Inheritance, cont.

Notes Chapter 6 and AJ Chapters 7 and 8

Preconditions and Inheritance

- precondition
  - what the method assumes to be true about the arguments passed to it

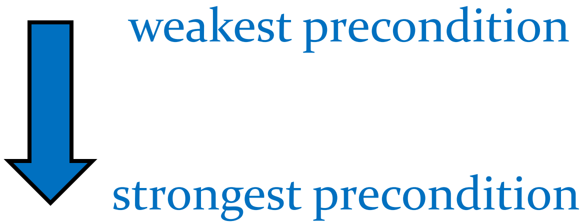
- inheritance (is-a)
  - a subclass is supposed to be able to do everything its superclasses can do

- how do they interact?

Strength of a Precondition

- to strengthen a precondition means to make the precondition more restrictive

```
// Dog setEnergy
// 1. no precondition
// 2. 1 <= energy
// 3. 1 <= energy <= 10
public void setEnergy(int energy)
{ ... }
```


weakest precondition

strongest precondition

Preconditions on Overridden Methods

- a subclass can change a precondition on a method *but it must not strengthen the precondition*
  - a subclass that strengthens a precondition is saying that it cannot do everything its superclass can do

```
// Dog setEnergy
// assume non-final
// @pre. none

public
void setEnergy(int nrg)
{ // ... }
```

```
// Mix setEnergy
// bad : strengthen precond.
// @pre. 1 <= nrg <= 10

public
void setEnergy(int nrg)
{
  if (nrg < 1 || nrg > 10)
  { // throws exception }
  // ...
}
```

- client code written for **Dog**s now fails when given a **Mix**

```
// client code that sets a Dog's energy to zero
public void walk(Dog d)
{
  d.setEnergy(0);
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Postconditions and Inheritance

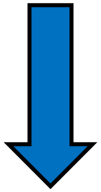
- postcondition
  - what the method promises to be true when it returns
    - the method might promise something about its return value
      - □ "returns size where size is between 1 and 10 inclusive"
    - the method might promise something about the state of the object used to call the method
      - □ "sets the size of the dog to the specified size"
    - the method might promise something about one of its parameters

- how do postconditions and inheritance interact?

Strength of a Postcondition

- to strengthen a postcondition means to make the postcondition more restrictive

```
// Dog getSize
// 1. no postcondition
// 2. 1 <= this.size
// 3. 1 <= this.size <= 10
public int getSize()
{ ... }
```



weakest postcondition

strongest postcondition

Postconditions on Overridden Methods

- a subclass can change a postcondition on a method *but it must not weaken the postcondition*
  - a subclass that weakens a postcondition is saying that it cannot do everything its superclass can do

```
// Dog getSize
//
// @post. 1 <= size <= 10

public
int getSize()
{ // ... }
```

```
// Dogzilla getSize
// bad : weaken postcond.
// @post. 1 <= size

public
int getSize()
{ // ... }
```

Dogzilla: a made-up breed of dog that has no upper limit on its size

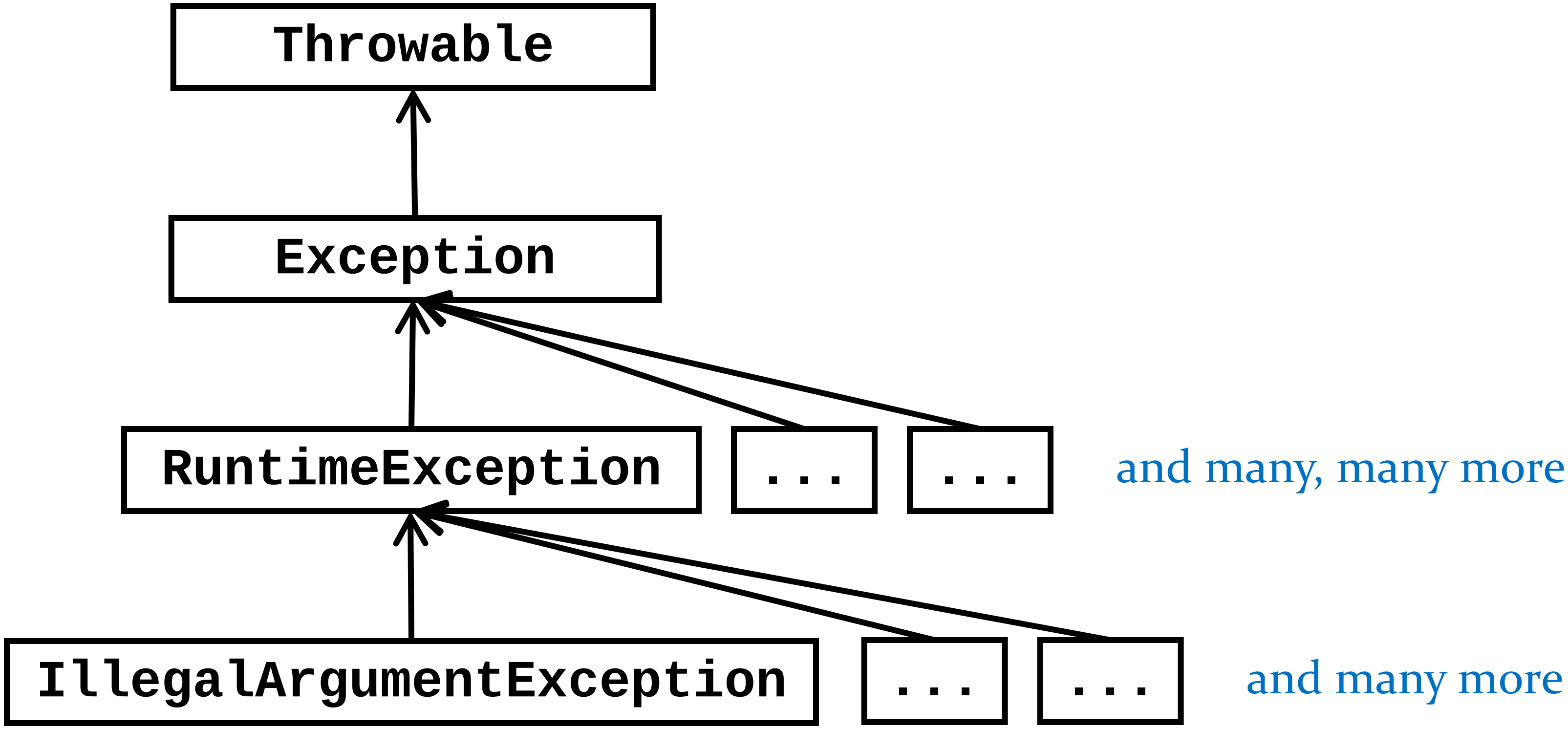
- client code written for **Dog**s can now fail when given a **Dogzilla**

```
// client code that assumes Dog size <= 10
public String sizeToString(Dog d)
{
  int sz = d.getSize();
  String result = "";
  if (sz < 4)         result = "small";
  else if (sz < 7)    result = "medium";
  else if (sz <= 10) result = "large";
  return result;
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Exceptions

- all exceptions are objects that are subclasses of **`java.lang.Throwable`**



Throwable

Exception

RuntimeException

...

...

and many, many more

IllegalArgumentException

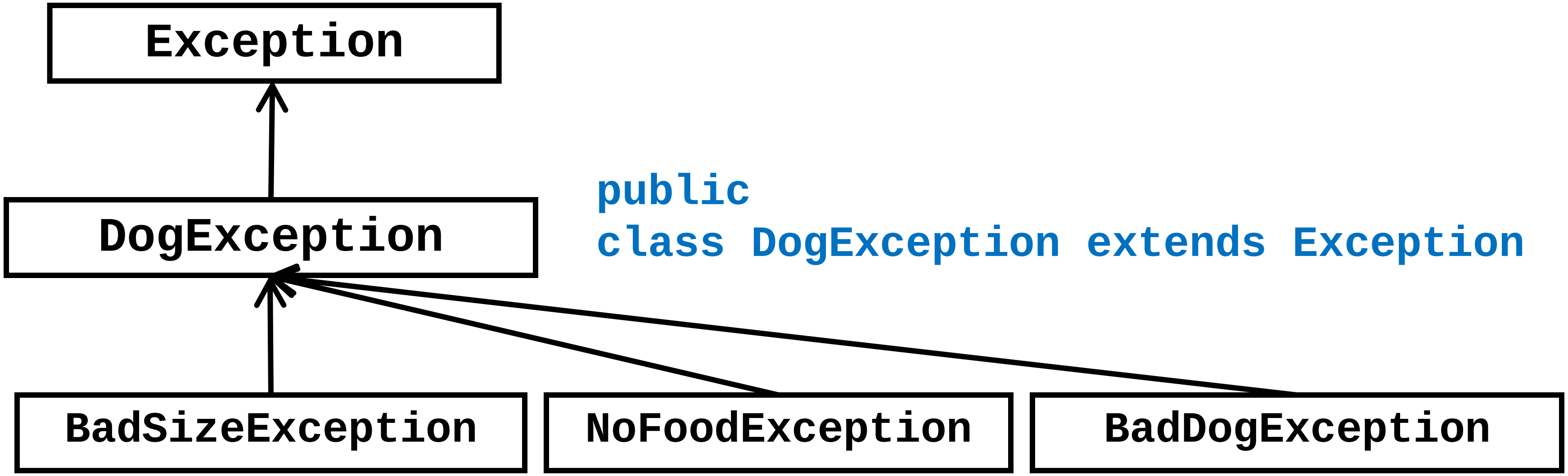
...

...

and many more

User Defined Exceptions

- you can define your own exception hierarchy
  - often, you will subclass Exception



```
public
class DogException extends Exception
```

Exceptions and Inheritance

- a method that claims to throw an exception of type **`X`** is allowed to throw any exception type that is a subclass of **`X`**
  - this makes sense because exceptions are objects and subclass objects are substitutable for ancestor classes

```
// in Dog
public void someDogMethod() throws DogException
{
  // can throw a DogException, BadSizeException,
  //             NoFoodException, or BadDogException
}
```

- if a subclass overrides a method that throws an exception then it must either
  1. throw the same type of exception
  2. throw a subclass of the exception type
  3. not throw an exception

```
// in Dog
public void someDogMethod() throws DogException
{ // ... }

// in Mix; bad, don't do this
@Override
public void someDogMethod() throws DogException,
                                   IOException
{ // ... }
```

```
// client
//  works if given a Dog instance but fails if
//  given a Mix instance that throws an IOException
public void someClientMethod(Dog d)
{
  try {
    d.someDogMethod();
  }
  catch(DogException ex) {
    // deal with the exception
  }
}
```

Which are Legal?

- in Mix

```
@Override
public void someDogMethod() throws BadDogException
```

✓

```
@Override
public void someDogMethod() throws Exception
```

✗

```
@Override
public void someDogMethod()
```

✓

```
@Override
public void someDogMethod()
      throws DogException, IllegalArgumentException
```

✗

Polymorphism

- inheritance allows you to define a base class that has attributes and methods
  - classes derived from the base class can use the public and protected base class attributes and methods
- polymorphism allows the implementer to change the behaviour of the derived class methods

```
// client code
public void print(Dog d) {
  System.out.println( d.toString() );
}
```

```
Dog toString
CockerSpaniel toString
Mix toString
```

```
// later on...
Dog           fido = new Dog();
CockerSpaniel lady = new CockerSpaniel();
Mix           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

```
// client code
public void print(Dog d) {
  System.out.println( d.toString() );
}                      Dog toString
                       CockerSpaniel toString
                       Mix toString
// later on...
Dog           fido = new Dog();
Dog           lady = new CockerSpaniel();
Dog           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

```
// client code
public void print(Object obj) {
  System.out.println( obj.toString() );
}

// later on...
Dog           fido = new Dog();
Dog           lady = new CockerSpaniel();
Dog           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
this.print(new Date());
```

```
Dog toString
CockerSpaniel toString
Mix toString
Date toString
```

Late Binding

- polymorphism requires *late binding* of the method name to the method definition
  - late binding means that the method definition is determined at run-time

non-static method

```
obj.toString()
```

run-time type of
the instance **`obj`**

- the declared type of an instance determines what methods can be used

```
Dog lady = new CockerSpaniel();
```

  - the name **`lady`** can only be used to call methods in **`Dog`**
  - **`lady.someCockerSpanielMethod()`** won't compile

- the actual type of the instance determines what definition is used when the method is called
  - **`lady.toString()`** uses the **`CockerSpaniel`** definition of **`toString`**

Abstract Classes

- often 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`**
  - 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 attributes 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

- an abstract class provides a partial definition of a class
  - the subclasses complete the definition

- an abstract class can define attributes and methods
  - subclasses inherit these
- an abstract class can define constructors
  - subclasses can 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
{
  // attributes, ctors, regular methods

  public abstract String getBreed();
}
```

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

```
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`** attribute to store the breed
- note that the breed is determined by the subclasses
  - the class **`PureBreed`** cannot give the **`breed`** attribute a value
  - but it can implement the method **`getBreed`**
- the class **`PureBreed`** defines an attribute 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;
  }

}
```

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...
}
```

Static Attributes and Inheritance

- static attributes behave the same as non-static attributes in inheritance
  - public and protected static attributes are inherited by subclasses, and subclasses can access them directly by name
  - private static attributes are not inherited and cannot be accessed directly by name
    - but they can be accessed/modified using public and protected methods
- the important thing to remember about static attributes and inheritance
  - there is only one copy of the static attribute shared among the declaring class and all subclasses

```
// the wrong way to count the number of Dogs created
public abstract class Dog {
  // other attributes...
  static protected int numCreated = 0;

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

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

  // other contructors, methods...
}
```

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

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

  // other contructors, 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 attribute shared among the declaring class and all subclasses
  - **`Dog`** declared the static attribute
  - **`Dog`** increments the counter everytime its constructor is called
  - **`Mix`** inherits and shares the single copy of the attribute
  - **`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 attribute 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 attributes...
  private static int numCreated = 0;  // bad style

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

  // ...
```

Hiding Attributes

- note that the **`Mix`** attribute **`numCreated`** has the same name as an attribute declared in a superclass
  - whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the attribute that is used
- if a subclass declares an attribute with the same name as a superclass attribute, we say that the subclass attribute hides the superclass attribute
  - 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*

```
public abstract class Dog {
  // Dog stuff...
  public static int getNumCreated() {
    return Dog.numCreated;
  }
}

public class Mix {
  // Mix stuff...
  public static int getNumCreated() {
    return Mix.numMixCreated;
  }
}
```

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`**

Abstract class vs. Interfaces

- recall that you typically use an abstract class when you have a superclass that has attributes 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

- in Java an *interface* is a reference type (similar to a class)
- 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

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

access—either public or package-private (blank)

interface name

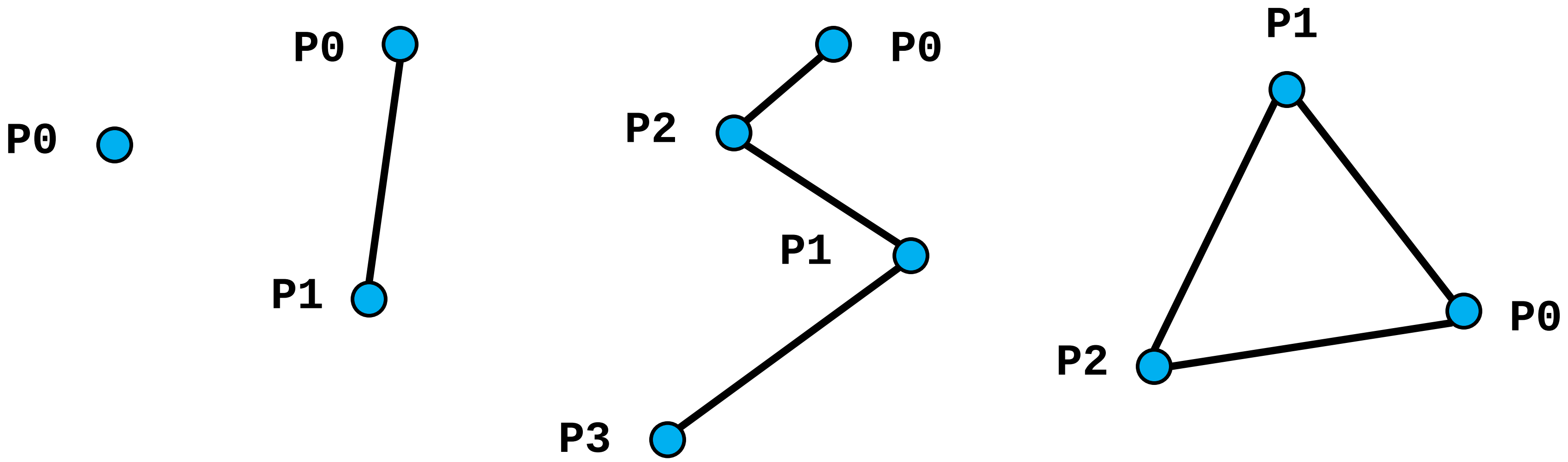
parent interfaces

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

Cell Interface

- i.e. the **`PolygonalModel`** class defined a shape using a collection of **`Triangle`** instances
  - there are many different types of geometric primitives that we might want to represent the shape with
    - point
    - line
    - polyline
    - triangle
    - polygon
    - …

- each primitive can be defined by a list of points and a list of edges connecting the points



point	line	polyline	triangle
P0	P0,P1	P0,P1,P2,P3	P0,P1,P2
	0,1	0,2 2,1 1,3	0,1 1,2 2,0

```
public interface Cell
{
  int         numberOfPoints();
  int         numberOfEdges();
  Vector3d[] getPoints();
  int[]      getEdges();
  // ...
}

public class Point    implements Cell { // ... }
public class Line     implements Cell { // ... }
public class PolyLine implements Cell { // ... }
public class Triangle implements Cell { // ... }
public class PolygonalModel implements Iterable<Cell>
{
  private List<Cell> cells;

  // ...
}

// client somewhere; reads a model from a file
PolygonalModel model = new PolygonalModel("model.stl");
for(Cell c : model) {
  draw(c);
}
```

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>
  implements List<E>,
             RandomAccess,
             Cloneable,
             Serializable
```

superclass

interfaces