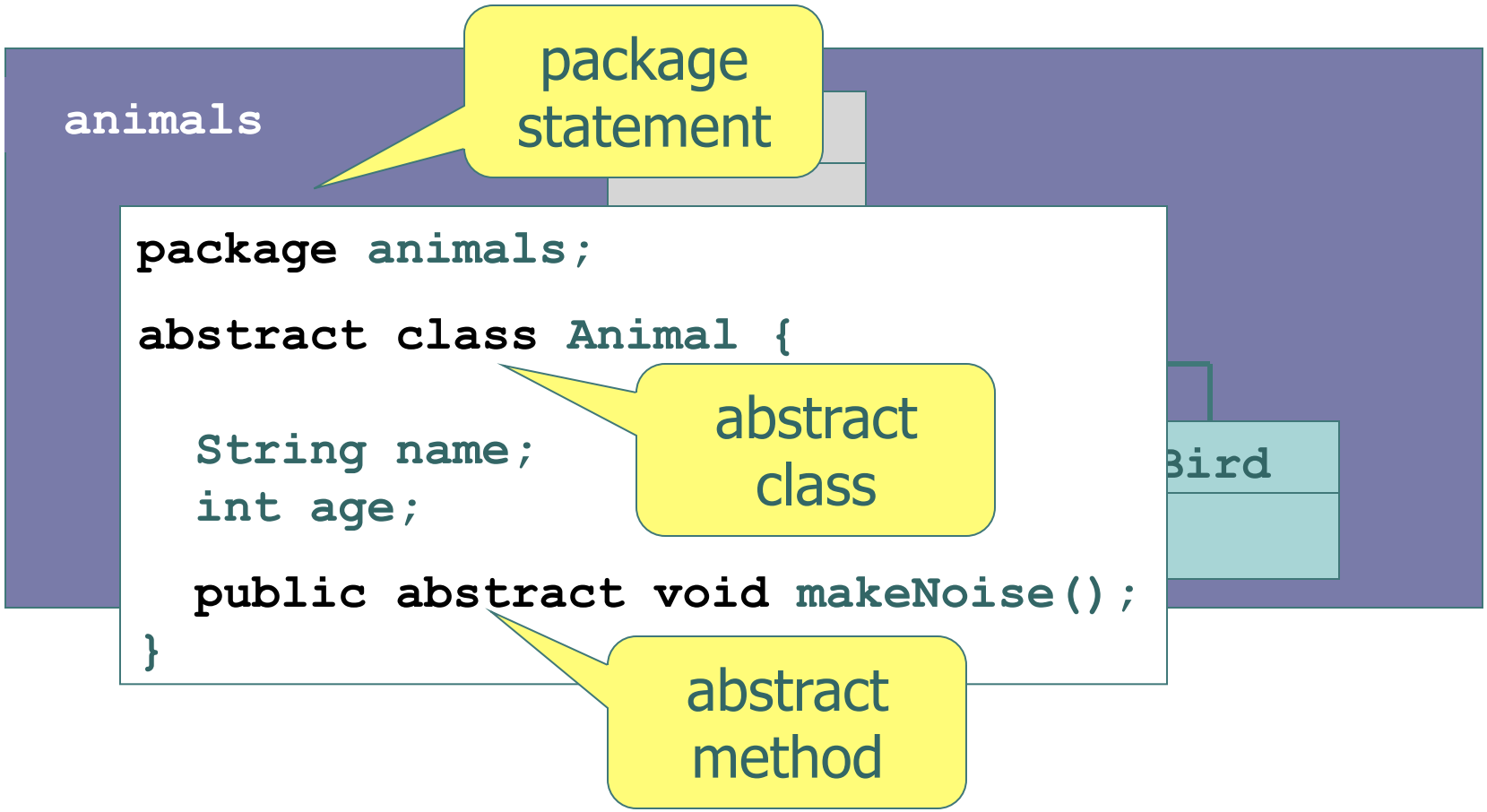
Packages

- The standard Java library is distributed over a number of packages, e.g.: java.lang, java.util, java.net, and so on.
- The sole purpose of package nesting is to manage unique names.
- So there is no relationship between nested packages, e.g.: java.util and java.util.jar have nothing to do with each other.

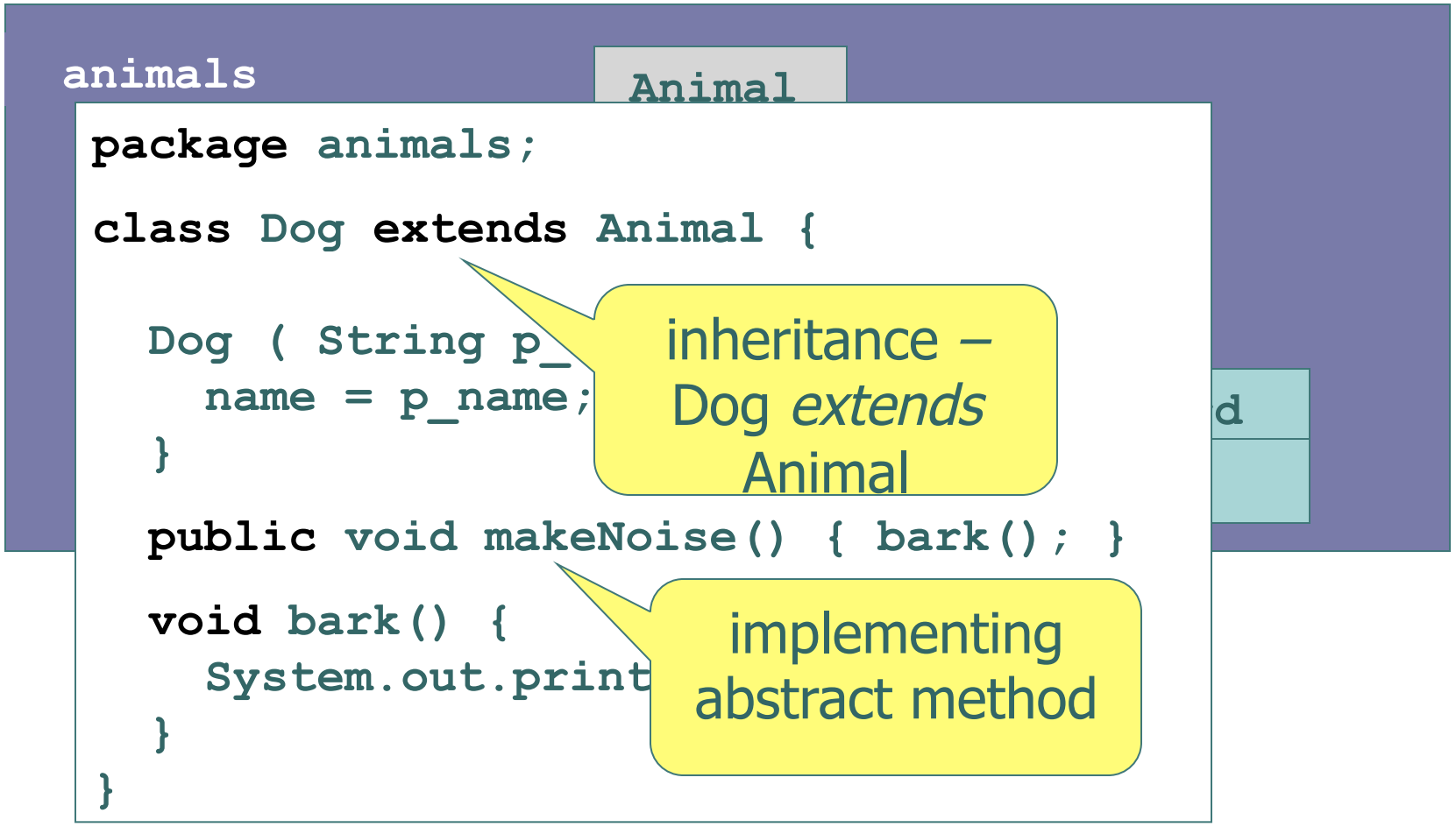
Some other Animals



Some other Animals



Some other Animals



Using our Animals

Import all classes defined in animals package

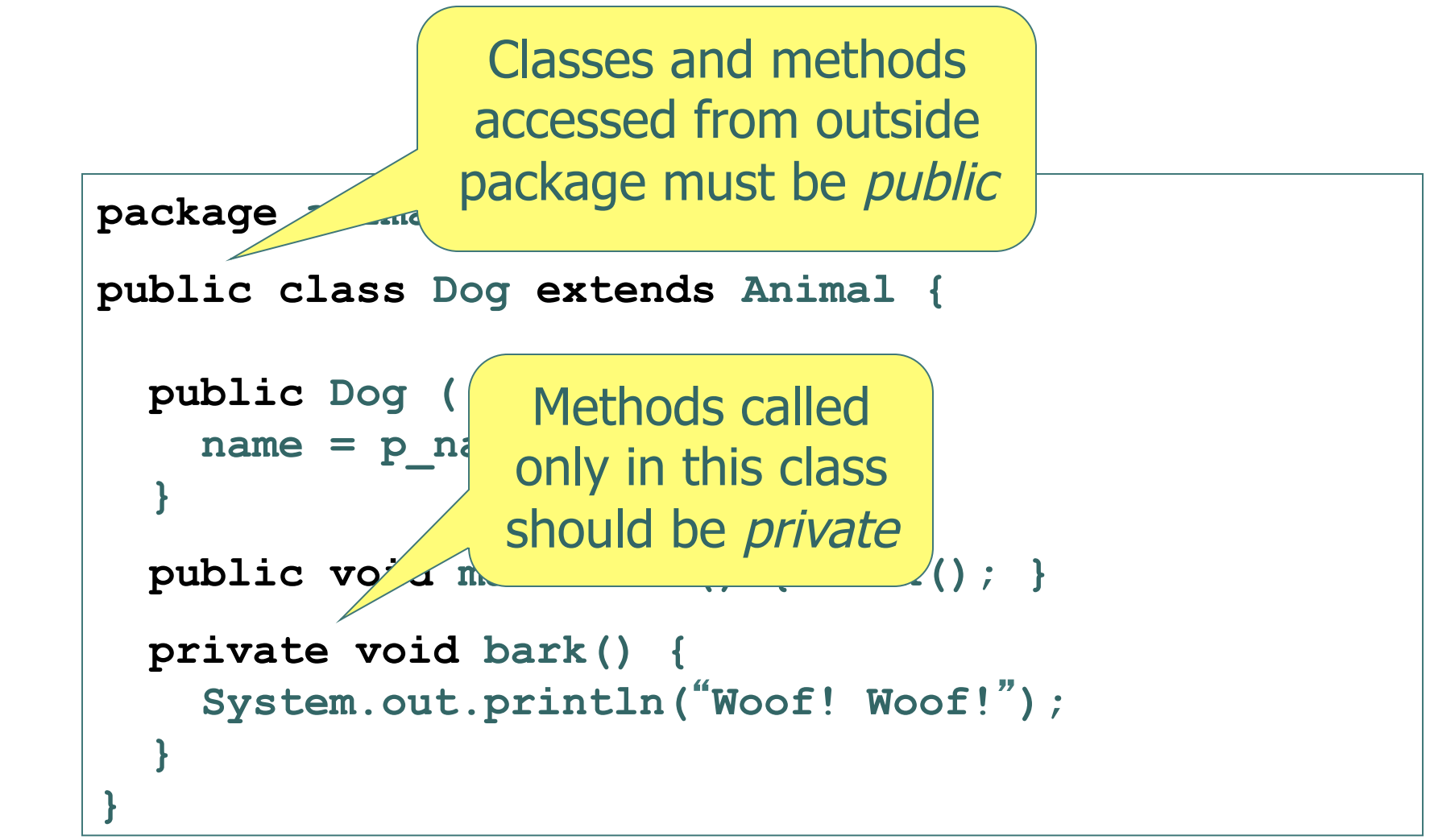
```
import animals.*;

class PetShop {

  public static void main( String[] args ) {

    Animal pet = new Dog("rover");
    pet.makeNoise();

    pet = new Bird();
    pet.makeNoise();
  }
}
```

Creating an object with the *new* keyword

Using our Animals

```
import animals.*;

class PetShop {

  public static void main( String[] args ) {

    Animal pet = new animals.Dog("rover");
    pet.makeNoise();

    pet = new Bird();
    pet.makeNoise();
  }
}
```

Access Modifiers



Class member access

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package - subclass	No	Yes	Yes	Yes
Same package – non subclass	No	Yes	Yes	Yes
Different package - subclass	No	No	Yes	Yes
Different package – non subclass	No	No	No	Yes

5. Streamlined Polymorphism

- As in C++, you can assign a derived-class object to a base-class reference.
- However, in Java, you do not need to declare a method as virtual. Dynamic binding is the default behavior.

5. Streamlined Polymorphism

```
Manager boss = new Manager("Carl Cracker", 80000,
   1987, 12, 15);
boss.setBonus(5000);
```

We make an array of three employees:

```
Employee[] staff = new Employee[3];
```

We populate the array with a mix of managers and employees:

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000,
   1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000,
   1990, 3, 15);
```

5. Streamlined Polymorphism

We print out everyone's salary:

```
for (int i = 0; i < staff.length; i++)
{
   Employee e = staff[i];
   System.out.println(e.getName() + " "
      + e.getSalary());;
}
```

This loop prints the following data:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Now `staff[1]` and `staff[2]` each print their base salary because they are `Employee` objects. However, `staff[0]` is a `Manager` object and its `getSalary` method adds the bonus to the base salary.

What is remarkable is that the call

```
e.getSalary()
```

picks out the *correct* `getSalary` method.

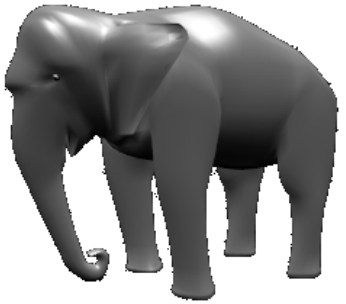
5. Streamlined Polymorphism

- If you do *not* want a method to be overridden and thus to be virtual, you tag it as final. (Why would you use final?)

```
class Employee
{
   . . .
   public final String getName()
   {
      return name;
   }
   . . .
}
```

Java has Single Inheritance

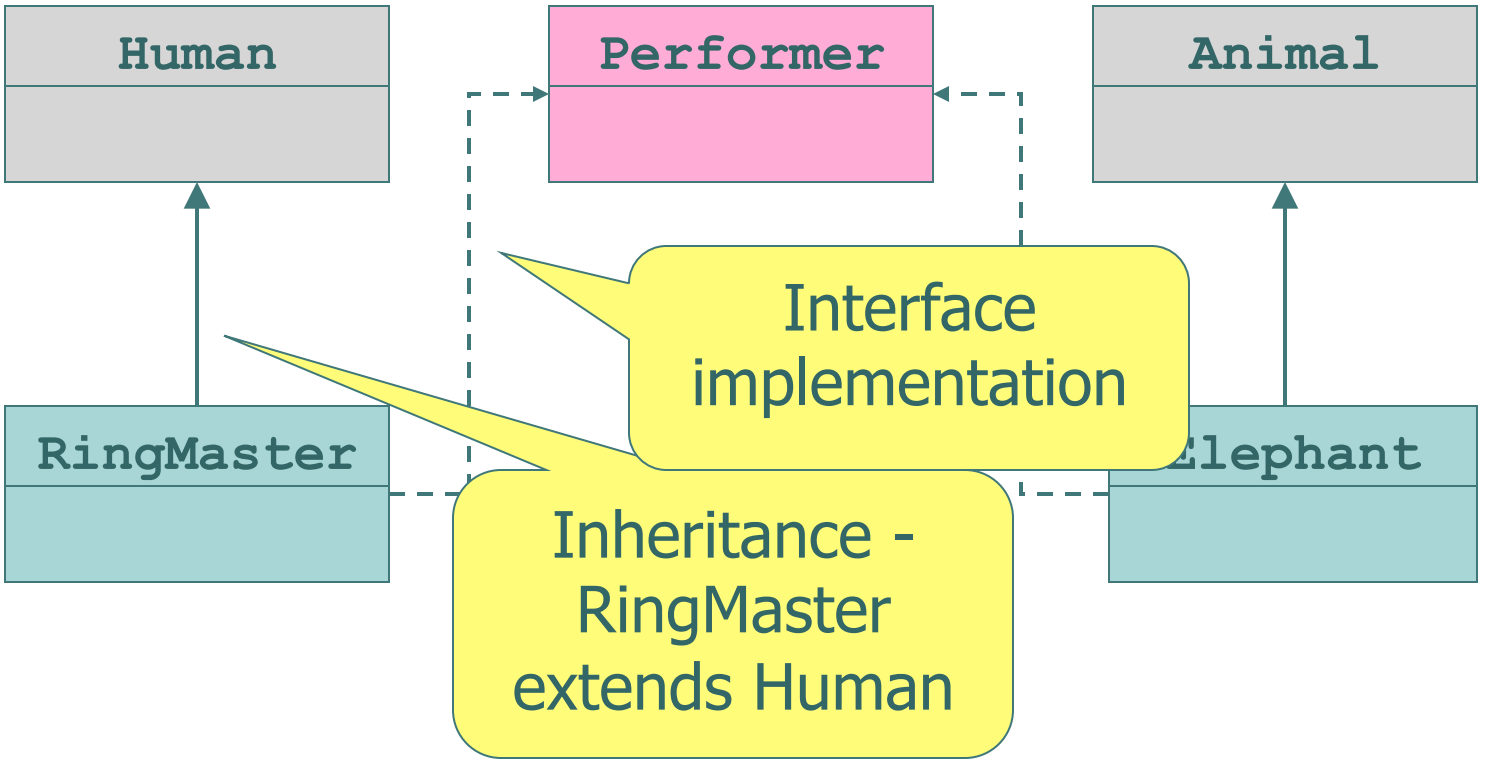
An Elephant is an Animal



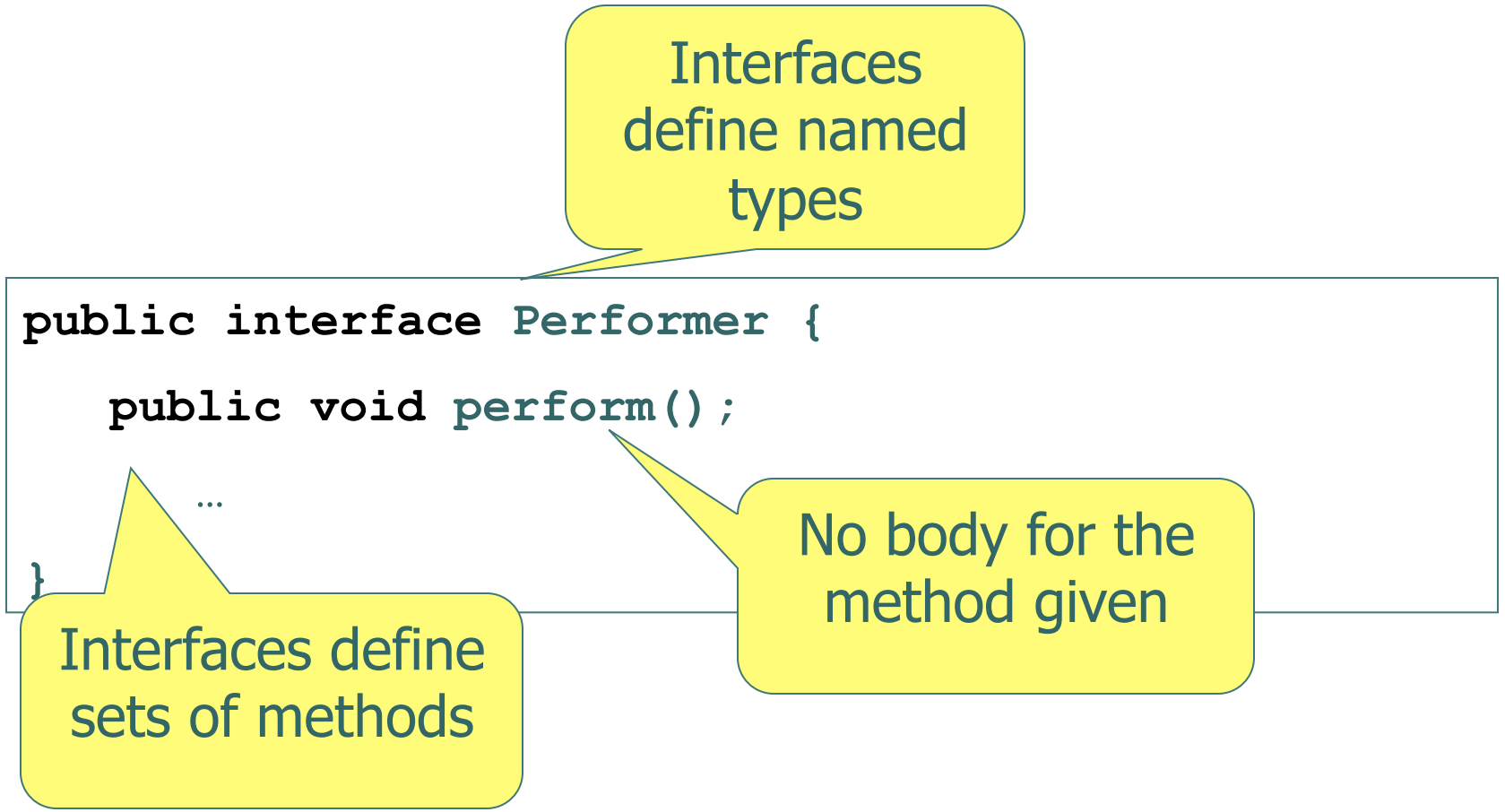
A RingMaster is a Human

What if we want them both to be Performers?

5. Streamlined Polymorphism, continued: Interfaces



Performer Interface



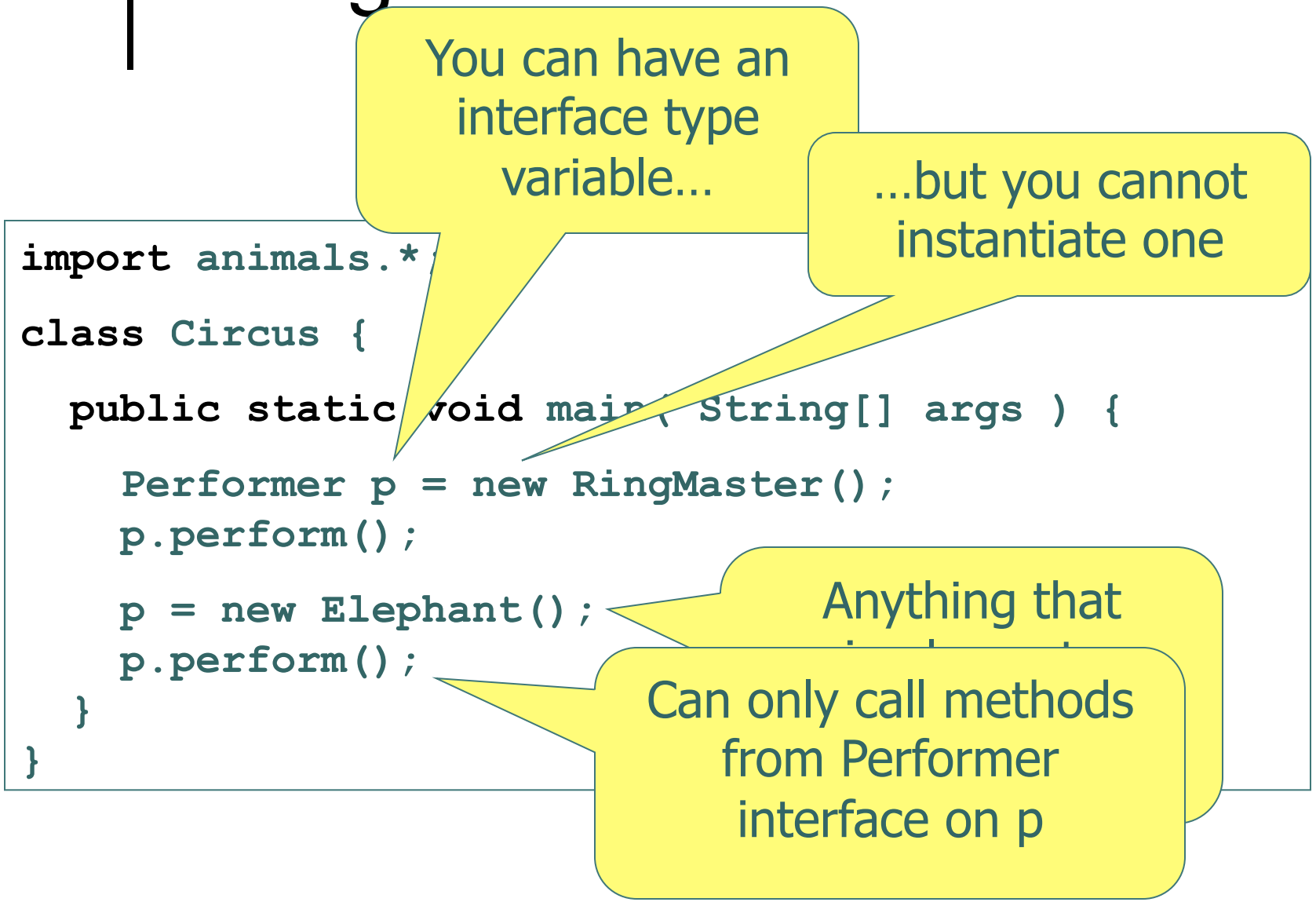
Implementing the Interface

Declare class as implementing interface

```
public class Elephant extends Animal
                      implements Performer {

   public void makeNoise() { trumpet(); }

   public void perform() { walk(); trumpet(); }

   …
}
```

Implement all methods from Performer

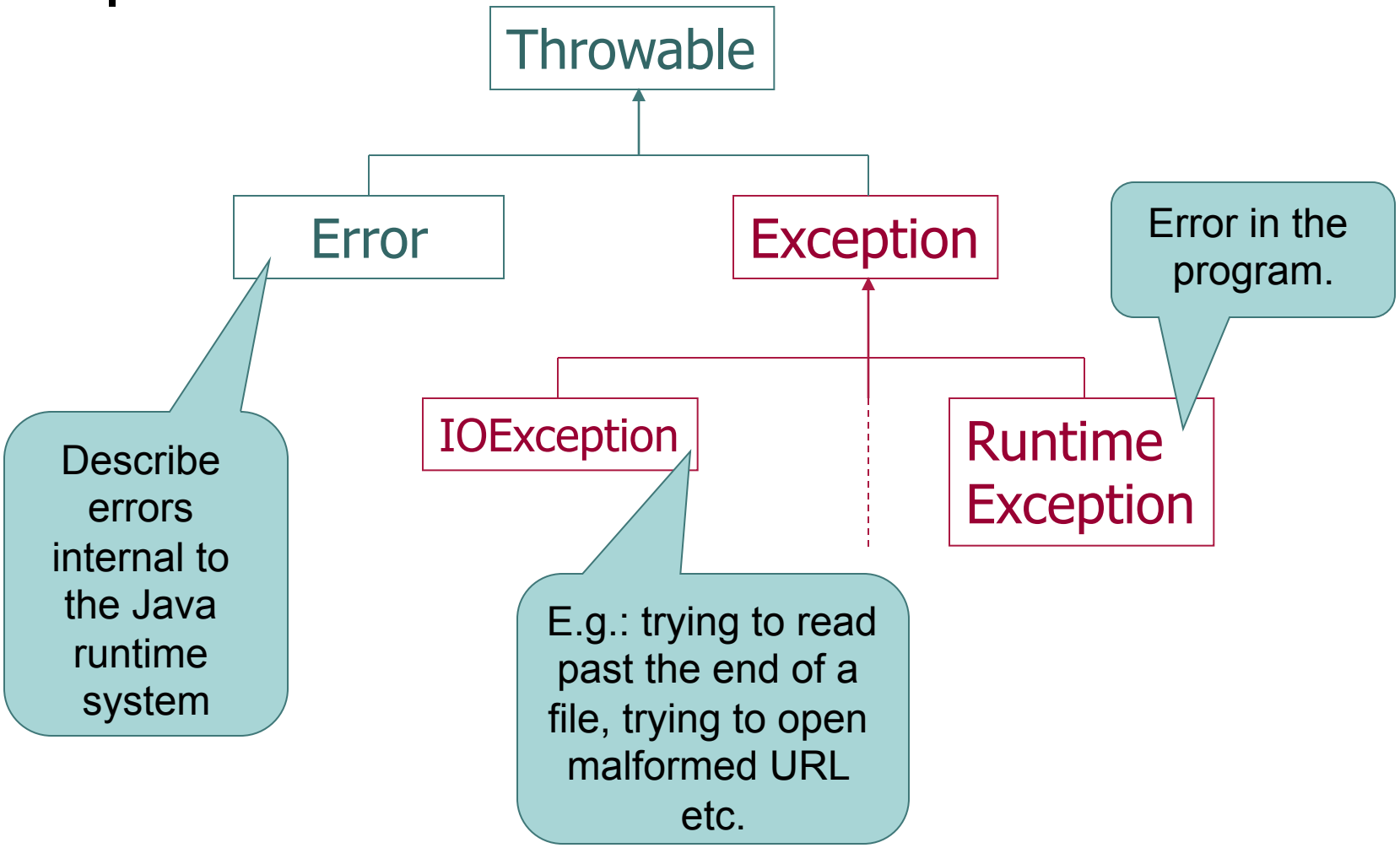
Using Performers



6. Exceptions

- Sometimes a method may fail
  - *Exceptional* behaviour
- Users expect program to behave sensibly:
  - Return to steady state and allow other commands.
  - Save work and terminate program.
- Methods can throw an Exception to signal that something is wrong
  - Catch and handle this special case

Exceptions


Throwable
Error
Exception
Error in the program.
Describe errors internal to the Java runtime system
IOException
Runtime Exception
E.g.: trying to read past the end of a file, trying to open malformed URL etc.

Exceptions

- **When can an exception happen?**
- 1. You call a method that throws a checked exception, for example, the readLine method of the BufferedReader class.
- 2. You detect an error and throw a checked exception with the *throw* statement.
- 3. You make a programming error, such as a[-1] = 0 that gives rise to an unchecked exception such as an ArrayIndexOutOfBoundsException.
- 4. An internal error occurs in the virtual machine or runtime library.

Exceptions, scenario 1

Method header: type to be returned

Plus what can go wrong.

```
public String readLine() throws
  IOException
```

(Method of the BufferedReader class of the standard library).

Exceptions, scenarios 3 and 4

- Internal Java errors (Error hierarchy) need not be advertised.
- Similarly, exceptions inheriting from RuntimeException should not be advertised. Instead you should try to fix the suspect code.

```
class MyAnimation
{
    . . .
    void drawImage(int i)
        throws ArrayIndexOutOfBoundsException // NO!!!
    {
        . . .
    }
}
```

Exceptions, scenario 2

- As with classes that are part of the API, you declare that your methods may throw an exception with a *throws* specification:

```
class MyAnimation
{
   . . .

   public Image loadImage(String s) throws IOException
   {
      . . .
   }
}

class MyAnimation
{
   . . .
   public Image loadImage(String s)
      throws EOFException, MalformedURLException
   {
      . . .
   }
}
```

Exceptions, scenario 2

- Suppose you have a method readData(), reading a file which should be of *len* characters.
- You may want readData() to throw an exception if EOF is met before that.
- You search the API and find that EOFException signals unexpected end of file.

Exceptions, scenario 2

```
String readData(BufferedReader in) throws EOFException
{
   . . .
   while (. . .)
   {
      if (ch == -1) // EOF encountered
      {
         if (n < len)
            throw new EOFException();
      }
      . . .
   }
   return s;
}
```

Exceptions, scenario 2

- 1. Find an appropriate exception class;
- 2. Make an object of that class;
- 3. Throw it.

```
throw new EOFException();
```

or, if you prefer,

```
EOFException e = new EOFException();
throw e;
```

Throwing your own exceptions

What happens if you cannot find an appropriate exception?

```
class Test {
   …
   public void unlockDoor() throws
                              AlreadyOpenException {
   if ( open ) {
      throw new AlreadyOpenException();
   }
}
class AlreadyOpenException extends Exception {}
```

Catching exceptions

- If an exception occurs that is not caught anywhere in a nongraphical application, the program will terminate and print a message to the console giving the type of the exception and a stack trace.
- A graphics program (both an applet and an application) prints the same error message, but the program goes back to its user interface processing loop.

Documentation for FileReader

FileReader

public FileReader(File file) throws FileNotFoundException

Creates a new FileReader, given the File to read from.

Parameters:

file - the File to read from

Throws:

FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

Try and Catch

You define a method (readFile) which calls a method (FileReader constructor) that is known to throw an exception. You either try-and-catch it…

```
class Test {
  …
  public void readFile() {
    File myFile = new File( “data.txt” );
    try {
     FileReader fr = new FileReader( myFile );
    } catch ( FileNotFoundException fnfe ) {
      System.out.println( “File not found.” );
    }
  }
}
```

Alternatively, Throws

…or you specify that readFile throws an exception (because it calls FileReader) and let readFile's caller handle the exception (with a try-catch clause).

```
class Test {

  …

  public void readFile() throws
                          FileNotFoundException {

    File myFile = new File( filename );

    FileReader fr = new FileReader( myFile );

  }
}
```

…and Finally

```
public void readFile( filename ) {
  File myFile = new File( filename );
  FileReader fr = new FileReader( myFile );
  try {
    int i = fr.read();
  } catch ( IOException ioe ) {
      System.err.println( "Error reading file" );
  } finally {
    try { if ( fr != null ) fr.close(); }
    catch ( IOException ioe ) {
      System.err.println( "Error closing stream" );
    }
  }
}
```

The code inside the Finally clause is executed under all circumstances. Either an exception is thrown, caught or none of the above.