Department of Computer Science
University of Bristol

COMSM0086– Object-Oriented Programming with Java

Abstract Classes, Interfaces, DoD, Static Elements and Arrays!

Simon Lock | simon.lock@bristol.ac.uk
Sion Hannuna | sh1670@bris.ac.uk

Recap: References

Classes and Reference Types

- every object belongs to a class
- classes act like types; for instance, references to an object are given a particular type when we declare it:

```
...
    Robot c3po = new Robot("C3PO");
...
```

the reference to ‘type’ Robot

here we specify the type of object the reference c3po can point to

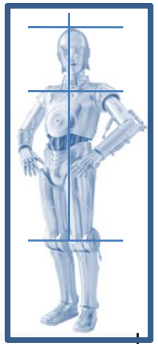
the constructor called must then relate to the type we specified

```
class Robot {
  String name;
  int numLegs;
  float powerLevel;

 Robot(String productName) {
    name = productName;
    numLegs = 2;
    powerLevel = 2.0f;
  }

  void talk(String phrase) {
    if (powerLevel >= 1.0f) {
      System.out.println(name + " says " + phrase);
      powerLevel -= 1.0f;
    } else {
      System.out.println(name +
      “ is too weak to talk.");
    }
  }

  void charge(float amount) {
    System.out.println(name + " charges.");
    powerLevel += amount;
  }
}
```



→ What effects can we achieve by introducing sub-class relationships?

RECAP: Sub-classes - inheritance

Fighting Code Duplication

- Problem: you have written a class (e.g. **Robot**), which almost does what you want, but requires some extensions
- Idea: **extend** features from the existing class by creating a **child class** that automatically receives all features of the parent class (e.g. **name**, **talk()**,…) without writing code again
- Implementation: you define a new class (e.g. **TranslationRobot**) inheriting **all** features from the existing parent class, but add or adapt features so that the new class does exactly what you want
- Result: leads to **DRY** (do-not-repeat-yourself) code where each feature has a **single code source**

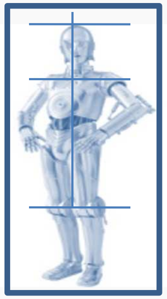
parent class Robot provides all its features to the child class

```
class Robot {
  String name;
  int numLegs;
  float powerLevel;

 Robot(String productName) {
    name = productName;
    numLegs = 2;
    powerLevel = 2.0f;
  }

  void talk(String phrase) {
    if (powerLevel >= 1.0f) {
      System.out.println(name+" says "+
                         phrase);
      powerLevel -= 1.0f;
    } else {
      System.out.println(name +
      “ is too weak to talk.");
  } }

  void charge(float amount) {
    System.out.println(name+" charges.");
    powerLevel += amount;
} }
```



‘extends’ signals inheritance from Robot class

```
public class TranslationRobot extends Robot {
  // class has everything that Robot has implicitly
  String substitute; //and more features

  TranslationRobot(String substitute) {
    this.substitute = substitute;
  }

  void translate(String phrase) {
    this.talk(phrase.replaceAll("a", substitute));
  }

  @Override
  void charge(float amount) { //overriding
    System.out.println(name + " charges double.");
    powerLevel = powerLevel + 2 * amount;
} }
```

added method here

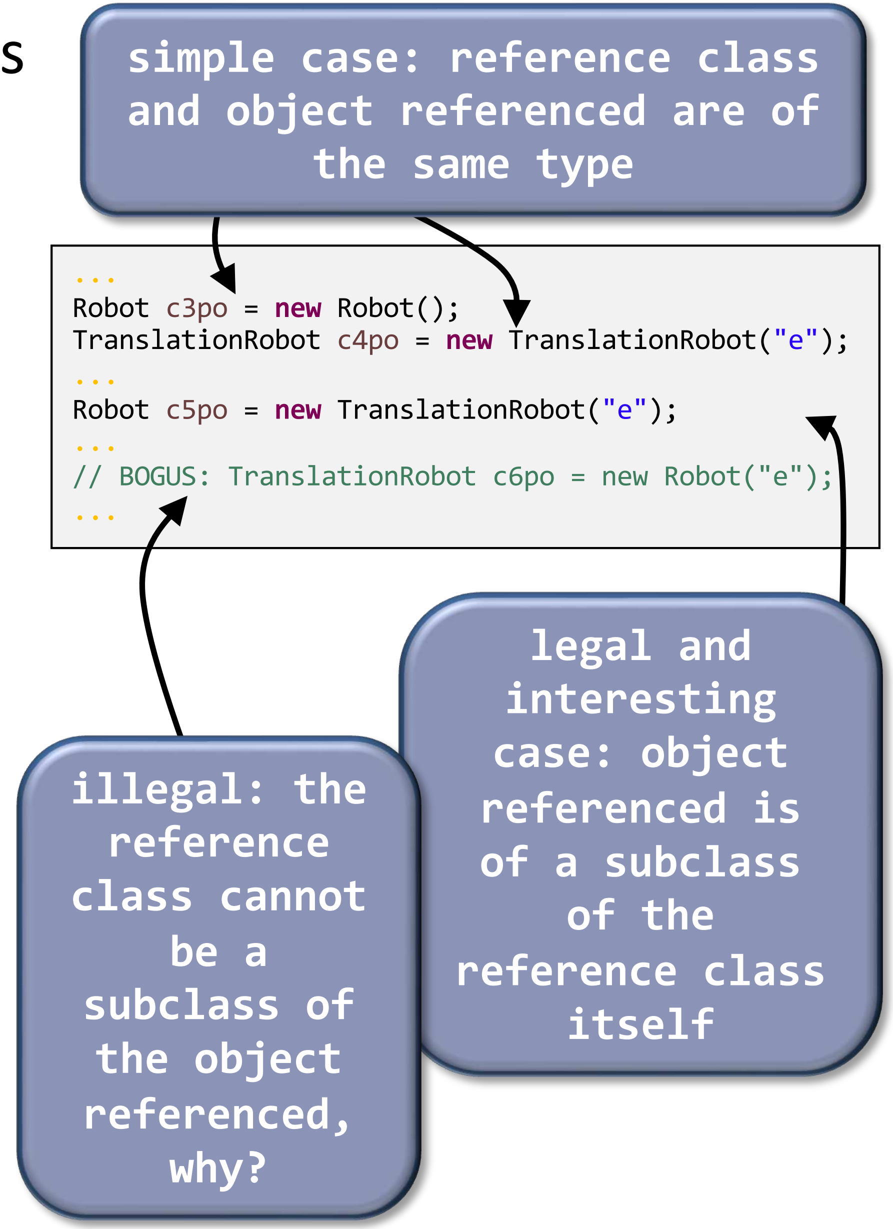
parent method is replaced here

Polymorphism

(... a first meeting with a powerful chimera ...)

Polymorphism (one reference, ‘many shapes’)

- every reference belongs to a class (the one that the reference was defined as)
- however, a reference can be made to **any object of a sub-class** of the reference’s class
- this does not change the **reference’s class**
- the principle that arises from the fact that one reference can refer to (potentially) various different classes is called **polymorphism**
- in essence, it lets you use an object of a sub-class as if it was an object of some super class



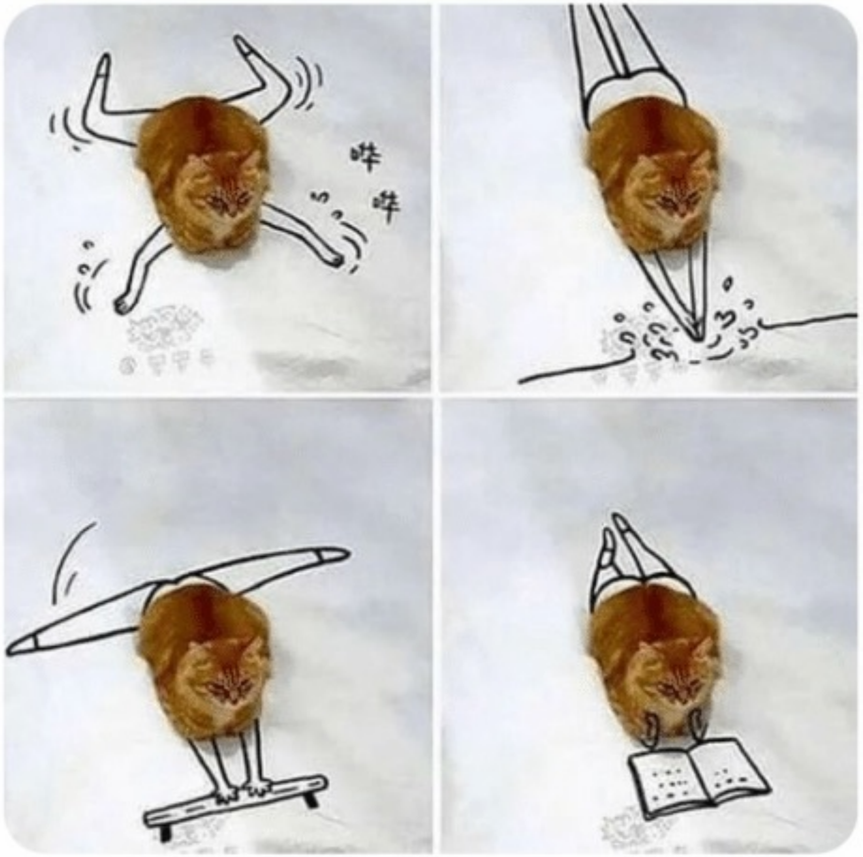
simple case: reference class and object referenced are of the same type

```
...
Robot c3po = new Robot();
TranslationRobot c4po = new TranslationRobot("e");
...
Robot c5po = new TranslationRobot("e");
...
// BOGUS: TranslationRobot c6po = new Robot("e");
...
```

illegal: the reference class cannot be a subclass of the object referenced, why?

legal and interesting case: object referenced is of a subclass of the reference class itself

Inheritance



Same base class, multiple implementations

Mammal Magic

Let’s look at the classic mammals example …

Arrays in Focus: Creating, Initialising and Iterating

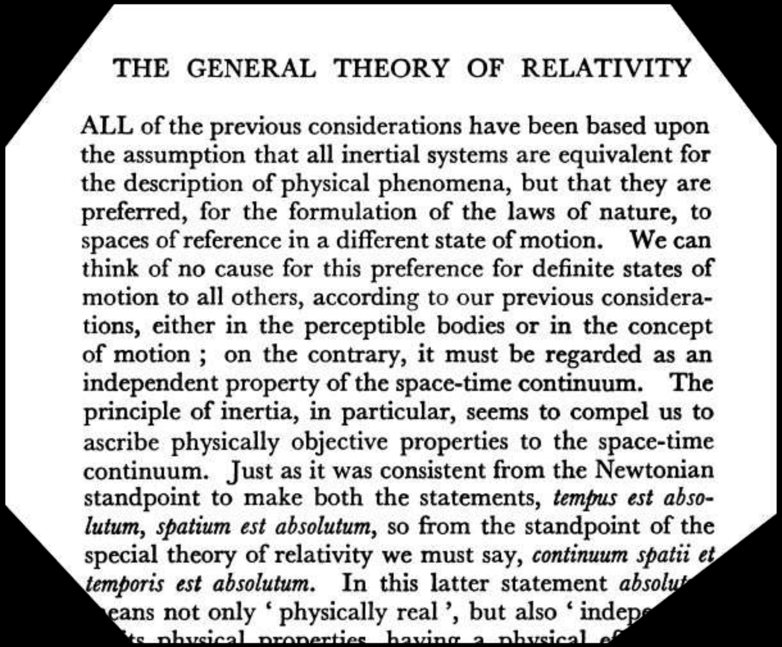
- arrays (in Java) are objects too...

```
public class RobotArrays {
  public static void main (String[] args) {
    Robot[] robotsA = new Robot[3]; //instantiate array of references to 3 Robot objects
    System.out.println(robotsA[0]); //at start, array locations carry null
    robotsA[0] = new Robot("C3PO"); //initialise entry at index 0
    robotsA[1] = new Robot("C4PO"); //initialise entry at index 1
    robotsA[2] = robotsA[0];         //initialise with same reference as index 0
    Robot[] robotsB = {              //neat initialisation syntax using {...}
              new Robot("C5PO"),
              robotsA[0],
              robotsA[1]
            };
    System.out.println(robotsB.length); //print size of array robotsB
    for (Robot robot : robotsB)         //loop through entries, assign current to robot
      System.out.println(robot.name);   //print name of current element
  }
```

REMEMBER: This is an Iterator using the ":" notation, which provides a reference "robot" to each object held in the array turn-by-turn (The reference is not a one to the array location itself, which is not an object.)

...beware, it is the wrath of the null that you need to defend against in your programming...

Abstract classes

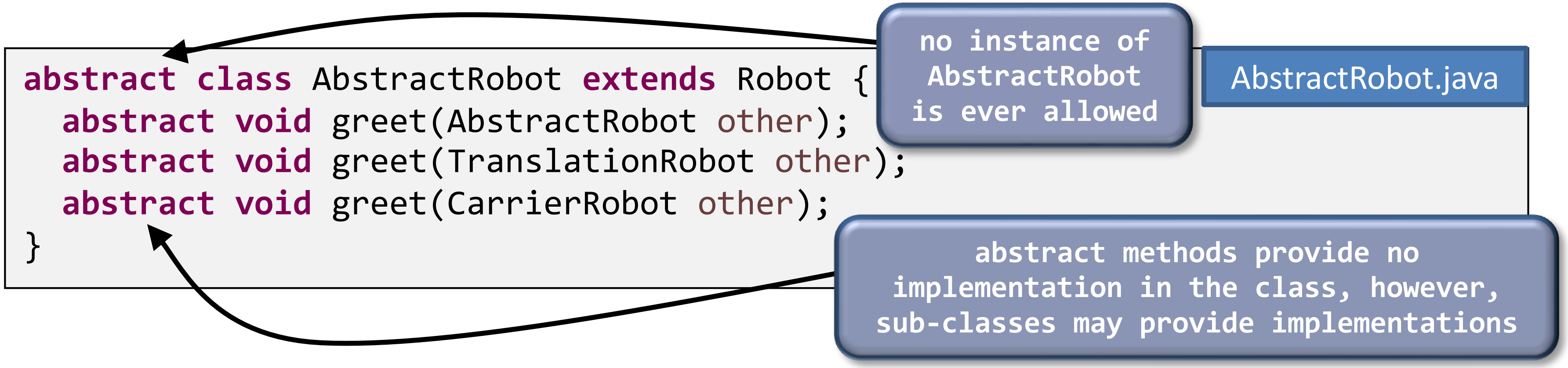


THE GENERAL THEORY OF RELATIVITY

ALL of the previous considerations have been based upon the assumption that all inertial systems are equivalent for the description of physical phenomena, but that they are preferred, for the formulation of the laws of nature, to spaces of reference in a different state of motion. We can think of no cause for this preference for definite states of motion to all others, according to our previous considerations, either in the perceptible bodies or in the concept of motion ; on the contrary, it must be regarded as an independent property of the space-time continuum. The principle of inertia, in particular, seems to compel us to ascribe physically objective properties to the space-time continuum. Just as it was consistent from the Newtonian standpoint to make both the statements, *tempus est absolutum*, *spatium est absolutum*, so from the standpoint of the special theory of relativity we must say, *continuum spatii et temporis est absolutum*. In this latter statement *absolut* eans not only ' physically real ', but also ' indepe ts physical properties, having a physical ef

Abstract Classes, Abstract Methods

- to prevent us from making instances of a class we apply the **abstract** keyword to the class
- abstract classes are often ones that are purely conceptual without any instances (e.g. a generic **Shape**, an **AbstractRobot**)



AbstractRobot.java

```java
abstract class AbstractRobot extends Robot {
  abstract void greet(AbstractRobot other);
  abstract void greet(TranslationRobot other);
  abstract void greet(CarrierRobot other);
}
```

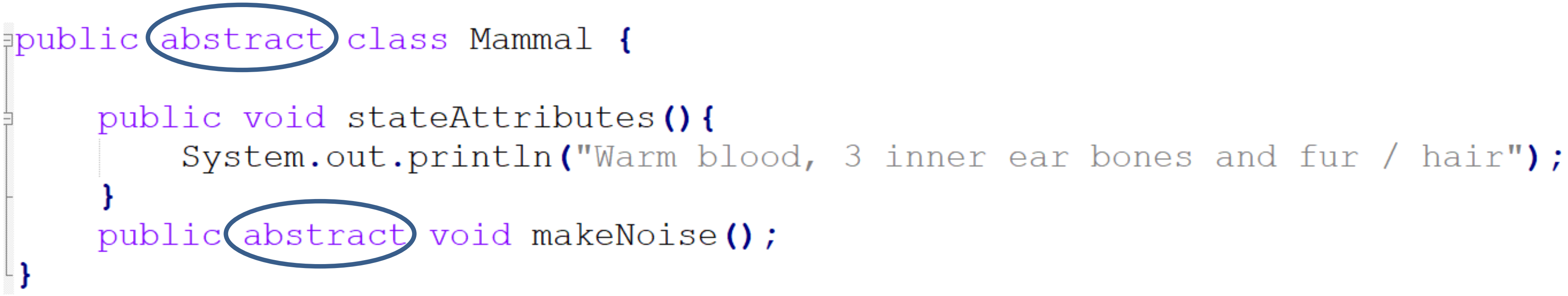
no instance of AbstractRobot is ever allowed

abstract methods provide no implementation in the class, however, sub-classes may provide implementations

- usually an **abstract** class contains **abstract** methods, that is methods which are declared, but supply no implementation (any non-abstract sub-class is forced to implement **all** these methods)
- a class with one or more abstract methods **must be** declared abstract itself

Fighting code duplication with inheritance

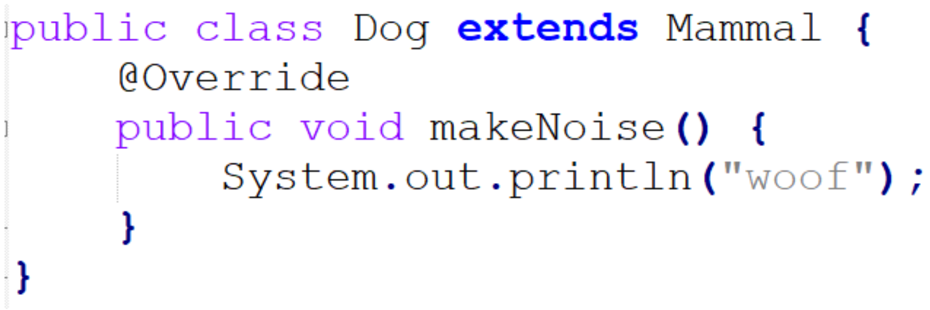
Define a parent class which defines attributes and methods common to the subclasses you plan to create:



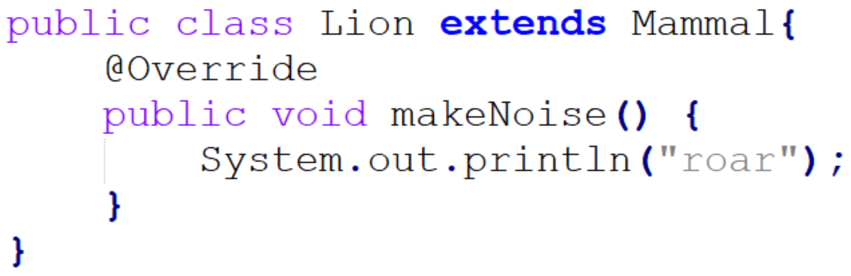
```java
public abstract class Mammal {

    public void stateAttributes(){
        System.out.println("Warm blood, 3 inner ear bones and fur / hair");
    }
    public abstract void makeNoise();
}
```

Extend parent class with subclasses which add and / or override the parent class's characteristics:

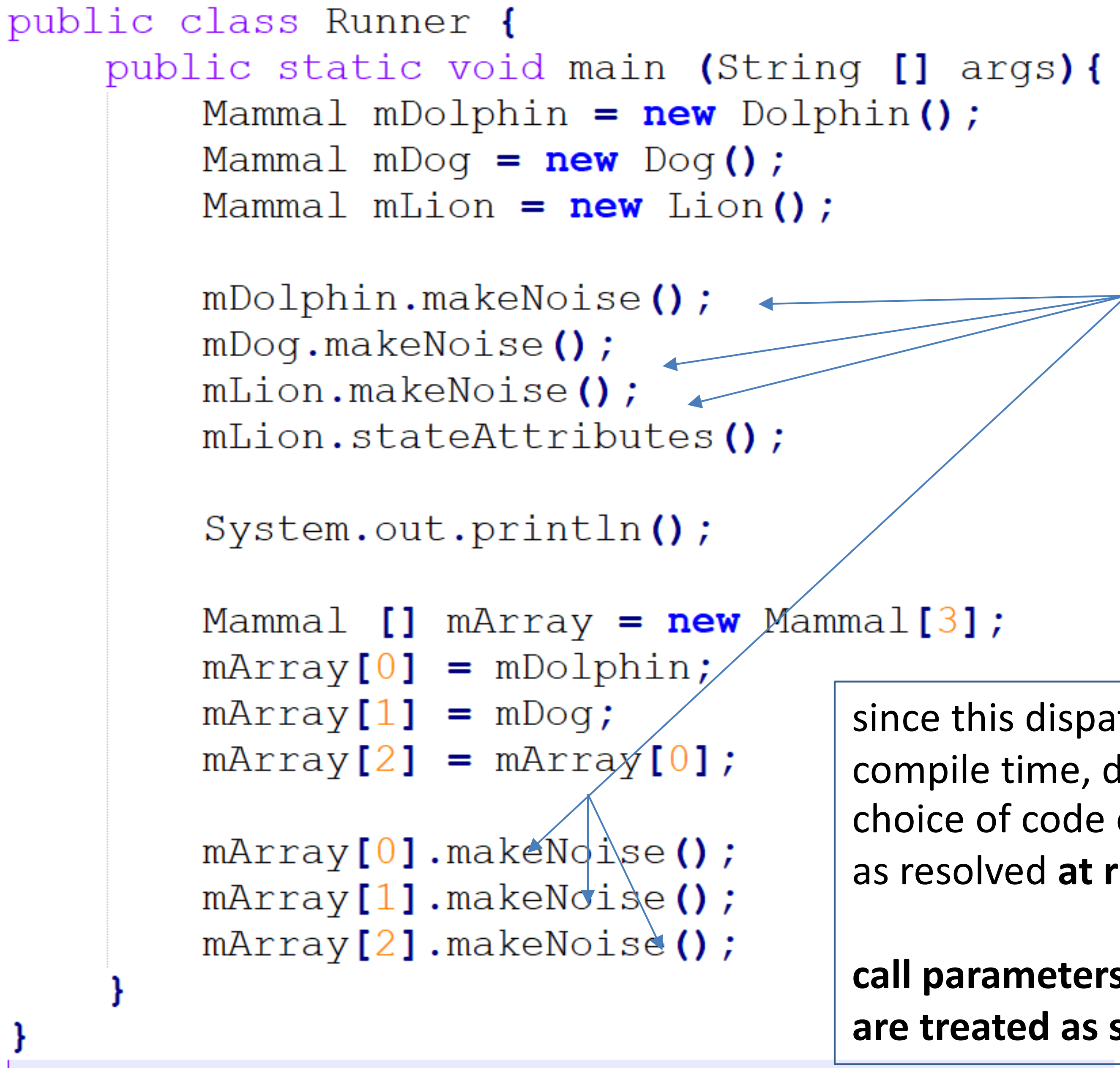


```java
public class Dog extends Mammal {
    @Override
    public void makeNoise() {
        System.out.println("woof");
    }
}
```



```java
public class Lion extends Mammal{
    @Override
    public void makeNoise() {
        System.out.println("roar");
    }
}
```

Polymorphism in action – single dynamic dispatch



```
public class Runner {
    public static void main (String [] args){
        Mammal mDolphin = new Dolphin();
        Mammal mDog = new Dog();
        Mammal mLion = new Lion();

        mDolphin.makeNoise();
        mDog.makeNoise();
        mLion.makeNoise();
        mLion.stateAttributes();

        System.out.println();

        Mammal [] mArray = new Mammal[3];
        mArray[0] = mDolphin;
        mArray[1] = mDog;
        mArray[2] = mArray[0];

        mArray[0].makeNoise();
        mArray[1].makeNoise();
        mArray[2].makeNoise();
    }
}
```

when an overridden method is called via a reference, the actual method to execute is selected based on the **type of the object** referenced, not the reference type

this is known as `**dynamic method dispatch**’

since this dispatch decision cannot be made at compile time, dynamic dispatch refers to the choice of code execution (i.e. the method call) as resolved **at runtime**

call parameters, even if they are references, are treated as static …

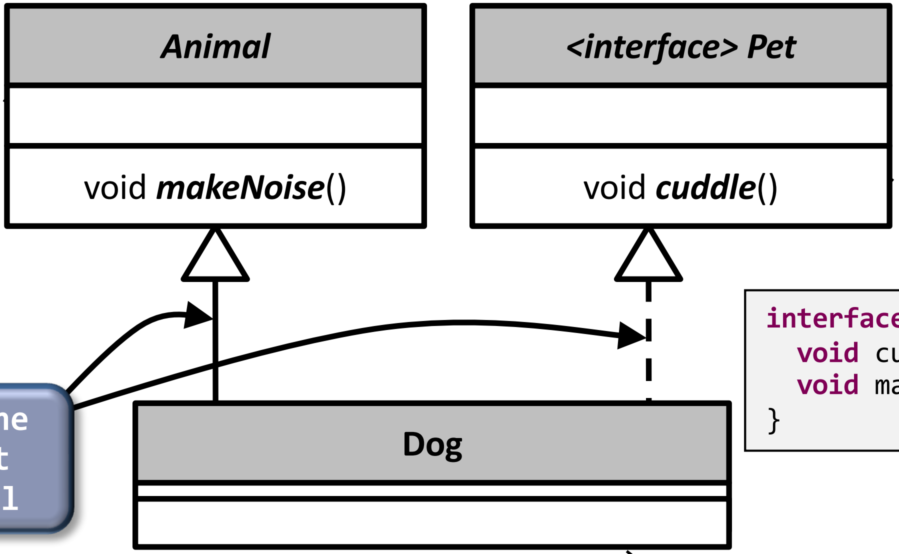
Interfaces

Interfaces

- an **`interface`** is a reference type in Java, stored in a .java file and can be seen as a set of only **`abstract`** methods
- thus, you cannot instantiate an interface and an interface does not contain any constructors
- when a class **`implements`** an **`interface`** it inherits all the (implicitly abstract) methods of the interface
- an interface may also contain constants, default and static methods, and nested types; implementations may exist only for static and default methods (more on this later)
- an interface **describes behaviours** that a class implements
- any class can implement multiple interfaces and an interface itself can extend multiple interfaces

Interface Example

Why do interfaces guarantee that there is no name clash?



```
abstract class Animal {
  abstract void makeNoise();
}
```

a class can inherit from one parent class and implement multiple interfaces as well

```
interface Pet {
  void cuddle();   //implicitly abstract
  void makeNoise();//implicitly abstract
}
```

for a Dog class to be a concrete (i.e. non-abstract) class it has to implement all abstract methods from all interfaces it implements and classes it extends

```
class Dog extends Animal implements Pet {

  @Override
  public void cuddle() { // from Pet
    System.out.println("Smelly cuddle!");
  }

  @Override
  public void makeNoise() { // from Animal
    System.out.println("Woof!");
} }
```

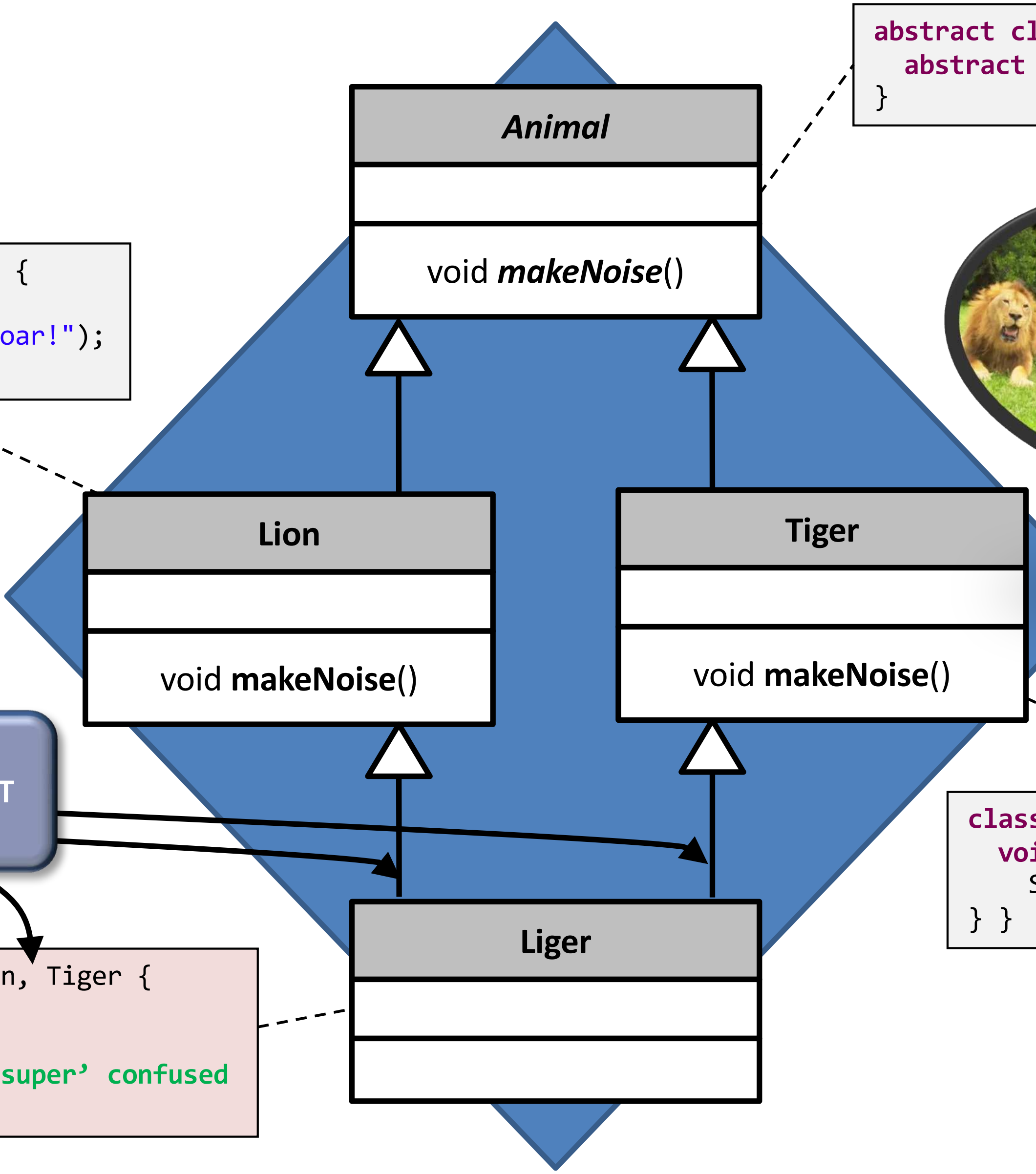
it is ok to have a method name identical to the one of another interface or even class defined in an interface since there is no implementation provided

slide idea originally by N. Wu

Deadly Diamond of Death

Deadly Diamond of Death (DDD)

```
abstract class Animal {
  abstract void makeNoise();
}
```



```
class Lion extends Animal {
  void makeNoise() {
    System.out.println("Roar!");
} }
```

multiple inheritance is NOT allowed in Java

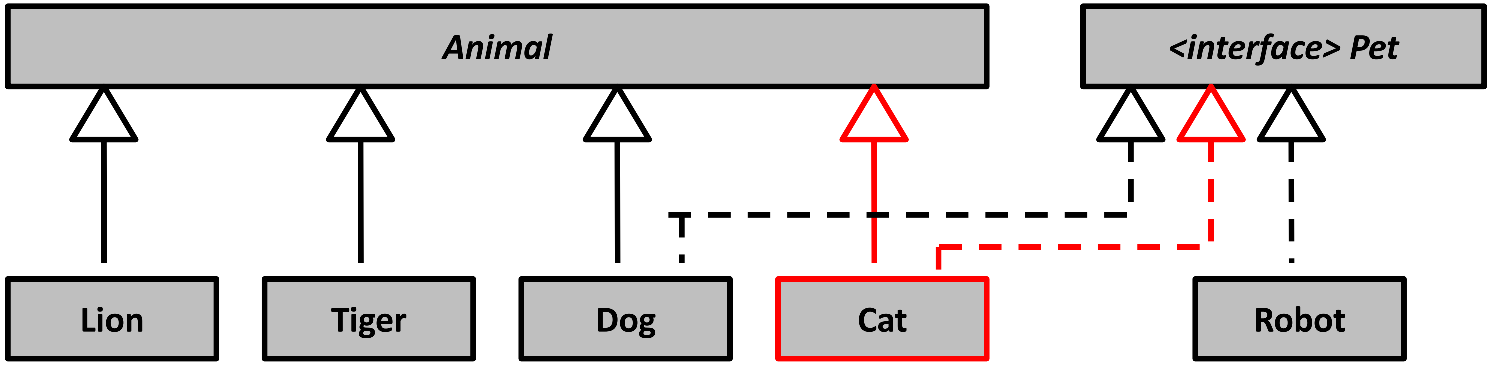
```
class Tiger extends Animal {
  void makeNoise() {
    System.out.println("Grrh!");
} }
```

```
class Liger extends Lion, Tiger {
  void makeNoise() {
    super.makeNoise();
    // Liger would be ‘super’ confused
} }
```

slide idea originally by N. Wu

Associating Classes to Multiple Concepts

- in multiple inheritance there is **no simple way** to resolve calls to **super** (one could use separate namespaces etc)
- in response, Java **forbids all** multiple inheritance:
  a class can only inherit from **one** parent
- `Liger' may still be implemented via **two has-a** relationships
- yet, we still would like a language construct that associates a single class with **several** parent concepts:



slide idea originally by N. Wu

Static Elements

(aka CLASS ELEMENTS)

(RESIST THEM)

Static Elements I

- like for **main**, elements declared as **static** are associated to a class, not actual object instances

RobotWorld.java

```
class RobotWorld {

  public static  void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    c3po.talk("'Hello, Java!'");
  }
}
```

the main method is a static element associated to the RobotWorld class, not individual objects – it can be called without any object instance

- **static** elements may be attributes, methods, blocks or nested classes
- **static** elements are shared between all instances of that class, **they can be accessed without instantiation**
- for an object oriented approach, avoid using statics unless you have a very good reason for it

Static Elements II

- **static** methods can access **static** data and can change the value of it
- **static** methods cannot use non-static data members or call non-static methods directly
- a **static** code block can be used to initialize the **static** data members (since constructors cannot do the job)
- a **static** code block is executed before the main method at the time of classloading, at which time all **static** attributes are allocated their memory

Staying in C-World

- you can pretend you’re not in Java and have nothing to do with objects
- to do that use the **static** keyword in front of your methods ~~functions~~ and attributes ~~variables~~
- if you do this, you demonstrate very nicely that you’ve *completely missed the point of* Java and object-orientation
- do bad things like:

```
class UseCWorld {


  public static  void main (String[] args){
    CWorld cw = null;
    for (int i = 1; i < 10; i = i + 1 ) {
          System.out.println( cw.fib(i) );
    }
  }
}
```

```
// DO NOT WRITE   CODE
// OF THIS STYLE, MOVE
// OUT OF C-WORLD PLEASE

class CWorld {
  static int x;
  static int y;

   static { x = 0; y = 10; }

  static int fib(int n) {
    switch (n) {
      case  0: return 1;
      case  1: return 1;
      default:
        return fib (n - 1) + fib (n - 2);
    }
  }

  public static void main (String[] args) {
    for (int i = x; i < y; i = i + 1 ) {
      System.out.println( fib(i) );
    }
  }
}
```