Programming and Algorithms II

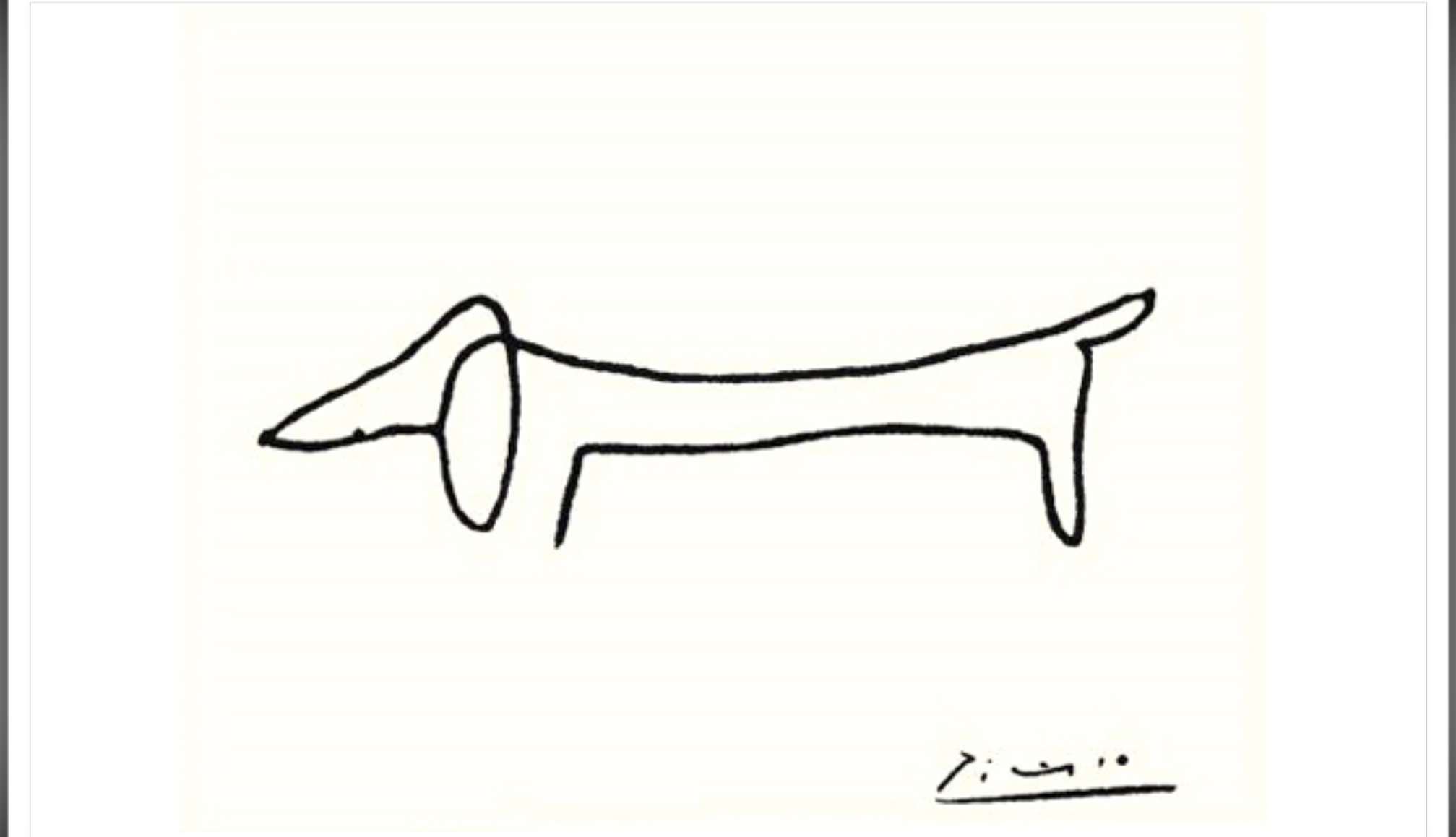
Lecture 5: Abstraction and Encapsulation

Nicolas Wu

nicolas.wu@bristol.ac.uk

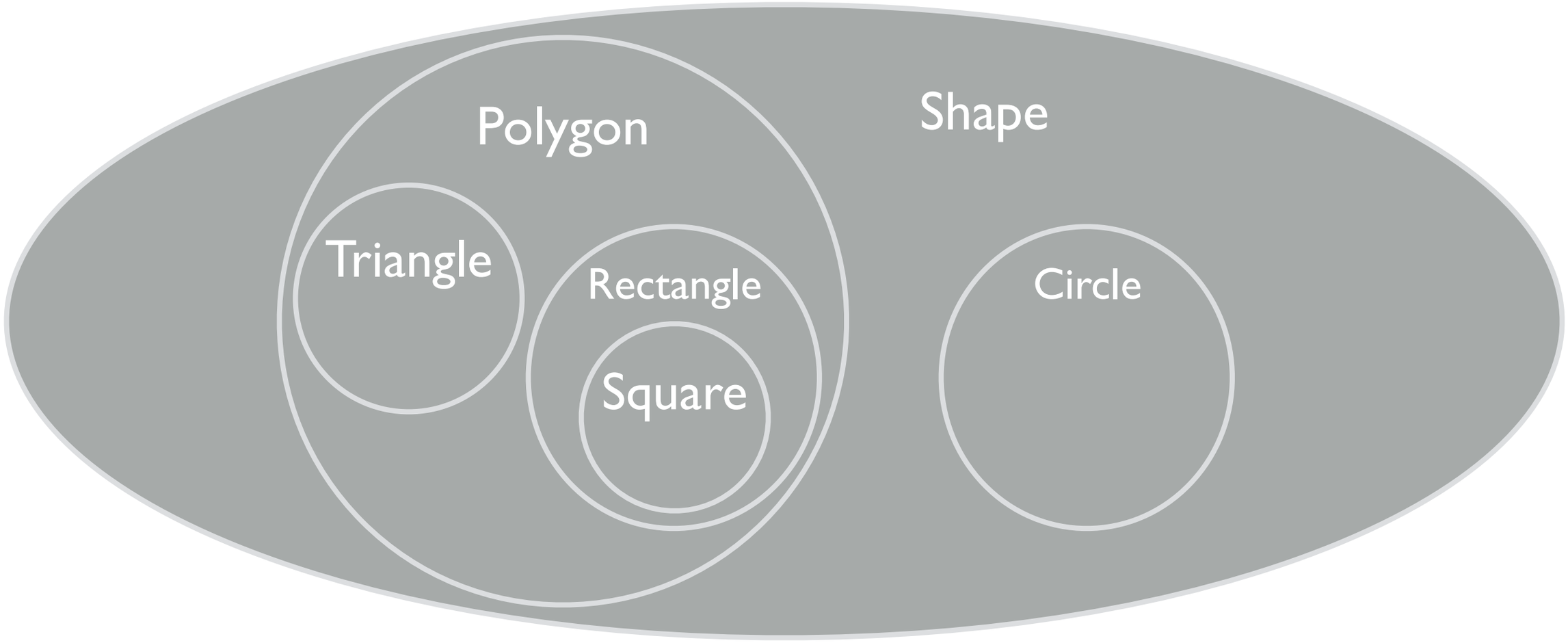
Department of Computer Science
University of Bristol

Abstraction



Abstraction

- Certain classes are too *abstract* to have any sensible direct object



Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

Q. Is this a sensible class?

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

Q. Is this a sensible class?

A. No: we can't have a shape without an area!

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

Q. Is this a sensible class?

A. No: we can't have a shape without an area!

A. Yes: a point is a shape without an area!

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

Q. Is this a sensible class?

A. No: we can't have a shape without an area!

A. Yes: a point is a shape without an area!

A. It depends on what we're trying to model: what's the invariance?

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

- To prevent us from making such an object, we use the *abstract* keyword to the class

```
abstract class Shape {
  Point centre;
  double area() { return 0; };
}
```

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

- To prevent us from making such an object, we use the *abstract* keyword to the class

Q. Is this enough?

```
abstract class Shape {
  Point centre;
  double area() { return 0; };
}
```

Abstraction

- Here's (almost) the shape class we had earlier

```
class Shape {
  Point centre;
  double area() { return 0; };
}
```

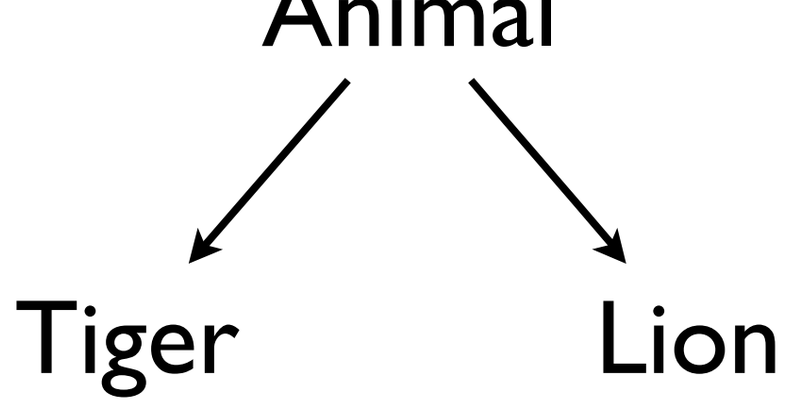
- To prevent us from making such an object, we use the *abstract* keyword to the class *and the method*

```
abstract class Shape {
  Point centre;
  abstract double area();
}
```

Deadly Diamond of Death

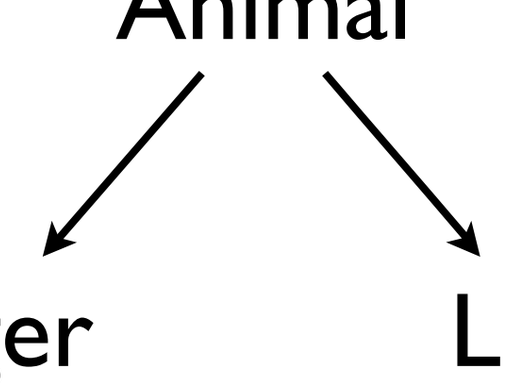
Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance



Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance




```
class Tiger extends Animal {
  void noise() { // GRRR! }
}
```

```
class Lion extends Animal {
  void noise() { // ROAR! }
}
```

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance

Animal

Tiger

Lion

So far so good, no diamonds, but wait, what's this?!

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance

Animal

Tiger

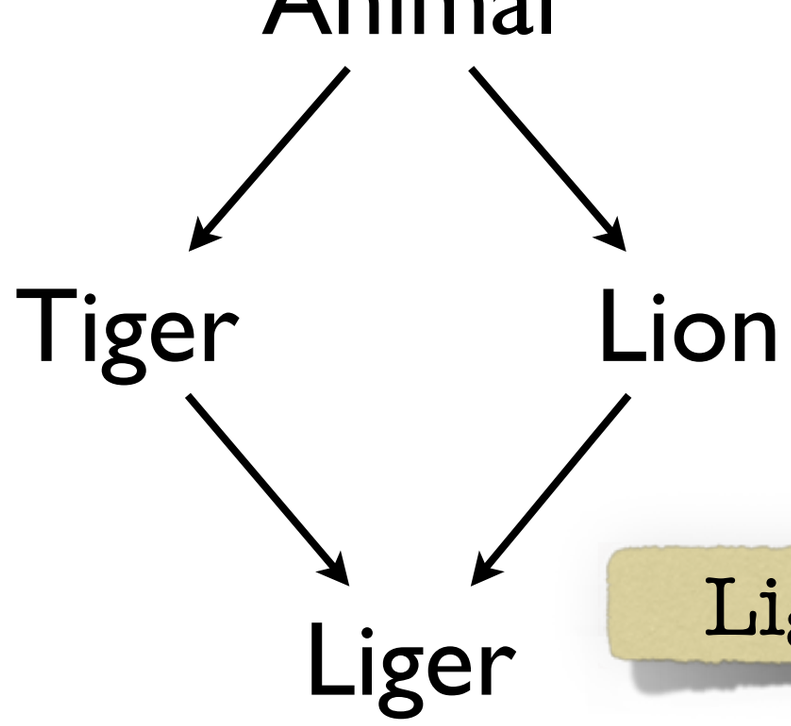
Lion

So far so good, no diamonds, but wait, what's this?!

A LIGER!

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance



Liger: super confused

```
class Liger extends (Tiger, Lion) {
   void noise() { super.noise(); };
}
```

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance
- The problem is that with multiple inheritance, there is no sensible way to resolve a call to **super**
- In Java we simply ban all multiple inheritance

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance
- The problem is that with multiple inheritance, there is no sensible way to resolve a call to **super**
- In Java we simply ban all multiple inheritance

Q. Are there no Ligers in Java?

Deadly Diamond of Death

- The *Deadly Diamond of Death*, (DDD) comes up when we consider inheritance
- The problem is that with multiple inheritance, there is no sensible way to resolve a call to **super**
- In Java we simply ban all multiple inheritance

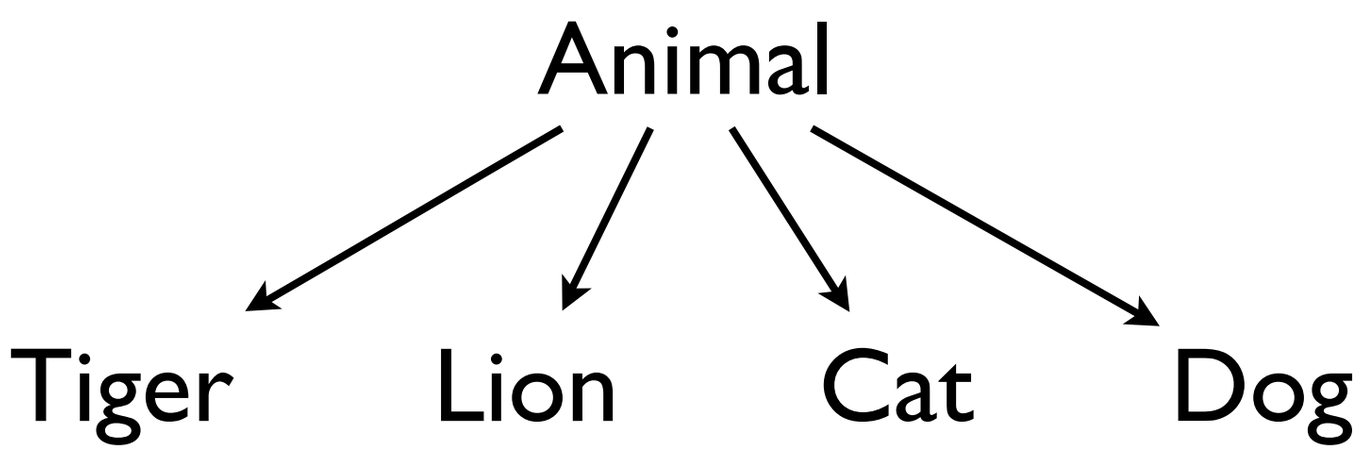
Q. Are there no Ligers in Java?

A. Not really. It would have to have to be a thing in its own right, and have a has-a lion and has-a tiger

Interfaces

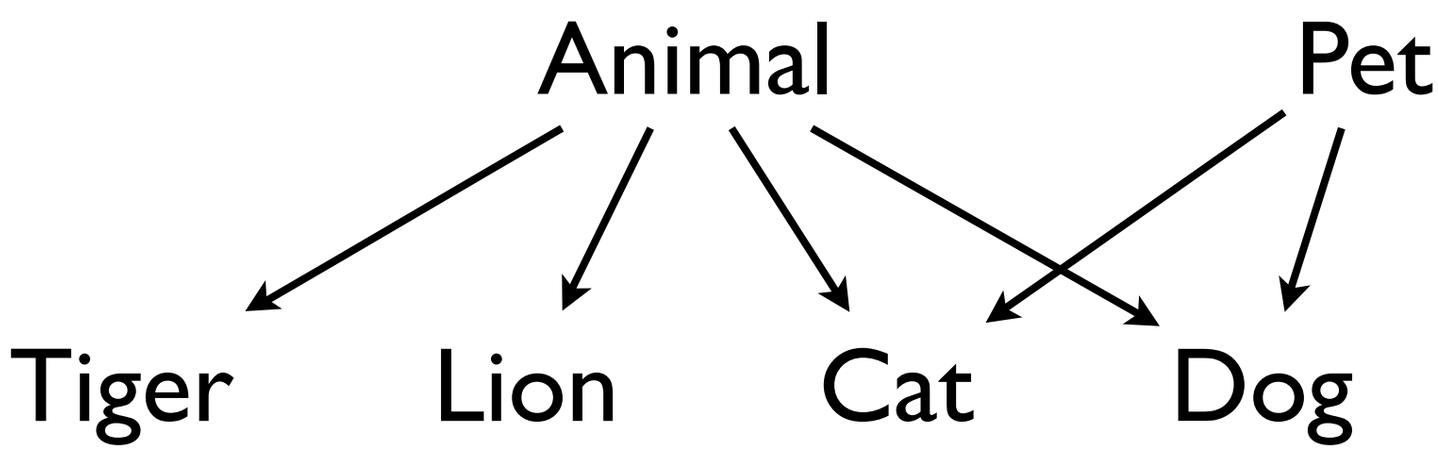
Interfaces

- Something like multiple inheritance is still something we'd really like to have



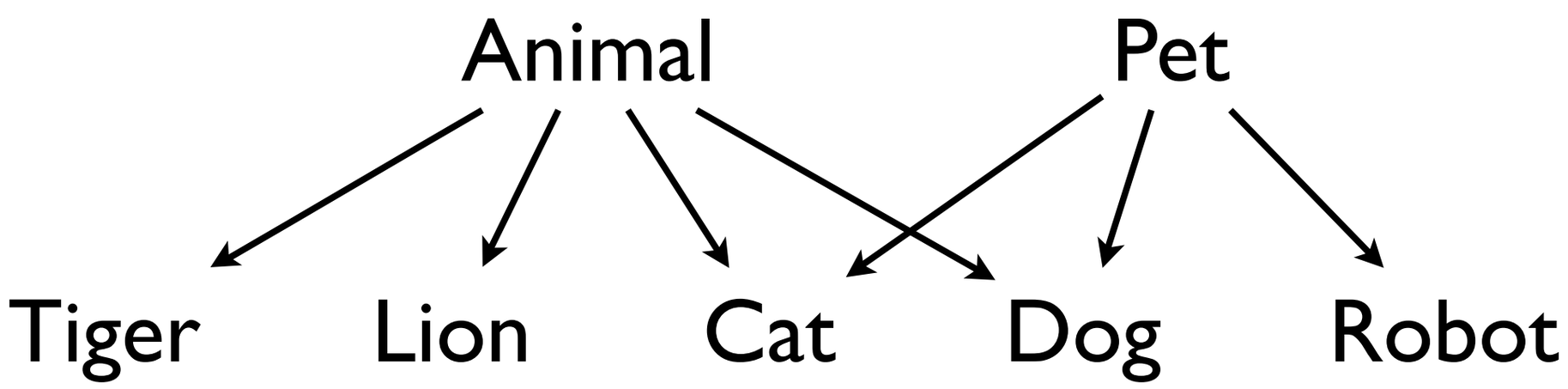
Interfaces

- Something like multiple inheritance is still something we'd really like to have



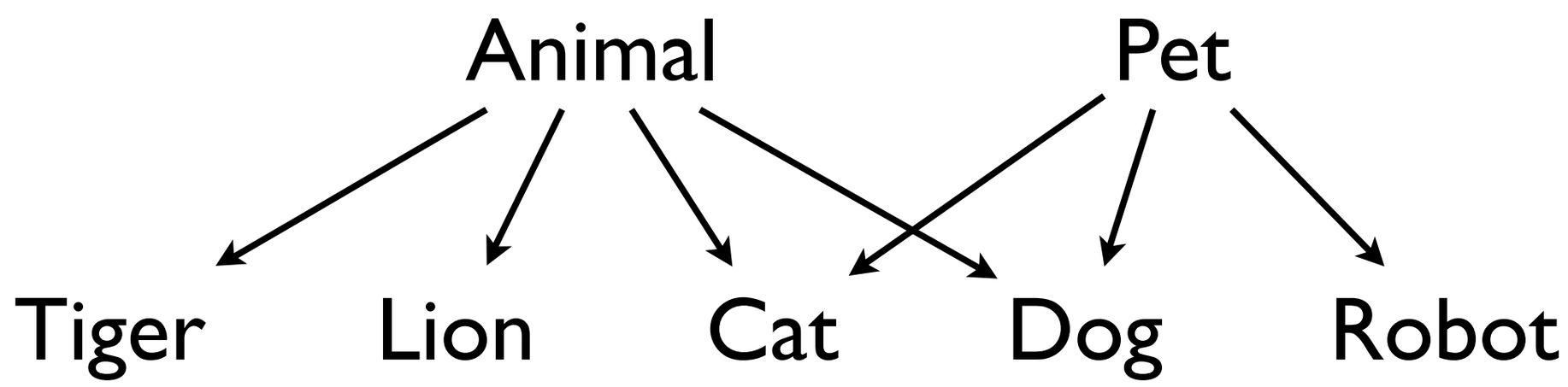
Interfaces

- Something like multiple inheritance is still something we'd really like to have



Interfaces

- Something like multiple inheritance is still something we'd really like to have



- We resolve this by adding an *interface* that can implemented by our classes

Interfaces

- An interface is like an abstract class with only final abstract methods and has no attributes

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

```
interface Pet {
  void cuddle();
}
```

```
class Dog extends Animal implements Pet {
  @Override
  void noise () { /* Woof! */ }; // from Animal

  @Override
  void cuddle () { /* Smelly cuddle */ }; // from Pet
}
```

Encapsulation

Encapsulation

- So far I've been **very** naughty
- In almost all cases, it's preferable to keep the attributes of a class hidden
- When we need to expose them, it's better to use a getter and setter instead
- This leaves us free to change the implementation of a class even after it's been shipped

Encapsulation

- If you promise nothing, you're useless
- But you don't need to say *how* you'll keep your promises
- When you expose an attribute, you promise both its existence and its implementation
- Encapsulation allows you to promise only the existence of an attribute, but leaves you free to implement however you wish

Encapsulation

- Relationships between attributes are *invariants*

```
class Polygon {
  Point[] points;
  int sides;
  …
}
```

- Here, *points* and *sides* are connected a formula:

```
points.length == sides
```

- This formula should *always* hold true, and we call it an *invariant* because it should never vary

Encapsulation

- Relationships between attributes are *invariants*

```
class Polygon {
  Point[] points;
  int sides;
  …
}
```

Q. Why does keeping the invariant true matter?

- Here, *points* and *sides* are connected a formula:

```
points.length == sides
```

- This formula should *always* hold true, and we call it an *invariant* because it should never vary

Encapsulation

- Relationships between attributes are *invariants*

```
class Polygon {
  Point[] points;
  int sides;
  …
}
```

Q. Why does keeping the invariant true matter?

A. Other programs might start **relying** on this relationship

- Here, *points* and *sides* are connected a formula:

```
points.length == sides
```

- This formula should *always* hold true, and we call it an *invariant* because it should never vary

Encapsulation

- Setting up the invariant is usually done in the constructor

```
class Polygon {
  Point[] points;
  int sides;

  Polygon(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  …
}
```

Q. Java removes the default constructor when we add one explicitly like this. Why is that good?

Encapsulation

- Setting up the invariant is usually done in the constructor

```
class Polygon {
  Point[] points;
  int sides;

  Polygon(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  …
}
```

Q. Java removes the default constructor when we add one explicitly like this. Why is that good?

A. Because now we can't break the invariant when creating an object!

Encapsulation

- However, abusing the invariant is all too easy!

```
Points[] ps = new Points[] {
   { new Point(0,0)
   , new Point(1,1)
   , new Point(2,0)
   }
Polygon p = new Polygon(ps);
p.points  = new Points[] { };
```

Q. Does the invariant hold true here?

Encapsulation

- However, abusing the invariant is all too easy!

```
Points[] ps = new Points[] {
   { new Point(0,0)
   , new Point(1,1)
   , new Point(2,0)
   }
Polygon p = new Polygon(ps);
p.points  = new Points[] { };
```

Q. Does the invariant hold true here?

A. No! points.size = 0, points.sides = 3

Encapsulation

- However, abusing the invariant is all too easy!

```
Points[] ps = new Points[] {
   { new Point(0,0)
   , new Point(1,1)
   , new Point(2,0)
   }
Polygon p = new Polygon(ps);
p.points  = new Points[] { };
```

Q. Does the invariant hold true here?

A. No! points.size = 0, points.sides = 3

Q. Is there another invariant we should have included?

Encapsulation

- However, abusing the invariant is all too easy!

```
Points[] ps = new Points[] {
   { new Point(0,0)
   , new Point(1,1)
   , new Point(2,0)
   }
Polygon p = new Polygon(ps);
p.points  = new Points[] { };
```

Q. Does the invariant hold true here?

A. No! points.size = 0, points.sides = 3

Q. Is there another invariant we should have included?

A. Yes! We could have insisted that points.length > 2

Encapsulation

- We solve this problem by *encapsulating* the attributes of our class
- Encapsulation is very *easy* to do

encapsulation = getters and setters + data hiding

- You should do this, right after this lecture!

Getters and Setters

Getters and Setters

- *Getters* and *setters* are what we name methods that are used to *access* and *mutate* attributes

Sometimes getters and setters are called accessors and mutators

- Often either the getter or the setter is simply passing the underlying attribute, and the other is ensuring *invariants* hold true

What if the variable is a simple pass-through?

Getters and Setters

- Adding a getter:

```
class Polygon {
   Point[] points;
   int sides;
   …
   Point[] getPoints() {
      return points;
   };
}
```

This case is easy: we just pass **points** through as the return value

Getters and Setters

- Adding a setter:

```
class Polygon {
   Point[] points;
   int sides;
   …
   Point[] getPoints() {
      return points;
   };
   void setPoints(Point[] points) {
      this.points = points;
      sides = points.length
   };
}
```

This time we also need to update **sides** to keep in step with the new **points**

Getters and Setters

- Now if we want to access or mutate the *points* we should use the *getPoints* or *setPoints* methods instead
- This requires discipline: we could still change *points* if we wanted to

Getters and Setters

- Now if we want to access or mutate the *points* we should use the *getPoints* or *setPoints* methods instead
- This requires discipline: we could still change *points* if we wanted to

Q. Should we add a **getSides** and **setSides** as well?

Getters and Setters

- Now if we want to access or mutate the *points* we should use the *getPoints* or *setPoints* methods instead
- This requires discipline: we could still change *points* if we wanted to

Q. Should we add a **getSides** and **setSides** as well?

A. Let's try!

Getters and Setters

- We could do a naive version of *getSides* quite easily:

```
class Polygon {
   Point[] points;
   int sides;
   …
   int getSides() {
      return sides;
   };
   …
}
```

Getters and Setters

- We could do a naive version of *getSides* quite easily:

```
class Polygon {
   Point[] points;
   int sides;
   …
   int getSides() {
      return sides;
   };
   …
}
```

Q. We could imagine returning **points.length** instead … is that a good idea?

Getters and Setters

- We could do a naive version of *getSides* quite easily:

```
class Polygon {
   Point[] points;
   int sides;
   …
   int getSides() {
      return sides;
   };
   …
}
```

Q. We could imagine returning **points.length** instead … is that a good idea?

A. Yes: **sides** isn't needed per se

Getters and Setters

- We could do a naive version of *getSides* quite easily:

```
class Polygon {
   Point[] points;
   int sides;
   …
   int getSides() {
      return sides;
   };
   …
}
```

Q. We could imagine returning **points.length** instead … is that a good idea?

A. Yes: **sides** isn't needed per se

A. But: In the general case **sides** is a memoized value: it might be efficient to store it here

Getters and Setters

- The setter isn't so obvious, let's consider this:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides) {
    this.sides = sides;
  };
  …
}
```

Getters and Setters

- The setter isn't so obvious, let's consider this:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides) {
    this.sides = sides;
  };
  …
}
```

Q. What's wrong with this code?

Getters and Setters

- The setter isn't so obvious, let's consider this:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides) {
    this.sides = sides;
  };
  …
}
```

Q. What's wrong with this code?

A. Changing **sides** will violate the invariance!

Getters and Setters

- The setter isn't so obvious, let's consider this:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides) {
    this.sides = sides;
  };
  …
}
```

Q. What's wrong with this code?

A. Changing **sides** will violate the invariance!

Q. Could we do something else?

Getters and Setters

- Here's another setter we might try:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides, Point[] ps) {
    this.sides = sides;
    Point[] result = Arrays.copyOf(points, sides);
    System.arrayCopy( points, 0
                    , result, points.length
                    , ps.length );
    points = result;
 };
  …
}
```

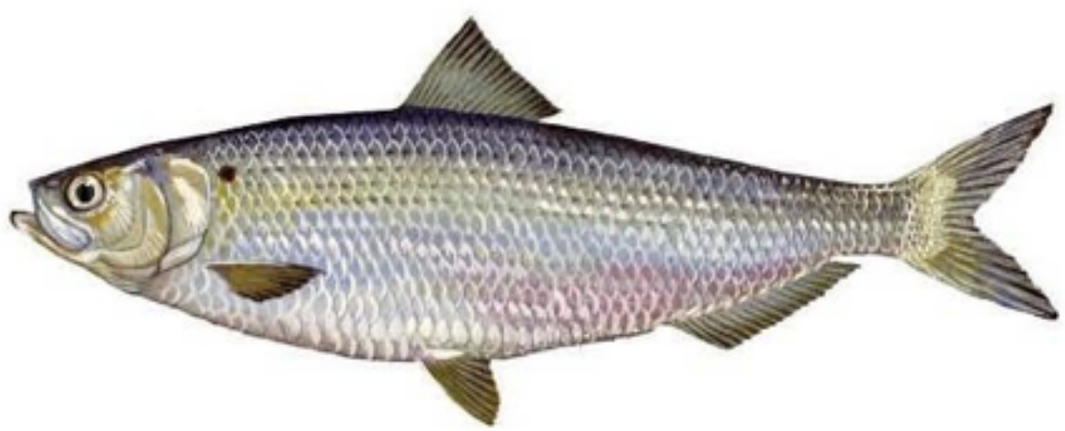
Getters and Setters

- Here's another setter we might try:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides, Point[] ps) {
    this.sides = sides;
    Point[] result = Arrays.copyOf(points, sides);
    System.arrayCopy( points, 0
                    , result, points.length
                    , ps.length );
    points = result;
 };
  …
}
```

This code is horribly smelly!

Q. What's wrong with it?



Getters and Setters

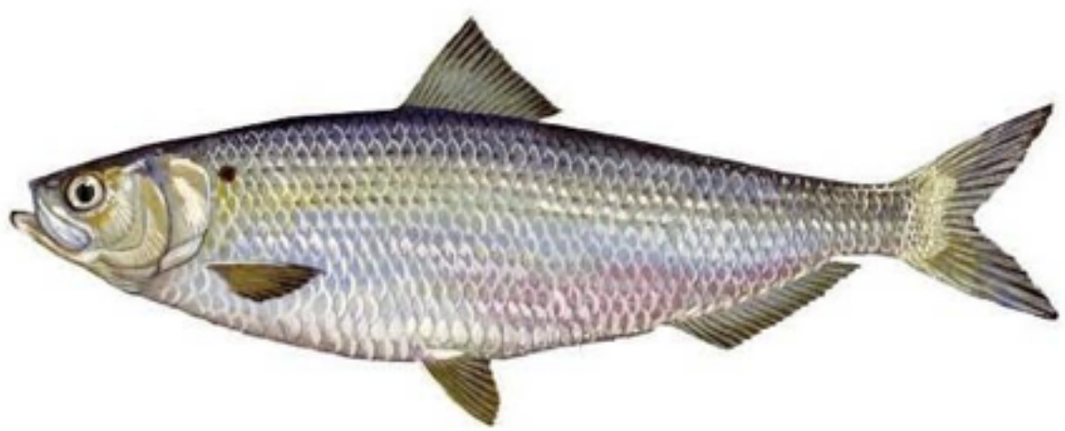
- Here's another setter we might try:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides, Point[] ps) {
    this.sides = sides;
    Point[] result = Arrays.copyOf(points, sides);
    System.arrayCopy( points, 0
                    , result, points.length
                    , ps.length );
    points = result;
 };
  …
}
```

A. We might want to **reduce** the number of sides

This code is horribly smelly!

Q. What's wrong with it?



Getters and Setters

- Here's another setter we might try:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides, Poi
    this.sides = sides;
    Point[] result = Arrays.copyOf(points, sides);
    System.arrayCopy( points, 0
                    , result, points.length
                    , ps.length );
    points = result;
 };
  …
}
```

A. We might want to **reduce** the number of sides

A. Array copying is usually bad, perhaps we might want to use a list instead

This code is horribly smelly!

Q. What's wrong with it?

Getters and Setters

- Here's another setter we might try:

```
class Polygon {
  Point[] points;
  int sides;
  …
  void setSides(int sides, Poi
    this.sides = sides;
    Point[] result = Arrays.co
    System.arrayCopy( points,
                    , result,
                    , ps.length );
    points = result;
 };
  …
}
```

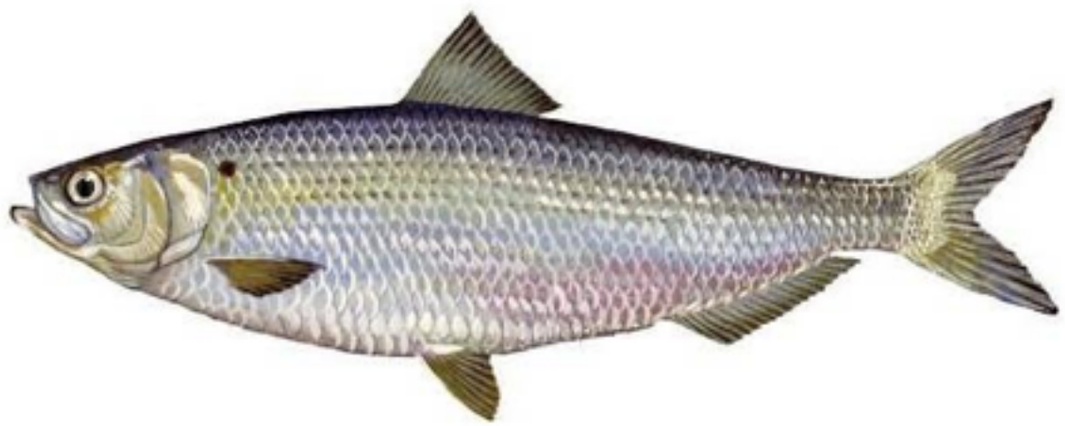
A. We might want to **reduce** the number of sides

A. Array copying is usually bad, perhaps we might want to use a list instead

A. Subclasses, like triangle, might require the sides never to change!

This code is horribly smelly!

Q. What's wrong with it?



Getters and Setters

- Sometimes, it's worth considering whether it's worth exposing a setter in the first place
- In our case, setting *sides* directly seems like a fundamentally bad idea

Getters and Setters

- Sometimes, it's worth considering whether it's worth exposing a setter in the first place
- In our case, setting *sides* directly seems like a fundamentally bad idea

Q. Even if we didn't add **setSides**, our code is still fishy, what's wrong with it?

Getters and Setters

- Sometimes, it's worth considering whether it's worth exposing a setter in the first place
- In our case, setting *sides* directly seems like a fundamentally bad idea

Q. Even if we didn't add **setSides**, our code is still fishy, what's wrong with it?

A. Programmers are **never** always disciplined or well behaved! I bet they'll **still** change **points** or **sides** directly!

Data Hiding

Data Hiding

- Classes and their features (methods and attributes) can be given *access modifiers*, which change their visibility
- Sometimes encapsulation is explained in terms of *data hiding*, but it's really the combination of providing good getters, setters, together with sensible data hiding
- These modifiers are there to protect you against yourself!
- NB! This is not about *security*: your class still exposes its bytecode for hackers to inspect)

Data Hiding

- You should routinely hide your data as much as possible

```
class Polygon {
  private Point[] points;
  private int sides;

  void setPoints(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  void getPoints(Points[] points) {
    return points;
  }
  void getSides() { return sides; }
}
```

Just sprinkle **private** on your attributes

Member Visibility

Modifier	Class	Package	World
Public	✓	✓	✓
Default (no modifier)	✓	✓	✗
Private	✓	✗	✗

Member Visibility

Modifier	Class	Package	World
Public	✓	✓	✓
Default (no modifier)	✓	✓	✗
Private	✓	✗	✗

main is **public**

Member Visibility

Modifier	Class	Package	World
Public	✓	✓	✓
Default (no modifier)	✓	✓	✗
Private	✓	✗	✗

main is **public**

methods usually **default**

Member Visibility

Modifier	Class	Package	World
Public	✓	✓	✓
Default (no modifier)	✓	✓	✗
Private	✓	✗	✗

main is **public**

methods usually **default**

attributes usually **private**

Data Hiding

- Now that we've made our implementation private, there's another benefit!

```
class Polygon {
  private Point[] points;
  private int sides;

  void setPoints(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  void getPoints(Points[] points) {
    return points;
  }
  void getSides() { return sides; }
}
```

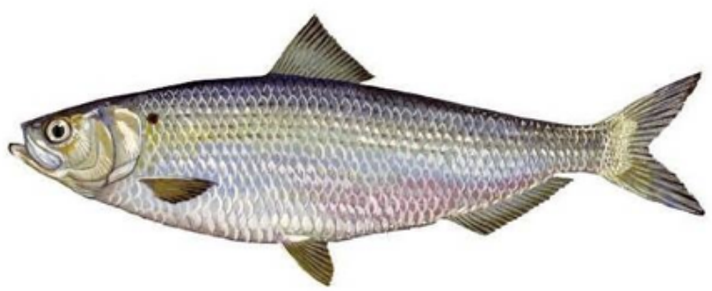
Data Hiding

- Now that we've made our implementation private, there's another benefit!

```
class Polygon {
  private Point[] points;
  private int sides;

  void setPoints(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  void getPoints(Points[] points) {
    return points;
  }
  void getSides() { return sides; }
}
```

Q. Which redundant detail can we safely remove?



Data Hiding

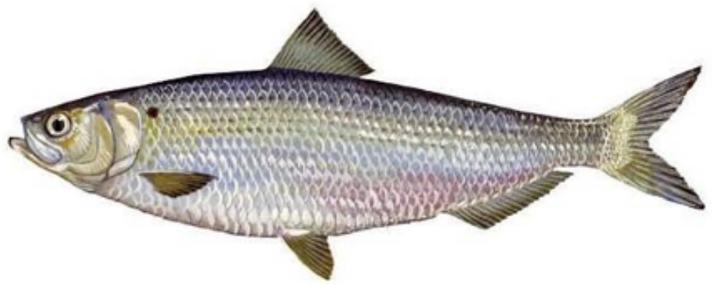
- Now that we've made our implementation private, there's another benefit!

```
class Polygon {
  private Point[] points;
  private int sides;

  void setPoints(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  void getPoints(Points[] points) {
    return points;
  }
  void getSides() { return sides; }
}
```

Q. Which redundant detail can we safely remove?

A. **sides** is now totally useless, because it is contained in **points.length**



Data Hiding

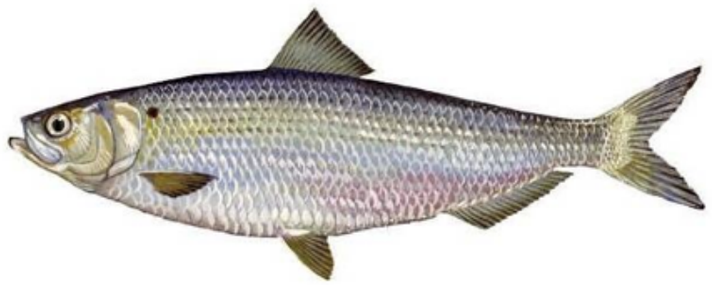
- Now that we've made our implementation private, there's another benefit!

```
class Polygon {
  private Point[] points;


  void setPoints(Points[] points) {
    this.points = points;
    sides = points.length;
  }
  void getPoints(Points[] points) {
    return points;
  }
  void getSides() { return points.length; }
}
```

Q. Which redundant detail can we safely remove?

A. **sides** is now totally useless, because it is contained in **points.length**



Data Hiding

- As a general guide, try to keep your methods and attributes private
- Limit your exposure to the world!

Immutable Objects

Immutable Objects

- Sometimes we want objects that *cannot* be mutated: we call these *immutable* objects

Q. So far, we've seen one class of objects is immutable, and one that's a good candidate ... which ones?

A. **String** is immutable

A. Perhaps **Point** ought to be immutable

Immutable Objects

- The *Point* class represents points in space
- The *origin* is a particularly special point: we'd be shocked if it changed
- The *final* keyword says that the value of a variable should not be changed

Immutable Objects

- It is the *value* that is made final: this applies to references too (since references are values)
- So here we might try to make just *origin* final:

```
final Point origin = new Point(0,0);
origin = new Point(4,2); // BOGUS!
```

```
Point.java:42: error: cannot assign a value to final variable origin
```

Immutable Objects

- It is the *value* that is made final: this applies to references too (since references are values)
- So here we might try to make just *origin* final:

```
final Point origin = new Point(0,0);
origin = new Point(4,2); // BOGUS!
```

```
Point.java:42: error: cannot assign a value to final variable origin
```

Q. This seems to have worked, so what's the catch?

Immutable Objects

- It is the *value* that is made final: this applies to references too (since references are values)
- So here we might try to make just *origin* final:

```
final Point origin = new Point(0,0);
origin = new Point(4,2); // BOGUS!
```

```
Point.java:42: error: cannot assign a value to final variable origin
```

Q. This seems to have worked, so what's the catch?

A. We can still abuse the origin **object**!

Immutable Objects

- It is the *value* that is made final: this applies to references too (since references are values)
- So here we might try to make just *origin* final:

```
final Point origin = new Point(0,0);
origin = new Point(4,2); // BOGUS!
```

```
Point.java:42: error: cannot assign a value to final variable origin
```

- Alas we can still modify the object referenced by *origin* like this:

```
origin.x = 4;
```

Immutable Objects

- To make Point immutable we need to make its attributes *final*

```
class Point {
  final double x;
  final double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y
  }
}
```

- If we try to assign to *x* or *y* the compiler screams:

```
Point origin = new Point(0, 0);
origin.x = 3
```

```
Point.java:42: error: cannot assign a value to final variable x
```

Immutable Objects

- To make Point immutable we need to make its attributes *final*

```
class Point {
  final double x;
  final double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y
  }
}
```

Q. There's still a bug, what's the problem?

- If we try to assign to *x* or *y* the compiler screams:

```
Point origin = new Point(0, 0);
origin.x = 3
```

```
Point.java:42: error: cannot assign a value to final variable x
```

Immutable Objects

- To make Point immutable we need to make its attributes *final*

```
class Point {
  final double x;
  final double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y
  }
}
```

Q. There's still a bug, what's the problem?

- If we try to assign to *x* or *y* the compiler screams:

```
Point origin = new Point(0, 0);
origin.x = 3
```

A. origin needs to be final too!

```
Point.java:42: error: cannot assign a value to final variable x
```

Immutable Objects

- To make Point immutable we need to make its attributes *final*

```
class Point {
  final double x;
  final double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y
  }
}
```

- And we also need to make sure that *origin* is final

```
final Point origin = new Point(0, 0);
```

- Now there's no way to mess around with *origin*

Immutable Objects

- If the object is immutable, is it useful?
- The object can't be modified, but we happily use one as a seed potato

```
class Point {
  final double x;
  final double y;

  …
  Point move(double dx, double dy) {
    return (new Point(x + dx, y + dy));
  }
}
```

Each time we move an object, it actually just returns a new object at the new location

Immutable Objects

- Strings are immutable for a number of reasons
- Strings are cached in the "String Pool", so that references to the same string really point to the piece of memory
- You'd be rather annoyed if the strings could be changed through references

Q. (Hard) What's the danger introduced by immutable strings?

Immutable Objects

- Strings are immutable for a number of reasons
- Strings are cached in the "String Pool", so that references to the same string really point to the piece of memory
- You'd be rather annoyed if the strings could be changed through references

Q. (Hard) What's the danger introduced by immutable strings?

A. The substring method can cause a **memory leak** in your program, where memory gets created but never released (this was fixed in JDK 1.7)

Immutable Objects

- Immutable objects don't change, so you can exploit the fact that their hash is stable
- Some objects assume as part of their invariance that they contain immutable things: such as a HashSet
- Immutable objects are thread safe, so they can be used in parallel algorithms

Immutable Objects

- The *final* keyword means different things in different contexts
  - On variables, it means the variable cannot be reassigned (as we saw)
  - On methods, it means the method cannot be overridden
  - On classes, it means the class cannot be extended

Immutable Objects

"Just as it is a good practice to make all fields private unless they need greater visibility, it is a good practice to make all fields final unless they need to be mutable."

— Brian Goetz
Java Concurrency in Practice