

# Inheritance (Part 3)

Abstract Classes

# Abstract Classes

---

- ▶ sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - ▶ e.g. you might want to know what a **Dog**'s bark sounds like but the sound of the bark depends on the breed of the dog
    - ▶ you want to add the method bark to **Dog** but only the subclasses of **Dog** can implement **bark**

# Abstract Classes

---

- ▶ sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - ▶ e.g. you might want to know the breed of a **Dog** but only the subclasses have information about the breed
    - ▶ you want to add the method **getBreed** to **Dog** but only the subclasses of **Dog** can implement **getBreed**

- 
- ▶ if the base class has methods that only subclasses can define *and* the base class has fields common to all subclasses then the base class should be abstract
    - ▶ if you have a base class that just has methods that it cannot implement then you probably want an interface
  - ▶ abstract :
    - ▶ (dictionary definition) existing only in the mind
  - ▶ in Java an abstract class is a class that you cannot make instances of
    - ▶ e.g. <http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html>

- 
- ▶ an abstract class provides a partial definition of a class
    - ▶ the subclasses complete the definition
  - ▶ an abstract class can define fields and methods
    - ▶ subclasses *inherit* these
  - ▶ an abstract class can define constructors
    - ▶ subclasses *must call* these
  - ▶ an abstract class can declare abstract methods
    - ▶ subclasses *must define* these (unless the subclass is also abstract)

# Abstract Methods

---

- ▶ an abstract base class can declare, *but not define*, zero or more abstract methods



```
public abstract class Dog
{
    // fields, ctors, regular methods

    public abstract String getBreed();
}
```



- ▶ the base class is saying "all **Dogs** can provide a **String** describing the breed, but only the subclasses know enough to implement the method"

# Abstract Methods

---

- ▶ the non-abstract subclasses must provide definitions for all abstract methods
  - ▶ consider `getBreed` in `Mix`

---

```
public class Mix extends Dog
{ // stuff from before...

    @Override public String getBreed() {
        if(this.breeds.isEmpty()) {
            return "mix of unknown breeds";
        }
        StringBuffer b = new StringBuffer();
        b.append("mix of");
        for(String breed : this.breeds) {
            b.append(" " + breed);
        }
        return b.toString();
    }
}
```



# PureBreed

---

- ▶ a purebreed dog is a dog with a single breed
  - ▶ one **String** field to store the breed
- ▶ note that the breed is determined by the subclasses
  - ▶ the class **PureBreed** cannot give the **breed** field a value
  - ▶ but it can implement the method **getBreed**
- ▶ the class **PureBreed** defines an field common to all subclasses and it needs the subclass to inform it of the actual breed
  - ▶ **PureBreed** is also an abstract class

---

```
public abstract class PureBreed extends Dog
{
    private String breed;

    public PureBreed(String breed) {
        super();
        this.breed = breed;
    }

    public PureBreed(String breed, int size, int energy) {
        super(size, energy);
        this.breed = breed;
    }
}
```



---

```
@Override public String getBreed()  
{  
    return this.breed;  
}  
  
}
```

# Subclasses of PureBreed

---

- ▶ the subclasses of **PureBreed** are responsible for setting the breed
  - ▶ consider **Komondor**

# Komondor

---

```
public class Komondor extends PureBreed
{
    private final String BREED = "komondor";

    public Komondor() {
        super(BREED);
    }

    public Komondor(int size, int energy) {
        super(BREED, size, energy);
    }

    // other Komondor methods...
}
```

# Inheritance (Part 4)

Static Features; Interfaces

# Static Fields and Inheritance

---

- ▶ static fields behave the same as non-static fields in inheritance
  - ▶ public and protected static fields are inherited by subclasses, and subclasses can access them directly by name
  - ▶ private static fields are not inherited and cannot be accessed directly by name
    - ▶ but they can be accessed/modified using public and protected methods

# Static Fields and Inheritance

---

- ▶ the important thing to remember about static fields and inheritance
  - ▶ *there is only one copy of the static field shared among the declaring class and all subclasses*
- ▶ consider trying to count the number of **Dog** objects created by using a static counter



---

```
// the wrong way to count the number of Dogs created
public abstract class Dog {
    // other fields...
    static protected int numCreated = 0; protected, not private, so that
    subclasses can modify it directly

    Dog() {
        // ...
        Dog.numCreated++;
    }

    public static int getNumberCreated() {
        return Dog.numCreated;
    }

    // other constructors, methods...
}
```

---

```
// the wrong way to count the number of Dogs created
public class Mix extends Dog
{
    // fields...

    Mix()
    {
        super();
        Mix.numCreated++;
    }

    // other constructors, methods...
}
```

---

```
// too many dogs!
```

```
public class TooManyDogs
{
    public static void main(String[] args)
    {
        Mix mutt = new Mix();
        System.out.println( Mix.getNumberCreated() );
    }
}
```

prints 2

# What Went Wrong?

---

- ▶ there is only one copy of the static field shared among the declaring class and all subclasses
  - ▶ **Dog** declared the static field
  - ▶ **Dog** increments the counter everytime its constructor is called
  - ▶ **Mix** inherits and shares the single copy of the field
  - ▶ **Mix** constructor correctly calls the superclass constructor
    - ▶ which causes **numCreated** to be incremented by **Dog**
  - ▶ **Mix** constructor then incorrectly increments the counter

# Counting Dogs and Mixes

---

- ▶ suppose you want to count the number of **Dog** instances and the number of **Mix** instances
  - ▶ **Mix** must also declare a static field to hold the count
    - ▶ somewhat confusingly, **Mix** can give the counter the same name as the counter declared by **Dog**

---

```
public class Mix extends Dog
{
    // other fields...
    private static int numCreated = 0; // bad style

    public Mix()
    {
        super(); // will increment Dog.numCreated
        // other Mix stuff...
        numCreated++; // will increment Mix.numCreated
    }

    // ...
}
```

# Hiding Fields

---

- ▶ note that the **Mix** field **numCreated** has the same name as an field declared in a superclass
  - ▶ whenever **numCreated** is used in **Mix**, it is the **Mix** version of the field that is used
- ▶ if a subclass declares an field with the same name as a superclass field, we say that the subclass field hides the superclass field
  - ▶ considered bad style because it can make code hard to read and understand
    - ▶ should change **numCreated** to **numMixCreated** in **Mix**

# Static Methods and Inheritance

---

- ▶ there is a big difference between calling a static method and calling a non-static method when dealing with inheritance
- ▶ *there is no dynamic dispatch on static methods*
  - ▶ therefore, you cannot override a static method



```
public abstract class Dog {  
    private static int numCreated = 0;  
    public static int getNumCreated() {  
        return Dog.numCreated;  
    }  
}
```

```
public class Mix {  
    private static int numMixCreated = 0;  
    public static int getNumCreated() {  
        return Mix.numMixCreated;  
    }  
}
```

notice no @Override

```
public class Komondor {  
    private static int numKomondorCreated = 0;  
    public static int getNumCreated() {  
        return Komondor.numKomondorCreated;  
    }  
}
```

notice no @Override

```
public class WrongCount {
    public static void main(String[] args) {
        Dog mutt = new Mix();
        Dog shaggy = new Komondor();
        System.out.println( mutt.getNumCreated() );
        System.out.println( shaggy.getNumCreated() );
        System.out.println( Mix.getNumCreated() );
        System.out.println( Komondor.getNumCreated() );
    }
}
```

```
prints 2
       2
       1
       1
```

# What's Going On?

---

- ▶ *there is no dynamic dispatch on static methods*
- ▶ because the declared type of **mutt** is **Dog**, it is the **Dog** version of **getNumCreated** that is called
- ▶ because the declared type of **shaggy** is **Dog**, it is the **Dog** version of **getNumCreated** that is called

# Hiding Methods

---

- ▶ notice that **Mix.getNumCreated** and **Komondor.getNumCreated** work as expected
- ▶ if a subclass declares a static method with the same name as a superclass static method, we say that the subclass static method hides the superclass static method
  - ▶ *you cannot override a static method, you can only hide it*
  - ▶ hiding static methods is considered bad form because it makes code hard to read and understand

- 
- ▶ the client code in **WrongCount** illustrates two cases of bad style, one by the client and one by the implementer of the **Dog** hierarchy
    1. the client should not have used an instance to call a static method
    2. the implementer should not have hidden the static method in **Dog**

# Interfaces

# Interfaces

---

- ▶ recall that you typically use an abstract class when you have a superclass that has fields and methods that are common to all subclasses
  - ▶ the abstract class provides a partial implementation that the subclasses must complete
  - ▶ subclasses can only inherit from a single superclass
- ▶ if you want classes to support a common API then you probably want to define an interface

# Interfaces

---

- ▶ in Java an *interface* is a reference type (similar to a class)
- ▶ an interface says what methods an object must have and what the methods are supposed to do
  - ▶ i.e., an interface is an API



# Interfaces

---

- ▶ an interface can contain *only*
  - ▶ constants
  - ▶ method signatures
  - ▶ nested types (ignore for now)
- ▶ there are no method bodies
- ▶ interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces

# Interfaces Already Seen

---

access—either public or  
package-private (blank)

interface  
name

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

# Interfaces Already Seen

---

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

access—either public or  
package-private (blank)

interface  
name

parent  
interfaces

```
public interface Collection<E> extends Iterable<E>
{
    boolean add(E e);
    void clear();
    boolean contains(Object o);
    // many more method signatures...
}
```

# Interfaces Already Seen

---

```
public interface List<E> extends Collection<E>
{
    boolean add(E e);
    void    add(int index, E element);
    boolean addAll(Collection<? extends E> c);
    // many more method signatures...
}
```

# Creating an Interface

---

- ▶ decide on a name
- ▶ decide what methods you need in the interface
- ▶ this is harder than it sounds because...
  - ▶ once an interface is released and widely implemented, it is almost impossible to change
    - ▶ if you change the interface, all classes implementing the interface must also change

# Function Interface

---

- ▶ in mathematics, a real-valued scalar function of one real scalar variable maps a real value to another real value

$$y = f(x)$$

# Creating an Interface

---

- ▶ decide on a name
  - ▶ `DoubleToDoubleFunction`
  
- ▶ decide what methods you need in the interface
  - ▶ `double at(double x)`
  - ▶ `double[] at(double[] x)`

# Creating an Interface

---

```
public interface DoubleToDoubleFunction {  
    double    at(double x);  
    double[] at(double[] x);  
}
```



# Classes that Implement an Interface

---

- ▶ a class that implements an interface says so by using the **implements** keyword
  - ▶ consider the function  $f(x) = x^2$

---

```
public class Square implements
    DoubleToDoubleFunction {
    public double at(double x) {
        return x * x;
    }
}
```

```
public double[] at(double[] x) {
    double[] result = new double[x.length];
    for (int i = 0; i < x.length; i++) {
        result[i] = x[i] * x[i];
    }
    return result;
}
}
```

# Implementing Multiple Interfaces

---

- ▶ unlike inheritance where a subclass can extend only one superclass, a class can implement as many interfaces as it needs to

```
public class ArrayList<E>  
    extends AbstractList<E>      superclass  
    implements List<E>,  
               RandomAccess,    interfaces  
               Cloneable,  
               Serializable
```