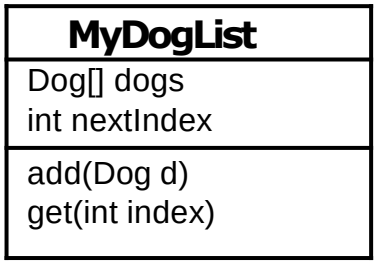
Chapter 8.

The Ultimate Superclass: Object

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

Polymorphism in Action

- Ocean was asked to build a simple Dog-specific list (pretending that we do not have the ArrayList class for the moment)
- He came up with the following design
  - Use a regular array with a fixed length to store the Dog objects
  - Use an `int` value to store the index of the next available position in the Dog array
  - Implement an `add()` method for adding a Dog object to the array at the next available position. When the array is full, the `add()` method will simply do nothing
  - Implement a `get()` method for retrieving a Dog object from the array. If the index argument is out of bound, simply return `null`



Polymorphism in Action

- Below is Ocean's first version of the Dog-specific list

```
public class MyDogList {
   private Dog[] dogs = new Dog[5];
   private int nextIndex = 0;

   public void add(Dog  d) {
      if (nextIndex < dogs.length) {
         dogs[nextIndex] = d;
         System.out.println("Dog added at " + nextIndex);
         nextIndex++;
      }
   }
   public Dog get(int index) {
      if (index >= 0 && index < dogs.length) {
         return dogs[index];
      } else {
         return null;
      }
   }
}
```

Polymorphism in Action

- Came as no surprise, there was a minor change to the specification that the list should keep Cats too
- Ocean had a few options here:
  1. Make a separate class, MyCatList, to store Cat objects
  2. Make a single class, MyDogAndCatList, that keeps 2 different arrays as instance variables, and has 2 different `add()` methods (i.e., `addDog()` and `addCat()`) and 2 different `get()` methods (i.e., `getDog()` and `getCat()`)
  3. Make a heterogeneous MyAnimalList class that can store any kind of Animal subclasses
- The first 2 options are quite clunky, while the 3rd option sounds the best (more generic)

Polymorphism in Action

- Below is Ocean's revised version of the Dog-specific list

```
public class MyAnimalList {
   private Animal[] animals = new Animal[5];
   private int nextIndex = 0;

   public void add(Animal a) {
      if (nextIndex < animals.length) {
         animals[nextIndex] = a;
         System.out.println("Animal added at " + nextIndex);
         nextIndex++;
      }
   }
   public Animal get(int index) {
      if (index >= 0 && index < animals.length) {
         return animals[index];
      } else {
         return null;
      }
   }
}
```

Don't panic. We are not making a new Animal object from the abstract Animal class, but a new array object of type Animal!

Polymorphism in Action

Example

```
public class AnimalTestDrive {
   public static void main(String[] args) {
      MyAnimalList list = new MyAnimalList();
      list.add(new Dog());
      list.add(new Cat());
   }
}
```

Sample output

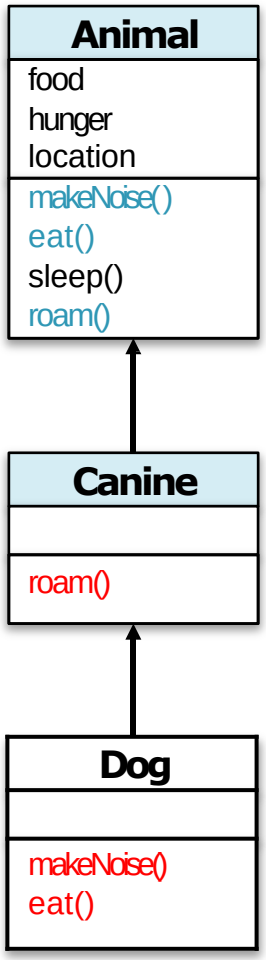
```
Animal added at 0
Animal added at 1
```

What about non-Animals?
Why not make the class generic enough to store anything?

To achieve this, we need a class above Animal, one that is the superclass of everything!

Polymorphism in Action

- Every class in Java extends the Object class
- Object class is the mother of all classes (i.e., it is the superclass of everything!)
- Any class that does not explicitly extend another class, implicitly extends the Object class
- Example
  - Dog extends Canine, and Canine extends Animal
  - Since Animal does not explicitly extend another class, it implicitly extends Object
  - Hence Dog extends Object (indirectly)



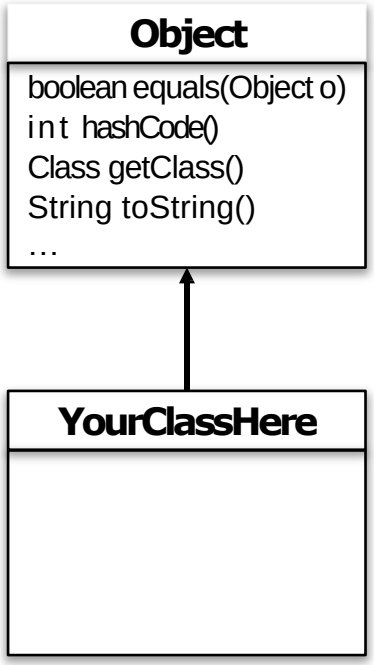
Polymorphism in Action

- With the Object class being the superclass of everything, it is possible to create a list that can store anything!

ArrayList
add(Object elem) Adds the object parameter to the list remove(int index) Removes the object at the index parameter remove(Object elem) Removes this object (if it is in the ArrayList) contains(Object elem) Returns true if there is a match for the object parameter isEmpty() Returns true if the list has no element indexOf(Object elem) Returns the index of the object parameter or –1 size() Returns the number of elements currently in the list get(int index) Returns the object currently at the index parameter …

The Ultimate Superclass: Object

- Every class you write inherits all the methods of the Object class
- The classes you have written inherit methods you do not even know you have
- Below are some of the methods of the Object class that you may be interested in
  - equals()
  - hashCode()
  - getClass()
  - toString()



The Ultimate Superclass: Object

```
public boolean equals(Object obj)
```

- The equals() method
  - Compares the object parameter with the current object, and returns true if they are the same
  - Implements the most discriminating possible equivalence relation on objects, i.e., for any non-null reference values x and y, this method returns true if and only if both x and y are referencing the same object
  - You should override this method if you would like to test whether 2 objects are equal in the sense that they contain the same information (note that you should also override the hashCode() method as well in this case)

The Ultimate Superclass: Object

Example

```
Dog d = new Dog();
Cat c = new Cat();

if (d.equals(c)) {
   System.out.println("true");
} else {
   System.out.println("false");
}
```

Sample output

```
false
```

The Ultimate Superclass: Object

```
public int hashCode()
```

- The hashCode() method
  - Returns a hash code value (integer) for the current object (e.g., to be used with hash tables)
  - By definition, if 2 objects are equal according to the equals() method, then calling the hashCode() method on each of the 2 objects must produce the same integer result
- Example:

```
Dog d = new Dog();
System.out.println(d.hashCode());
```

- Sample output:

```
2018699554
```

Typically, the Object class implements this method by converting the internal address of the object into an integer

The Ultimate Superclass: Object

```
public final Class getClass()
```

- The getClass() method
  - Returns a Class object that represents the runtime class of the current object
  - The keyword final means this method cannot be overridden
- Example:

```
Dog d = new Dog();
System.out.println(d.getClass());
```

- Sample output:

```
class Dog
```

The Ultimate Superclass: Object

```
public String toString()
```

- The `toString()` method
  - Returns a string that "textually represents" the current object
  - Recommended that all subclasses override this method
- Example:

```
Dog d = new Dog();
System.out.println(d.toString());
```

- Sample output

```
Dog@7852e922
```

In the Object class implementation, this string is composed of the class name, the at-sign character @ and the unsigned hexadecimal representation of the hash code of the current object

Using Object as a Polymorphic Type

If polymorphic types are so good, why don't we just make all our methods take and return Object references?

- That defeats the whole point of 'type-safety', one of Java's greatest protection mechanisms for your code
- Recall that Java is a strongly-typed language
- The compiler checks to make sure that the object on which a method is being called is actually capable of responding
- In other words, one can call a method on an object reference only if the class of the reference type actually has that particular method

Using Object as a Polymorphic Type

Examples

```
FamilyDoctor d = new FamilyDoctor();
d.treatPatient();
d.giveAdvice();
```

d is a FamilyDoctor reference and FamilyDoctor has a treatPatient() method and a giveAdvice() method

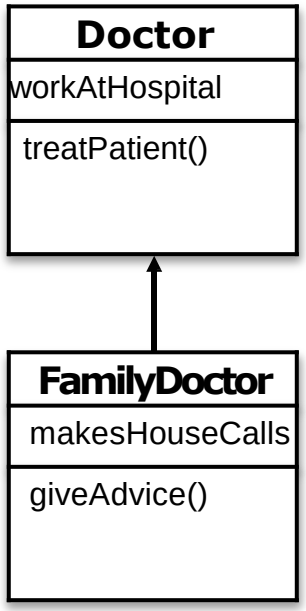
```
Doctor d = new FamilyDoctor();
d.treatPatient();
```

d is a Doctor reference and Doctor has a treatPatient() method

```
Doctor d = new FamilyDoctor();
d.giveAdvice();
```

This won't compile!

d is a Doctor reference but Doctor does not have a giveAdvice() method



Using Object as a Polymorphic Type

- Using Object as a polymorphic type has a price to pay
- Consider an ArrayList declared to hold Dog objects, i.e., ArrayList<Dog>
- When you put an object into an ArrayList<Dog>, it goes in as a Dog, and comes out as a Dog
- Example:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

The get() method returns a Dog reference

Using Object as a Polymorphic Type

- Now consider an ArrayList declared to hold Object objects, i.e., ArrayList<Object>
- The ArrayList will literally take any kind of objects
- Everything comes out of an ArrayList<Object> as an Object reference, regardless of what the actual object is or what the reference type was when you added the object to the list!
- Example:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

This won't compile!

The compiler cannot assume the object that comes out of an ArrayList<Object> is of any type other than Object

Using Object as a Polymorphic Type

- The problem with having everything treated polymorphically as an Object is that the objects appear to lose their true essence
- Example: When a Dog won't act like a Dog

```
public class BadDog {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Dog sameDog = getObject(dog);
      sameDog.makeNoise();
   }
   public static Object getObject(Object o) {
      return o;
   }
}
```

This won't compile!

getObject() returns an Object reference which cannot be assigned to a Dog reference variable

Using Object as a Polymorphic Type

- Example: When a Dog won't act like a Dog

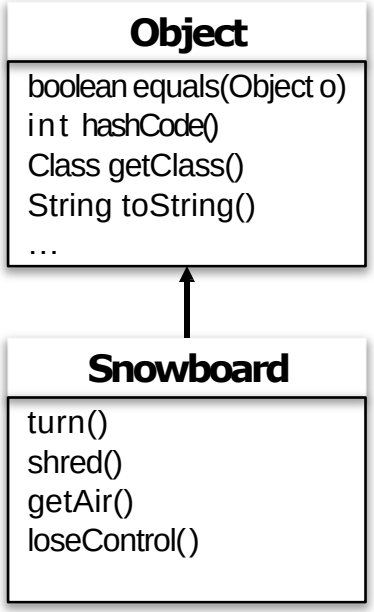
```
public class BadDog2 {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Object sameDog = getObject(dog);
      sameDog.makeNoise();
   }
   public static Object getObject(Object o) {
      return o;
   }
}
```

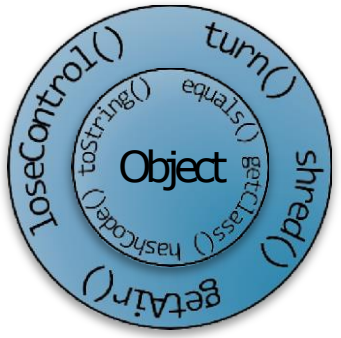
This won't compile!

The compiler decides whether you can call a method based on the reference type, not the actual object type. Note that the Object class does not have a makeNoise() method!

Objects are Object

- An object contains everything it inherits from each of its superclasses
- Every object, regardless of its actual class type, is also an instance of the Object class
- Any object in Java can be treated not just as an instance of its own class, but also an Object
- When you create an object, say a Snowboard, you get a single object on the heap, but this object wraps itself around an inner core representing the Object portion of itself

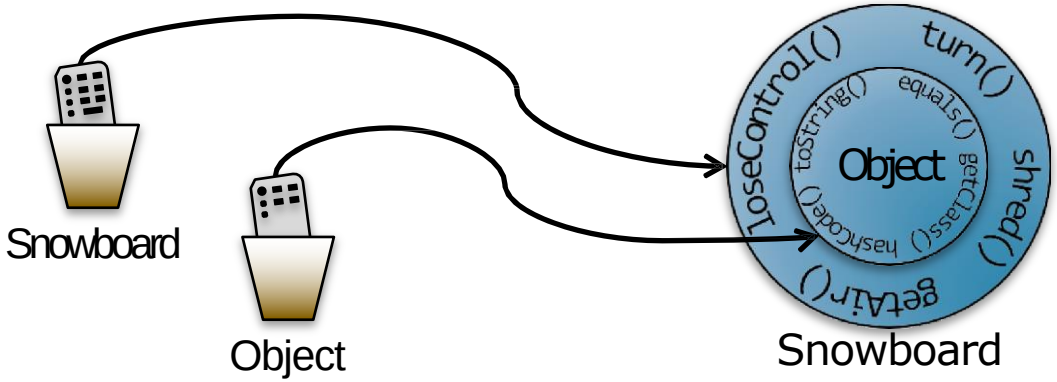




Snowboard

Objects are Object

- If a reference is like a remote control, the remote control takes more and more buttons as you move down the inheritance tree
  - A remote control (reference) of type Object has only a few buttons (for the exposed methods of the Object class)
  - A remote control of type Snowboard includes all the buttons from the Object class, plus any new buttons of the Snowboard class
- An Object reference to an object, say a Snowboard, can see only the Object portion of the object, and access only the instance variables and methods of the Object class



Casting

- A cast can be used to assign an object reference of one type to a reference variable of a subtype, e.g.,

```
Dog dog = new Dog();
Object o = dog;
Dog sameDog = (Dog) o;
```

- At runtime, a cast will fail if the object on the heap is not of a type compatible with the cast!
- To play safe, use the instanceof operator to check if an object is an instance of a certain class before type casting, e.g.,

```
Object o = new Dog();
if (o instanceof Dog) {
   System.out.println("It is a Dog");
}
```

Casting

- Example: When a Dog becomes a Dog again

```
public class BadDog3 {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Object o = getObject(dog);
      if (o instanceof Dog) {
         Dog sameDog = (Dog) o;
         sameDog.makeNoise();
      }
   }
   public static Object getObject(Object o) {
      return o;
   }
}
```

Casting an Object reference to a Dog object back to a Dog reference

Chapter 8.

End

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong