Some slides contain lots of animation; you must be in class to fully understand them.

More OO Programming

** inheritance
** abstract classes
** the **`Object`** class

Chapter 9 – “Big Java” book
Chapters 7+8 – “Head First Java” book
Chapters 8-10 – “Introduction to Java Programming” book
Chapter 3 – “Java in a Nutshell” book

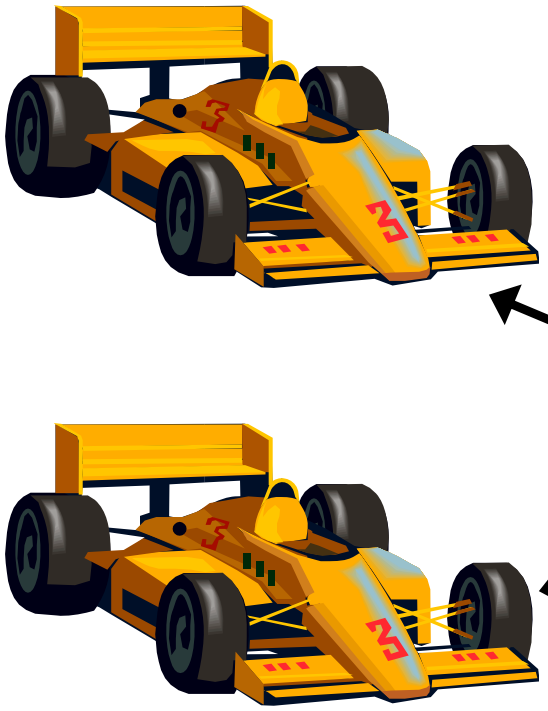
State and Instance Variables (Revision)

- Instance variables are what makes individual objects unique!
  - In other words, what makes them different from other objects (or instances) of the same class.

```
public class RaceCar {
   private String bodyColour;
   public RaceCar(String bodyColour) {
      this.bodyColour = bodyColour;
   }

  public static void main(String[] args) {
     RaceCar carOne = new RaceCar("Yellow");
     RaceCar carTwo = new RaceCar("Red");
  }
}
```

Comparing objects (Revision)

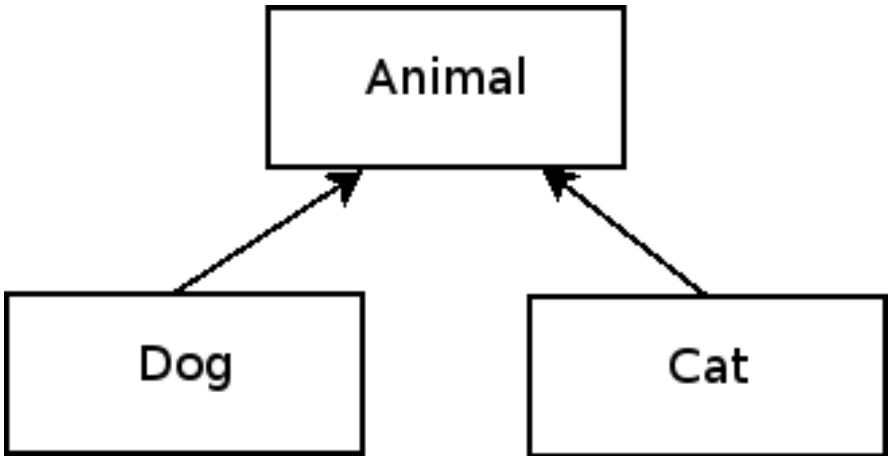
- Two objects with the same state are not the same object.
  - Otherwise, we would have to share the same car!

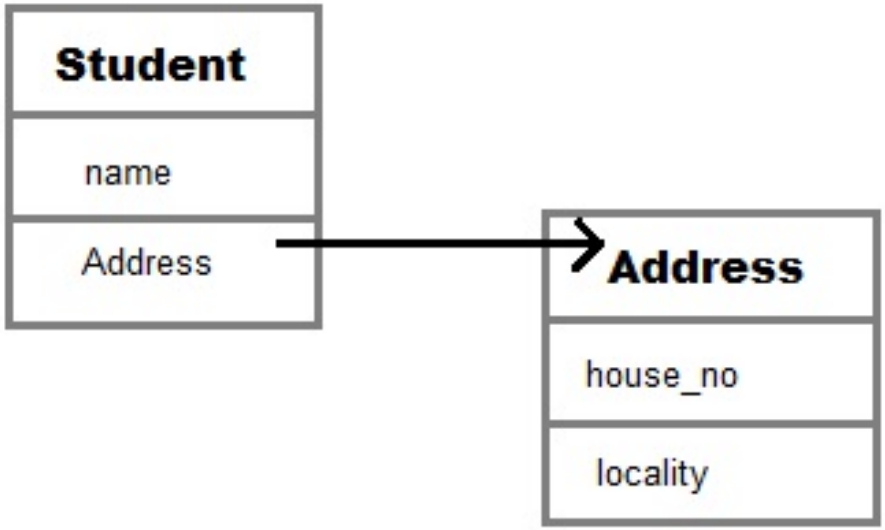


```
public class RaceCar {
   private String bodyColour;
   public RaceCar(String bodyColour) {
      this.bodyColour = bodyColour;
   }

  public static void main(String[] args) {
     RaceCar myCar = new RaceCar("Yellow");
     RaceCar yourCar = new RaceCar("Yellow");
   }
}
```

Class relationships

- There are two primary types of relationships between classes in Java:
  - aggregation (referred as has-a)
  - inheritance (referred as is-a)





Aggregation (1/2)

- Written as has-a
- There is no special Java keyword to indicate this relationship
  - Objects are instance variables in the class.
- Example:
  - The Car object contains four Tire objects, one Steering Wheel object, and so on.

Different types of the class Tire.

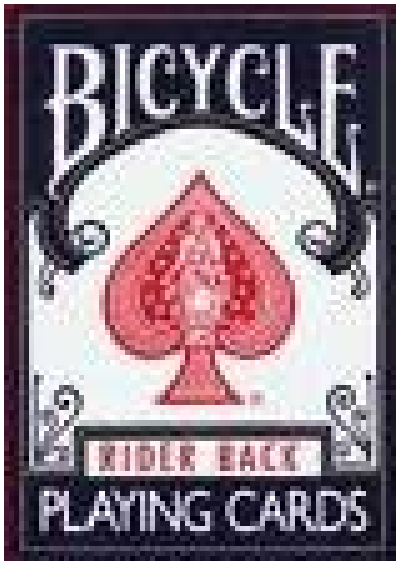
Aggregation (2/2)

```
public class RangeRover {
   public Tire[] array = new Tire[4];
   public void drive() {
      // do driving things ...
   }
}
```

has

The 4 Tires may initially have the same state, but they are not the same object.

Another example of aggregation

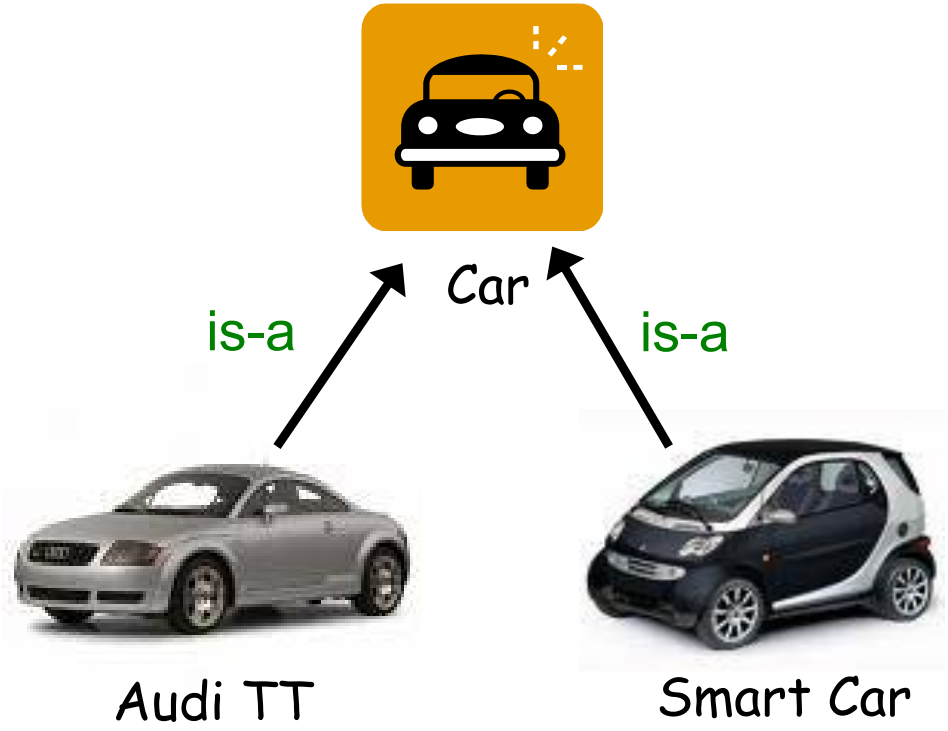


A Deck of Cards has/has-a cards

In fact, a deck of cards object has many card objects.

Inheritance (1/2)

- The inheritance relationship is also known as: Parent/Child or Superclass/Subclass.
  - written as is-a
  - via the **`extends`** keyword
  - Audi TT and Smart Car are subclasses of the class Car, making Car the superclass!
  - They are specialisations of the class ...



An Audi TT is-a car.
A Smart Car is-a car.

Inheritance (2/2)

- Subclasses inherit the properties (attributes and operations) of their superclass.

```
public class Car {
   public String bodyColour;
   public void drive() {
      // do driving things ...
   }
}
```

```
public class SmartCar extends Car {

}
```

```
public class TestCars {
  public static void main(String[] args) {
     Car c = new Car();
     c.bodyColour = "Red";
     c.drive();

     SmartCar d = new SmartCar();
     d.bodyColour = "Blue";
     d.drive();
  }
}
```

Are the highlighted lines allowed?

… and things for you to try out!

Doctor

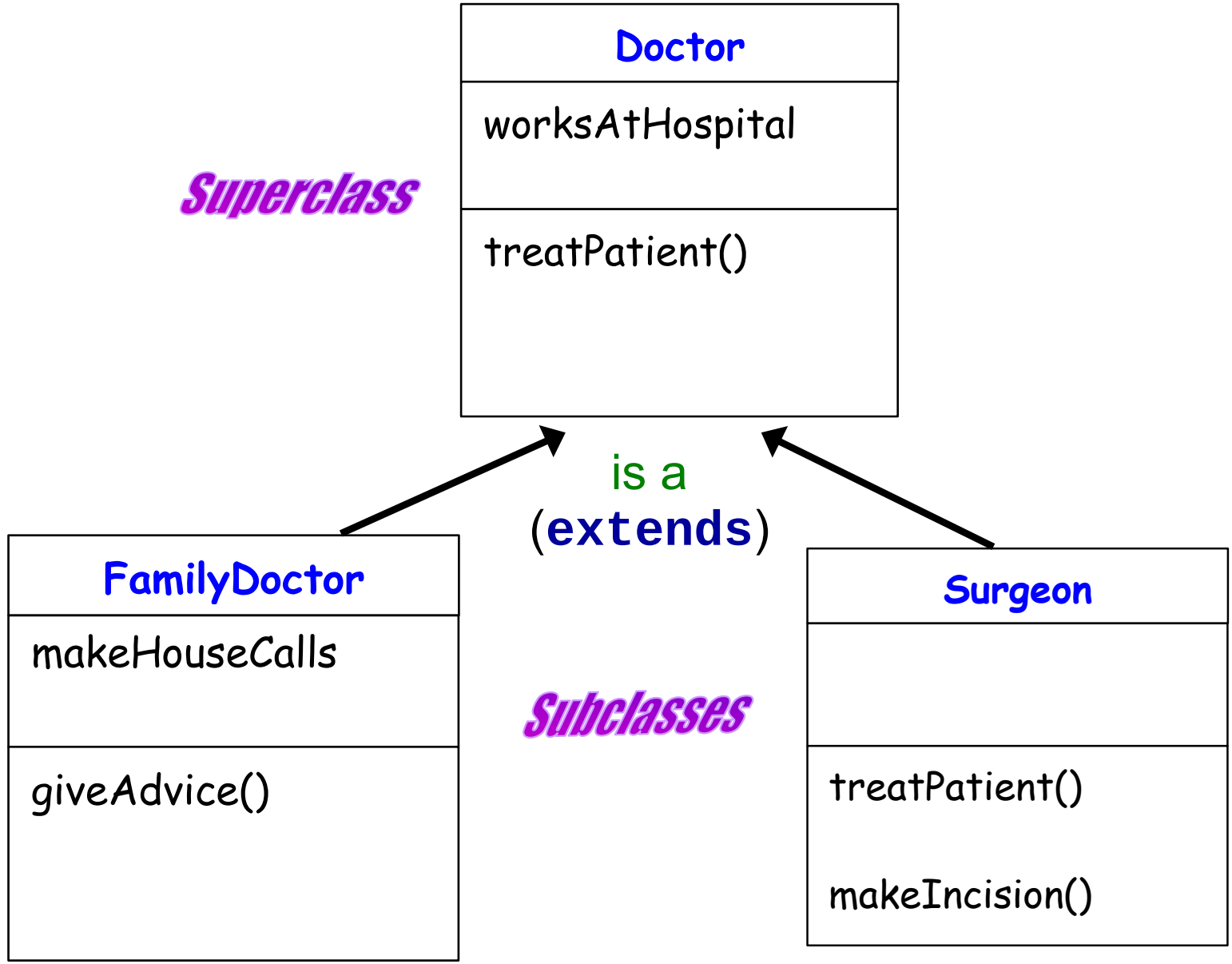
- To provide specialisations, subclasses override methods that they inherit from the superclass.

```
public class Doctor {
  boolean worksAtHospital;
  void treatPatient() {
    // perform a checkup
    System.out.println("checkup");
  }
}
```

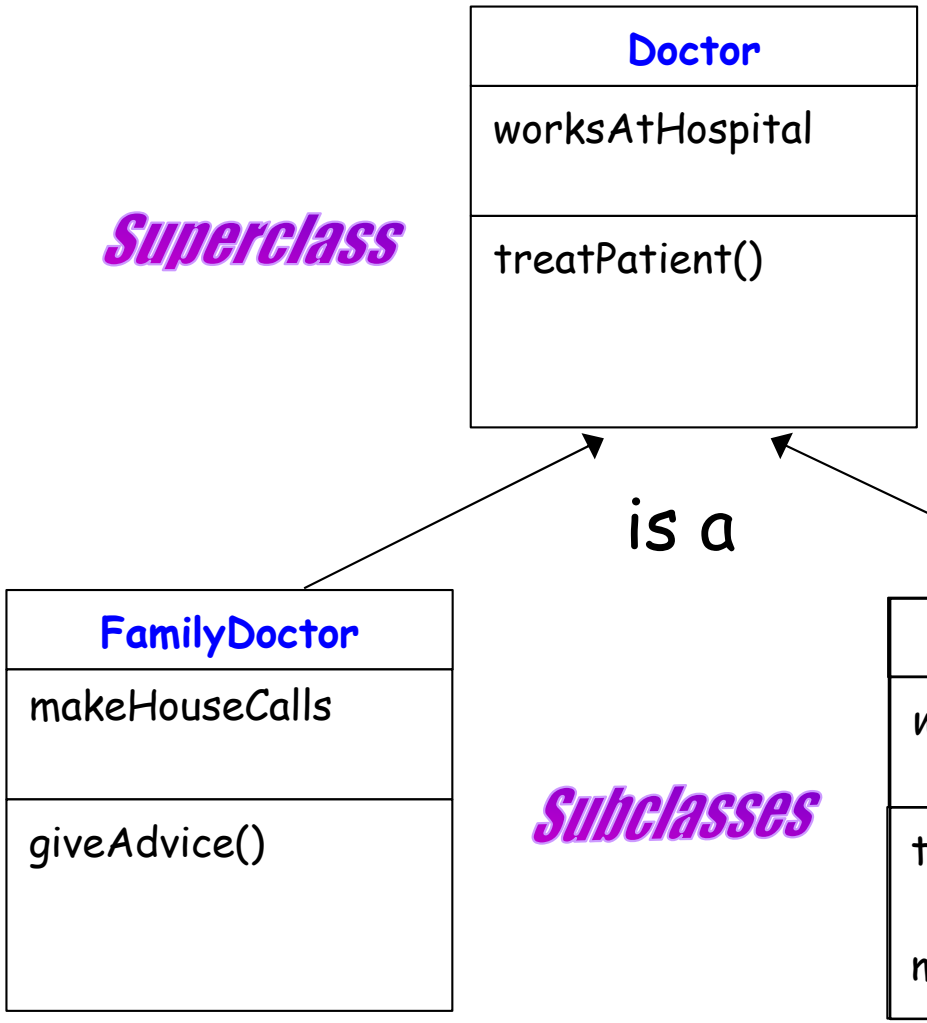
```
public class FamilyDoctor
                    extends Doctor {
  boolean makesHouseCalls;
  void giveAdvice() {
    // perform a checkup
    System.out.println("advise");
  }
}
```

```
public class Surgeon
                    extends Doctor {
  void treatPatient() {
    // perform surgery
    System.out.println("surgery");
  }
  void makeIncision() {
    // make incision
    System.out.println("incise");
  }
}
```

Inheritance Class Diagram for `Doctor` (1/4)



Inheritance Class Diagram for `Doctor` (2/4)

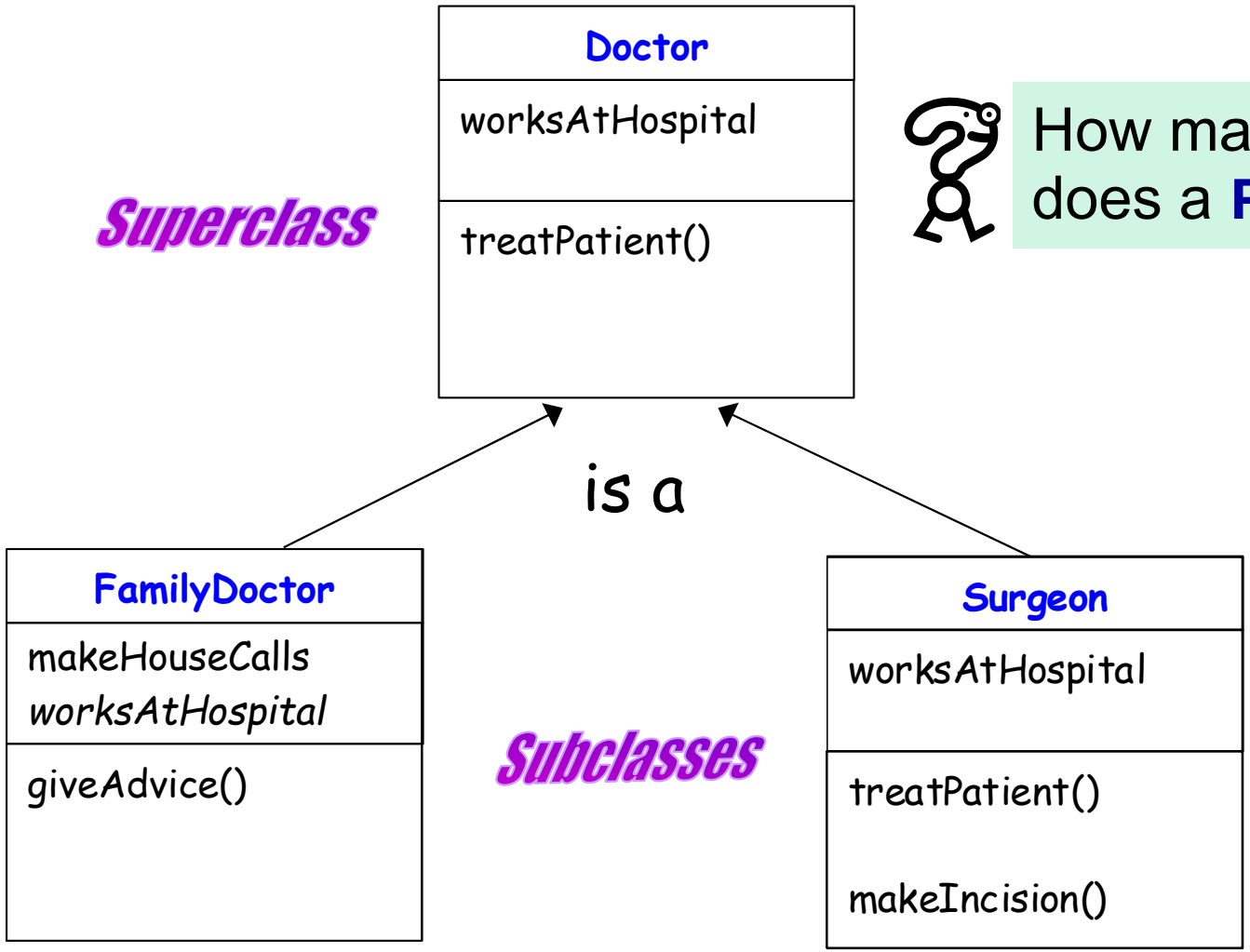


How many instance variables does a **`Surgeon`** have?

0? 1

(inherited from **`Doctor`**)

Inheritance Class Diagram for `Doctor` (3/4)

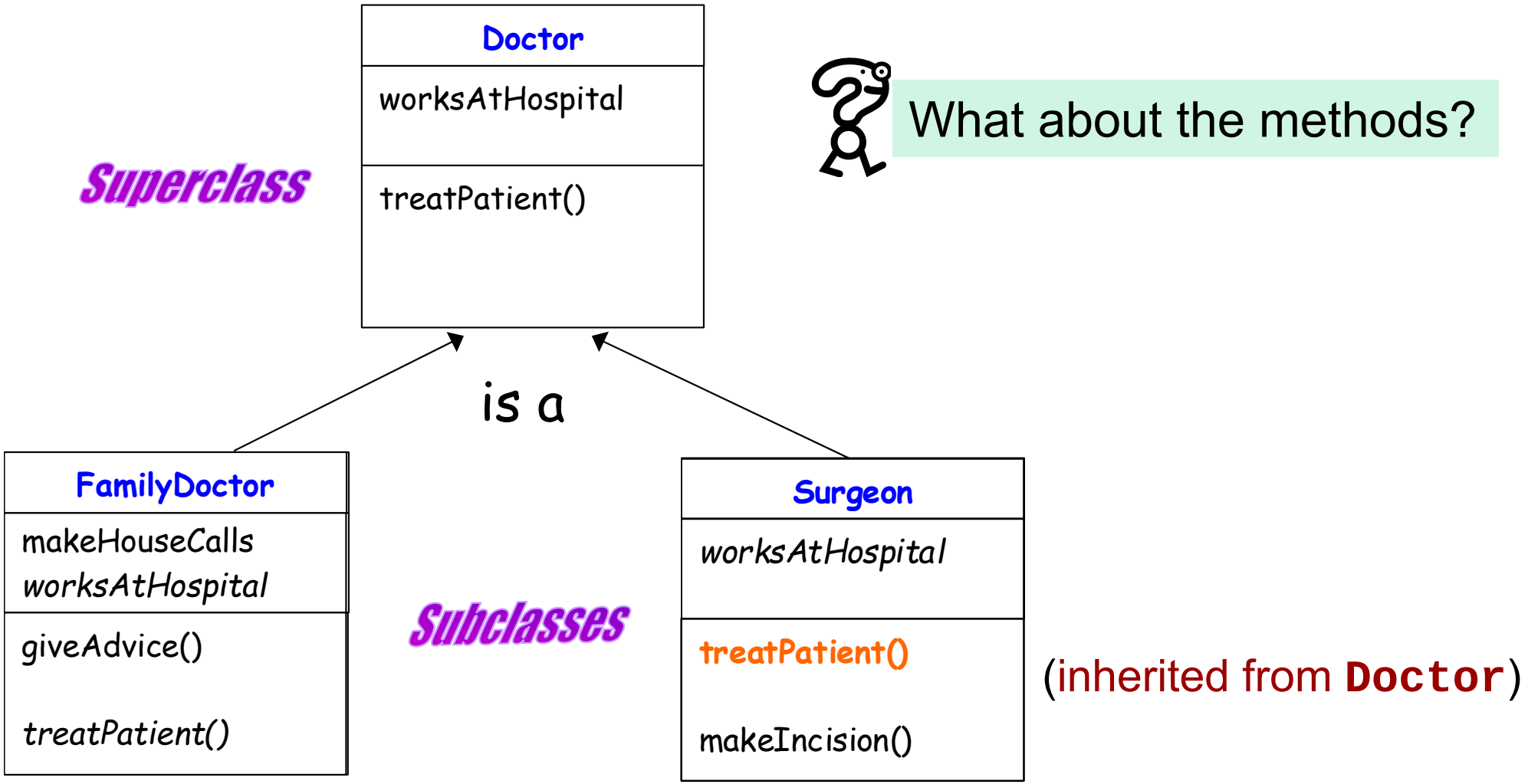


How many instance variables does a **`FamilyDoctor`** have?

1? 2

(inherited from **`Doctor`**)

Inheritance Class Diagram for `Doctor` (4/4)



… and things for you to try out!

Access Modifiers (and the inheritance relationship) [Revision]

- **`public`**
  - **`public`** instance variables and methods are inherited
- **`protected`**
  - **`protected`** instance variables and methods are inherited
- **`private`**
  - any **`private`** instance variables and methods are not inherited and cannot be seen by the subclass

Examples (1/2)

```
public class Parent {
   protected double cash;
   protected void spendMoney(double amount) {
     cash -= amount;
   }
   // do other Parent things
}
```

I control (and use) my cash. It is mine!

```
public class Child extends Parent {
   private void buyGadgets(double amount) {
      this.spendMoney(amount);
   }
  // do other Child things
}
```

✓ Allowed, since the **`Child`** has inherited this method.

I inherited money! I can spend it or add to it (lol) !

Examples (2/2)

```
public class AnotherClass {
  public static void main(String[] args) {
     Parent p = new Parent();
     p.spendMoney(1000000.32);

     Child c = new Child();
     c.buyGadgets(50.94);
  }
}
```

✓ ✗ Is allowed IF **`AnotherClass`** is in the same package (directory) as **`Parent`** – otherwise it is not allowed, as the variable is **`protected`** and **`AnotherClass`** is not a subclass of **`Parent`**!

✗ ✗ Not allowed in any circumstance, because the method is **`private`**.

… and things for you to try out!

Design Process

Look for objects that have common attributes and behaviours.

Design the class (*superclass*) that represents the common state and behaviour.

Decide if a subclass needs behaviours (*methods implementation*) that are specific to that particular subclass type.

Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behaviour.

Finish the class hierarchy.

IS-A / HAS-A

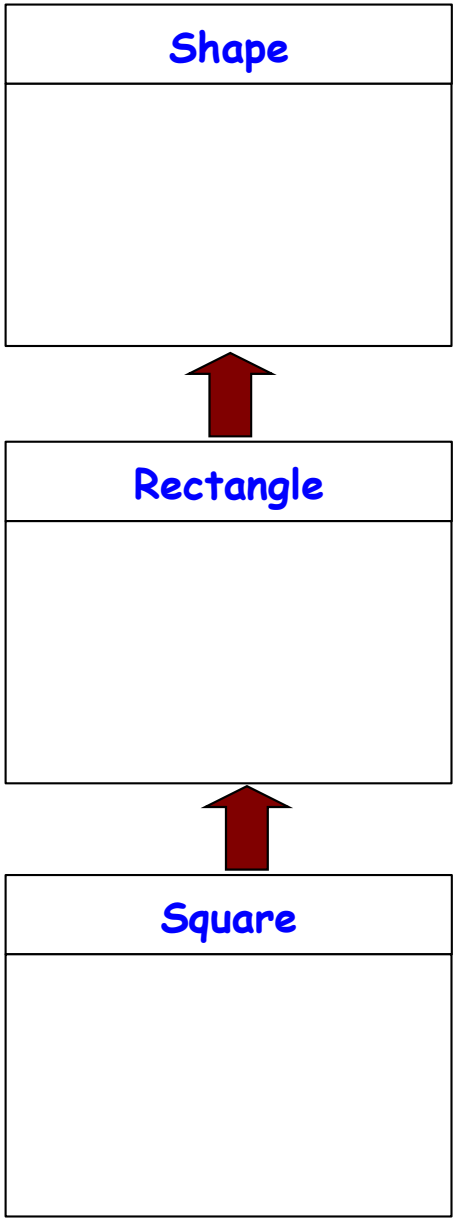
- Make sure you get the combination right!
  - Triangle IS-A Shape.
  - Cat IS-A Feline.
  - Refrigerator **`extends`** Kitchen?
    - Does not really work; Refrigerator IS-A Kitchen makes no sense.
    - There is a relationship but not an inheritance relationship, rather an aggregation relationship, i.e. Kitchen HAS-A Refrigerator.

Kitchen
Refrigerator fridge; Dishwasher washer; Sink sink; Stove s;

All of these objects are part of a **`Kitchen`** object.
The **`Kitchen`** has a reference to them (i.e. has-a), rather than inherits.

IS-A

- The inheritance chain:
  - If class B extends class A, then class B is class A.
  - If class C extends class B, then class C is class A and B.



is-a works all the way up

does not work the other way

A **Rectangle** is a **Shape**.

A **Shape** is not necessarily a **Square** or even a **Rectangle**.

A **Square** is a **Rectangle** and is a **Shape**.

A **Rectangle** is not necessarily a **Square**.

Inheritance

Avoids duplicating code

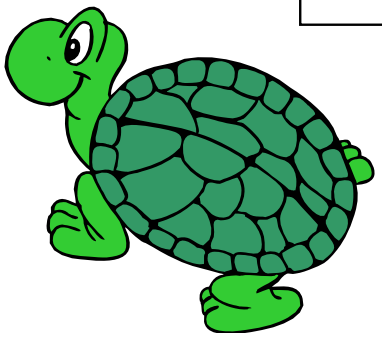
Allows specialisation
(You can be better, be different, be an individual!) *

* But yet still part of the same "group" ! ☺

Improving design through inheritance …

- Our two **`Rabbit`** and **`Turtle`** classes from before:

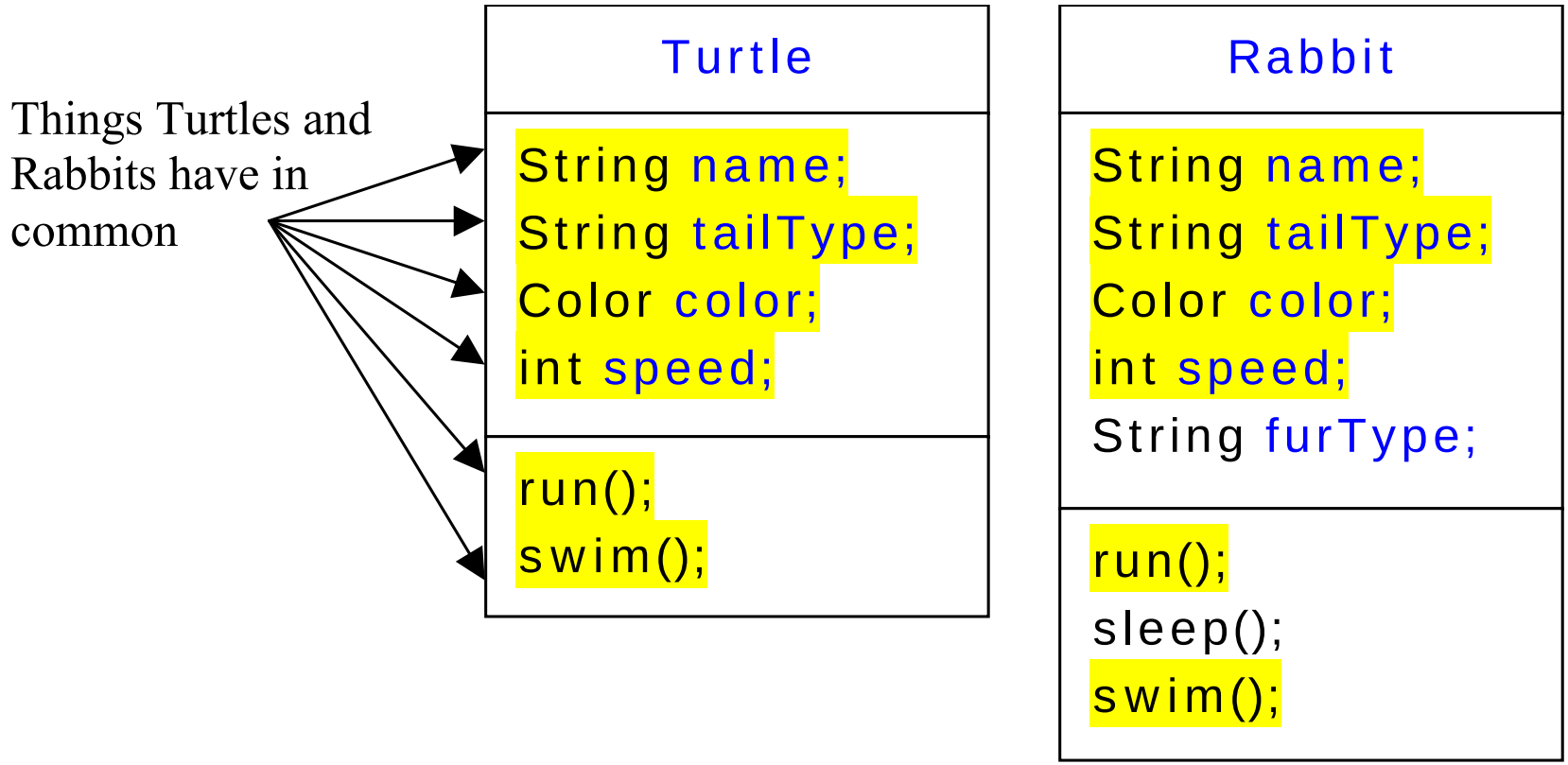
Turtle
String name; String tailType; Color color; int speed;
run(); swim();



Rabbit
String name; String tailType; Color color; int speed; String furType;

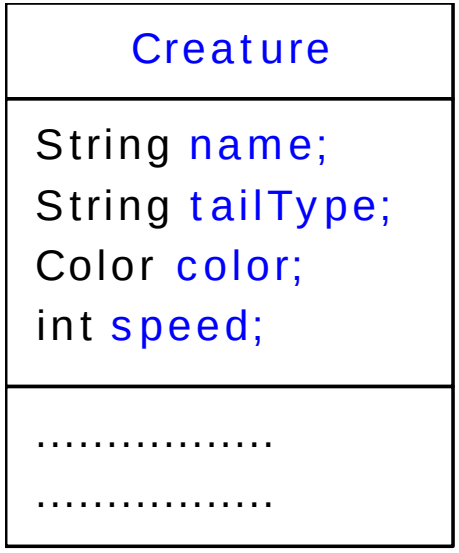
run();
sleep();
swim();

Another look at the `Rabbit` and `Turtle` classes …



Creature.java

- For our type of creatures, we could have a parent class **Creature**, that defines all the base (or generic) functionality of the attributes and methods that **Turtle** and **Rabbit** have in common.




```
import java.awt.*;
public class Creature {
  protected String name, tailType;
  protected Color color;
  protected int speed;

  public int run(int duration,  boolean  zigzag) {
    System.out.println("I run like a generic creature!");
    if (zigzag) return (int)(speed*duration/2);
    return speed*duration;
   }

  public void swim(int duration) {
    System.out.println("I swim for " + duration +
                       " minutes like a generic creature");
   }

  // accessor/mutator methods
  public void setName(String n) { this.name = n; }
  public String getName() { return this.name;}

  // etc ...
}
```

Queen Mary University of London

New Rabbit.java

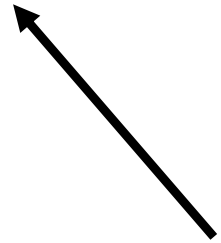
```
import java.awt.*;
public class Rabbit extends Creature {
  protected String furType;

  public void sleep(int duration) {
    System.out.println("I sleep for "+duration+ " minutes like a Rabbit!");
  }

  public int run(int duration, boolean zigzag) {
    System.out.println("I run like a Rabbit!");
    if (zigzag) return (int)(speed*duration*0.75);
    return speed*duration;
  }

  // Specific Rabbit accessor/mutator methods.
  // Can override Creature's methods if necessary.

  public void setFurType(String n) {
    // insert code to do some checking on furtype
    this.furType =  n;
  }
  public String getFurType() { return this.furType; }
}
```

Definitions of new variables and methods.

Redefinition of a method that requires specialised behaviour.

New Turtle.java

```
import java.awt.*;
public class Turtle extends Creature {
   public int run(int duration,  boolean  zigzag) {
      System.out.println("I run like a Turtle!");
      if (zigzag)
        return (int)(speed*duration*0.25);
      return speed*duration;
   }
}
```



Redefinition of a method that requires specialised behaviour.

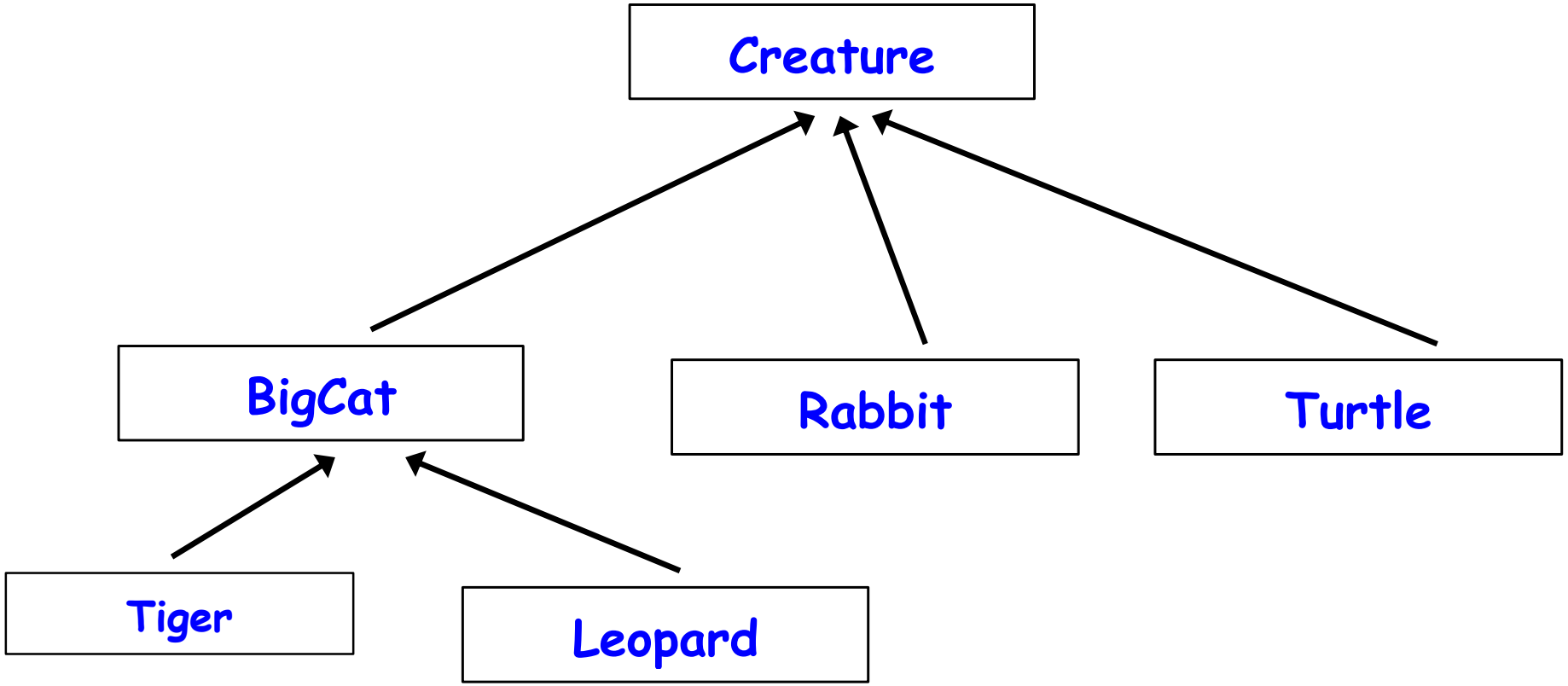
Testing …

```
public class CreatureTest {
  public static void main(String[] args) {
    System.out.println("Rabbit test: ");
    Rabbit r = new Rabbit();
    r.setSpeed(10);
    System.out.println(r.run(5, true));
    r.sleep(8);
    r.swim(2);

    System.out.println("Turtle test: ");
    Turtle t = new Turtle();
    t.setSpeed(5);
    System.out.println(t.run(4, true));
    t.swim(6);
    // Remember: Turtles can't sleep and
    //           do not have a sleep() method.
    //           So you cannot call t.sleep(5);
  }
}
```

```
Rabbit test:
.....................................
.....................................
.....................................
.....................................

Turtle test:
.....................................
.....................................
.....................................
.....................................
```

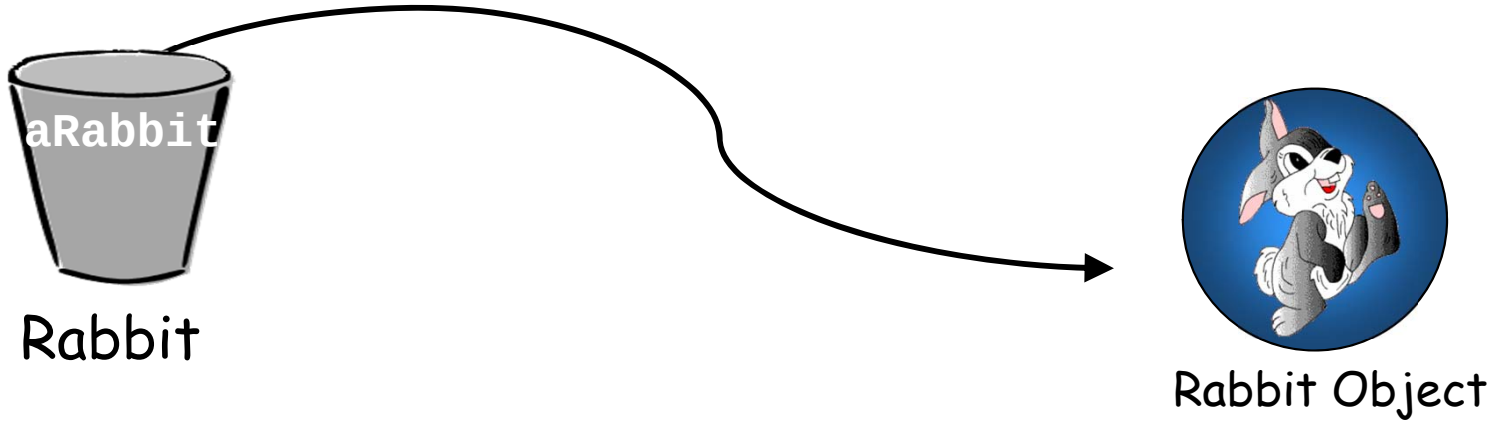
… and things for you to try out!

Other creatures …

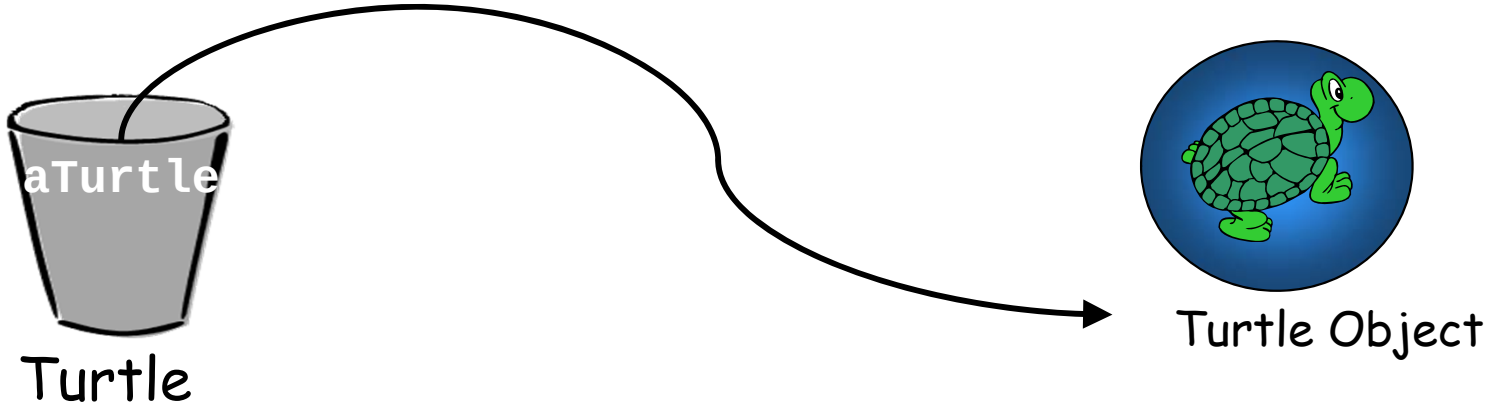


We could extend our design to more creatures …

Instantiating objects (Revision)

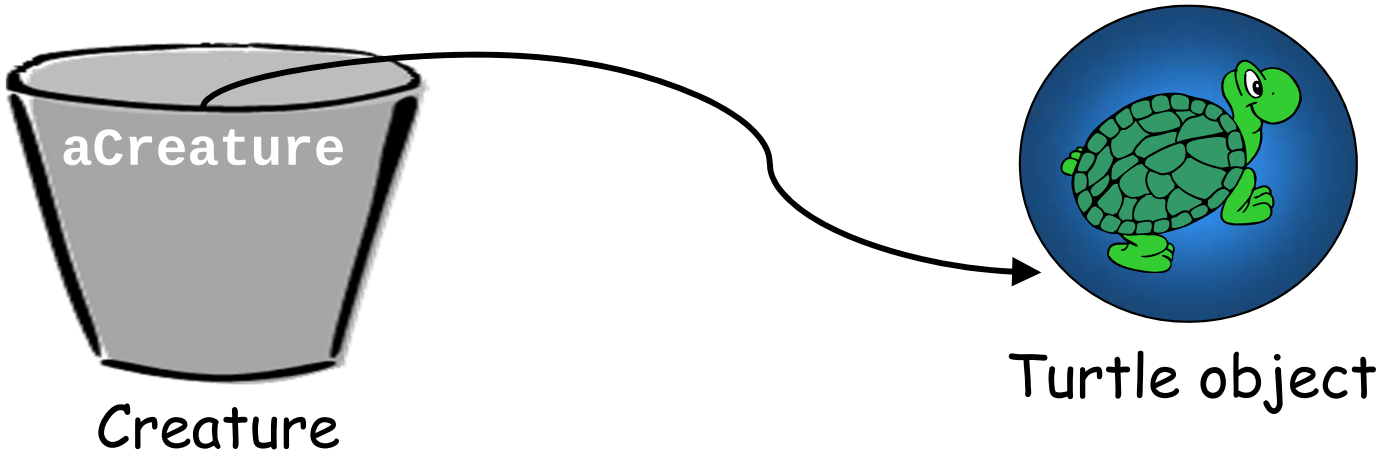



```
Rabbit aRabbit = new Rabbit();
```




```
Turtle aTurtle = new Turtle();
```

Different but valid types …




```
Creature aCreature = new Turtle();
```

This is what it means to treat a subclass (**`Turtle`**) as an instance of the superclass (**`Creature`**).

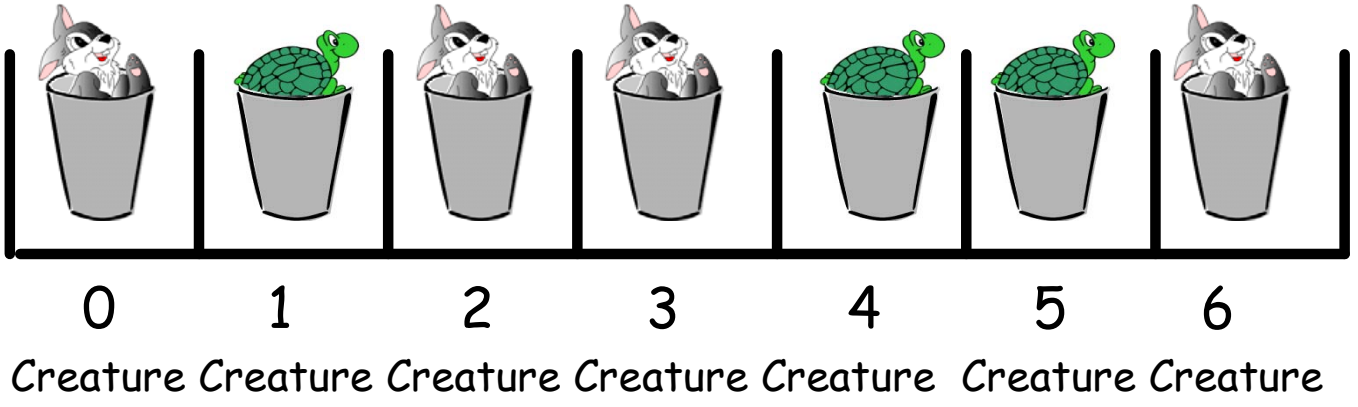
A turtle is-a creature (and thus, can be treated as one).
Important: This is not necessarily true the other way around; you do not know that a **`Creature`** is-a **`Turtle`** (it could be, e.g. a **`Rabbit`**).

Polymorphism

- Polymorphism → Using a single definition (superclass) with different types (subclass).
  - ✓ This is what allows us to treat a **`Turtle`** or a **`Rabbit`** as a **`Creature`**.

```
Creature c = new Rabbit(); // Also allowed!
```

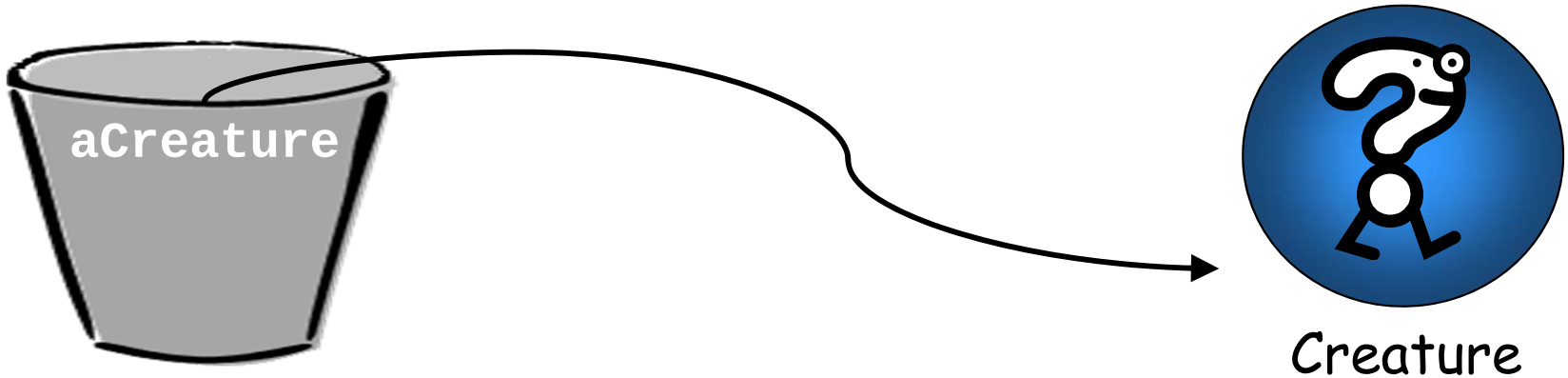
  - ✓ We can now create arrays that contain both **`Rabbit`** and **`Turtle`** objects!



More about this later.

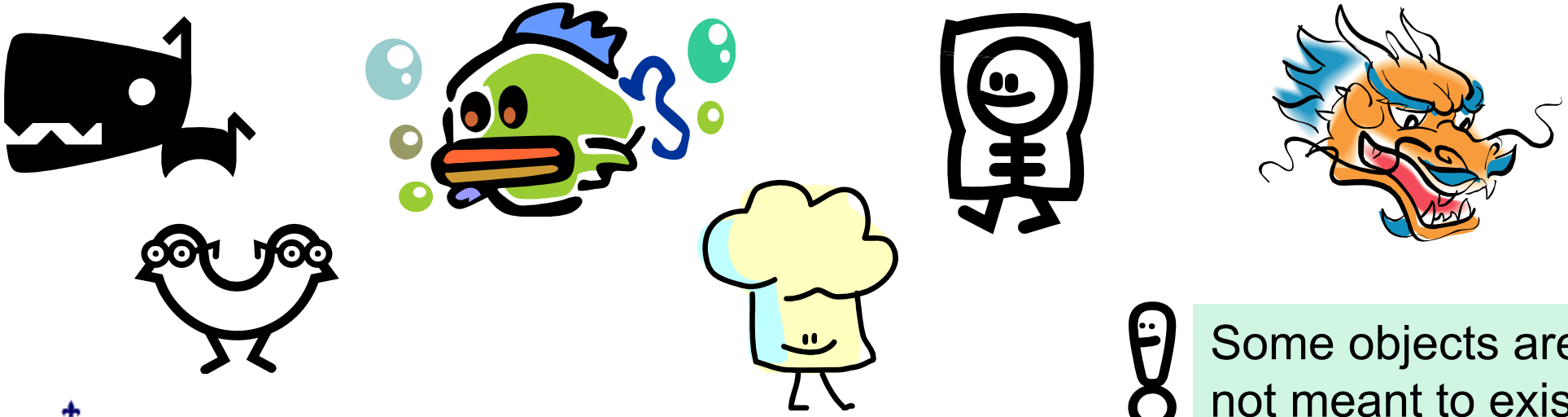
```
Creature cArray[] = new Creature[7];
```

Creatures?




```
Creature aCreature = new Creature();
```

- What does a **`Creature`** object look like? How should it look like?



Some objects are not meant to exist!

Abstract classes

- At the moment we can create a **`Creature()`** object.
- However, it would be nice to have a way to prevent the creation of a template object.

`abstract`

```
import java.awt.*;

public abstract class Creature {
  protected String name, tailType;
  protected Color color;
  protected int speed;

  public int run(int duration, boolean zigzag) {
    // stuff
    return 1;
  }
  public void swim(int duration) {
    // stuff
  }
}
```

What does it mean to be abstract?

- The compiler will not let you instantiate an **abstract** class.
  - Nobody can EVER make an instance of it.
  - The only use it has is in being extended.

```
public class MakeCreatures {
  public void go() {
    Creature aCreature = new Creature();

    Creature bCreature = new Rabbit();
    bCreature.run(5, true);
  }
}
```

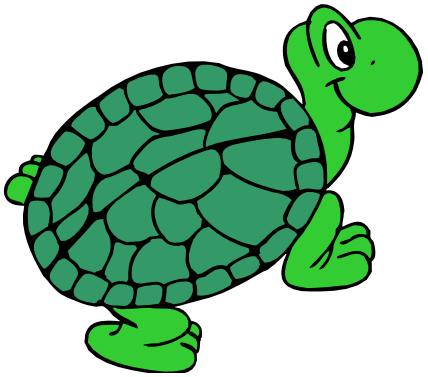
The compiler will not even let you do this!

… and things for you to try out!

Abstract *versus* Concrete

- A non-abstract class is called a concrete class.
- Lots of classes in the API are **abstract**!

- Abstract classes can do more than just prevent one class from becoming an object instance.
  - Example:
    - Do turtles and rabbits run in the same way?
      - ✓ Sometimes, it does not make sense to have a default implementation.
      - ✓ However, we would still like to specify that all creatures have a particular behaviour.



Abstract methods + Example (1/2)

- **`abstract`** in terms of classes ⇒ that class must be extended, in order to be instantiated
- **`abstract`** for methods ⇒ the method must be overridden in the child class
- Example:

```
import java.awt.*;

public abstract class Creature {
  protected String name, tailType;
  protected Color color;
  protected int speed;

  public abstract int run(int duration, boolean zigzag);

  public void swim(int duration) {
    // code to do stuff
   }
}
```

Example (2/2)

```
import java.awt.*;
public class Turtle extends Creature {
}
```

This will not work now!
`Turtle` must implement **`run()`** or be declared **`abstract`**.
In fact, a subclass must implement ALL abstract methods from its superclass (or be declared **`abstract`**).

```
import java.awt.*;
public class Turtle extends Creature {
  public int run(int duration, boolean zigzag) {
    System.out.println("I run like a Turtle!");
    if (zigzag) return (int)(speed*duration*0.25);
    return speed*duration;
  }
}
```

This will now work! The **`Rabbit`** class must do the same.

… and things for you to try out!

Polymorphism in action

```
public class RabbitList {
  private Rabbit[] rabbits = new Rabbit[5];
  private int nextIndex = 0;

  public void addToList(Rabbit r) {
    if (nextIndex < rabbits.length) {
      rabbits[nextIndex] = r;
      System.out.println("Rabbit added at " + nextIndex);
      nextIndex++;
    }
  }
}
```

Only holds **`Rabbit`** objects!

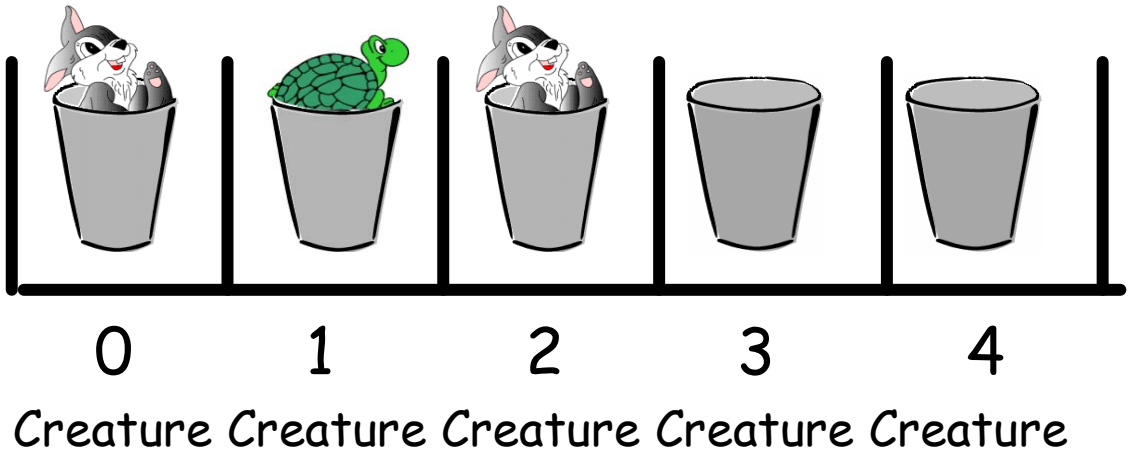
```
public class CreatureList {
  private Creature[] creatures = new Creature[5];
  private int nextIndex = 0;

  public void addToList(Creature r) {
    if (nextIndex < creatures.length) {
      creatures[nextIndex] = r;
      System.out.println("Creature added at " + nextIndex + r);
      nextIndex++;
    }
  }
}
```

Holds any kind of **`Creature`**, even those that we haven't coded yet!*

* After we code them up, of course!

Testing (version 1)

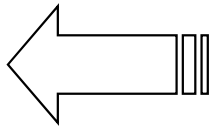
```
public class CreatureTest {
  public static void main(String[] args) {
    CreatureList list = new CreatureList();
    Rabbit r = new Rabbit();
    Turtle t = new Turtle();
    list.addToList(r);
    list.addToList(t);
    list.addToList(new Rabbit());
  }
}
```



Testing (version 2 – using an `ArrayList`)

- Better just to use an **`ArrayList`**!

```
import java.util.*;
public class CreatureTest {
  public static void main(String[] args) {
    ArrayList<Creature> clist = new ArrayList<Creature>();
    Rabbit r = new Rabbit();
    Turtle t = new Turtle();
    clist.add(r);
    clist.add(t);
    clist.add(new Rabbit());
    for (Creature c : clist)
      c.run(5, true);
  }
}
```



We can run all our creatures without knowing what kind of **`Creature`** (i.e. **`Rabbit`** or **`Turtle`**) they are!
This is because **`run()`** is defined in **`Creature`**.

Another example: using Doctors

```
import java.util.*;
public class DoctorTest {
  public static void main(String[] args) {
    ArrayList<Doctor> dlist = new ArrayList<Doctor>();
    dlist.add(new Doctor());
    dlist.add(new Surgeon());
    dlist.add(new FamilyDoctor());
    dlist.add(new Surgeon());

    for (Doctor d : dlist) {
      d.treatPatient();
      d.giveAdvice();
    }
  }
}
```

✓ ✓ ✗

Allowed because **`treatPatient()`** is defined in **`Doctor`**; all subclasses will have this method.

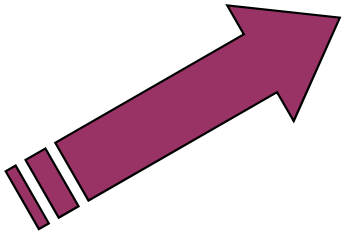
✗ NOT allowed because **`giveAdvice()`** is not defined in **`Doctor`** (only in **`FamilyDoctor`**). The **`ArrayList`** is of type **`Doctor`**, so it can only do things that all Doctors can do – Surgeons can not give advice.

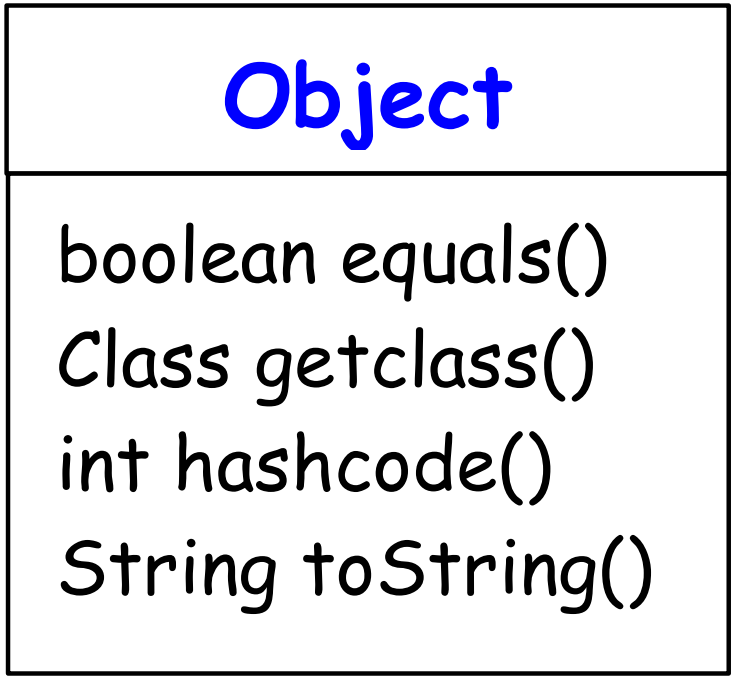
… and things for you to try out!

The “mega” class

- **`java.lang.Object`** is the ultimate parent of **EVERY** class in java.
  - It is implicitly inherited by every class.

All of these methods will work for every class written in Java. However, they may not work as you expect …





Methods of the “mega” class (1/2)

- **`equals()`** determines when one object is equal to another
  - Example

```
Turtle t = new Turtle();
Rabbit r = new Rabbit();
System.out.println((t == r ? "True": "false"));
```

Outputs: `false`

- **`toString()`** allows objects to be printed
  - Example

```
Turtle t = new Turtle();
System.out.println(t);
```

Outputs: `Turtle@7r234f`

Remember: If you don't override the inherited method from the **`Object`** class, you simply get the object's address.

Methods of the "mega" class (2/2)

- **`hashCode()`** is a unique ID for every object, usually based on its memory address
  - Example

```
Rabbit r = new Rabbit();
System.out.println(r.hashCode());
```

    Outputs: `8348748`

- **`getClass()`** returns the class of the object
  - Example

```
Rabbit r = new Rabbit();
System.out.println(r.getClass());
```

    Outputs: `class Rabbit`

Comparing Objects

- Comparing objects is a bit trickier than comparing **`int`**s or **`char`**s.
  Consider:

```
Rabbit r1 = new Rabbit();
Rabbit r2 = new Rabbit();
if (r1 == r2) {
  System.out.println("yes");
}
else {
  System.out.println("no");
}
```

The result will be **no**.
Why?

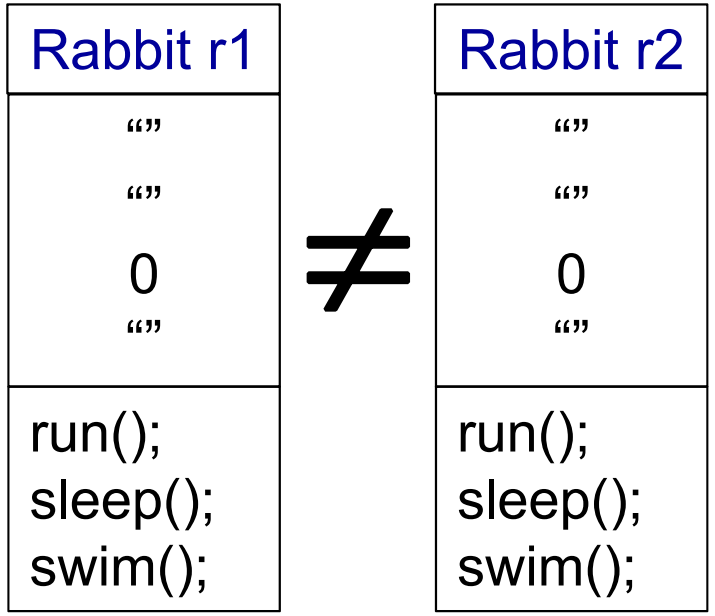
- Because although the two objects have the same state, they have different identities. (You can think of the identity of an object as its address in memory.)
- The **==** operator compares identities for objects.

Using `equals()` to compare objects (1/2)

- We usually implement an **`equals`** method for our classes. Though this may not always be appropriate …

```
public boolean equals(Object o) {
  Rabbit r = (Rabbit) o;
  if ((r.getName().equals(this.getName())) &&
     (r.getFurType().equals(this.getFurType())) &&
     (r.getTailType().equals(this.getTailType())) &&
     (r.getSpeed() == this.getSpeed() &&
     (r.getColor() == this.getColor()) {
    return true;
  }
  else {
    return false;
  } // end if
}
```

Instances of the object with the same attribute values.



Using `equals()` to compare objects (2/2)

- Now the following...

```
Rabbit r1 = new Rabbit();
Rabbit r2 = new Rabbit();
if (r1.equals(r2)) {
  System.out.println("yes");
}
else {
  System.out.println("no");
}
```

will produce **yes**.

Exactly when two objects are considered equal will depend on your application!

Objects everywhere ...

- Why not make everything (e.g. arrays or other collection classes like **`ArrayList`**), an **`Object`** (i.e. **`Object[] obArray`**)?
  1. If we did, we would loose abilities.
  2. If the **`Creature`** array was an **`Object`** array, we would not be able to loop through it and treat it like a **`Creature`**.
     - Objects do not **`run()`**
     - Objects do not **`sleep()`**

Superclasses

- We learnt that Inheritance == Superclasses, so why do it?
  - Can be treated as object of superclass. [Reverse is not true!]
  - Suppose many classes inherit from one superclass
    - Can make an array of superclass references.
    - Treat all objects like superclass objects.
  - Explicit cast
    - Convert the superclass reference to a subclass reference (downcasting).
    - Can only be done when superclass reference is actually referring to a subclass object.

Overriding methods

- Subclass can redefine a superclass method
  - When the method is mentioned in the subclass, then the subclass method version is used.
  - Can access the original superclass method with **`super.methodName`**

- To invoke the superclass constructor explicitly (which is called implicitly by default), use

```
super(); // can pass arguments if needed
```

  - If called explicitly, it must be the first statement.

Examples: method overriding

```
public class Superclass {
  public void printMethod() {
    System.out.println("Printed in Superclass.");
  }
}
```

```
public class Subclass extends Superclass {
  // overrides printMethod() in Superclass
  public void printMethod() {
    super.printMethod();
    System.out.println("Printed in Subclass");
  }
  public static void main(String[] args) {
    Subclass s = new Subclass();
    s.printMethod();
  }
}
```

… and things for you to try out!