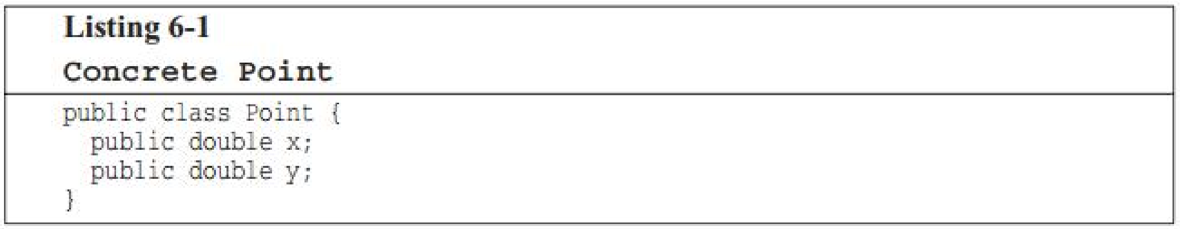
Object & DS

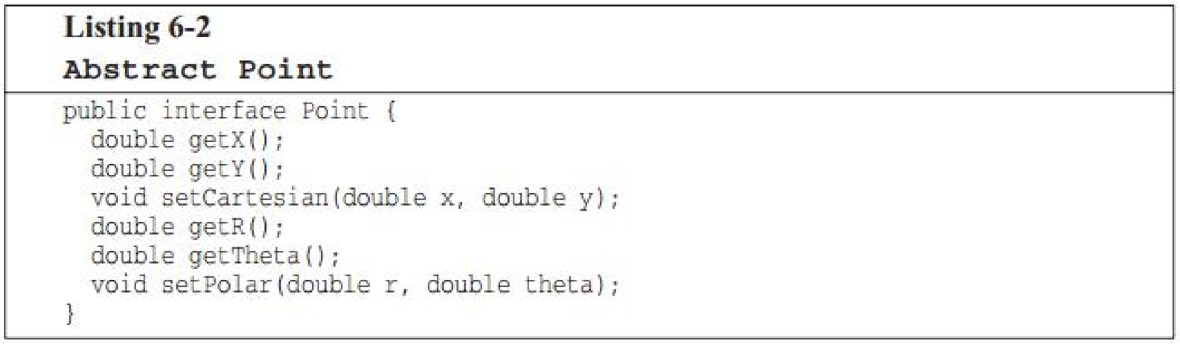
Data Abstraction:



Listing 6-1

Concrete Point

```
public class Point {
  public double x;
  public double y;
}
```



Listing 6-2

Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation(6.1) and the other completely hides it(6.2).

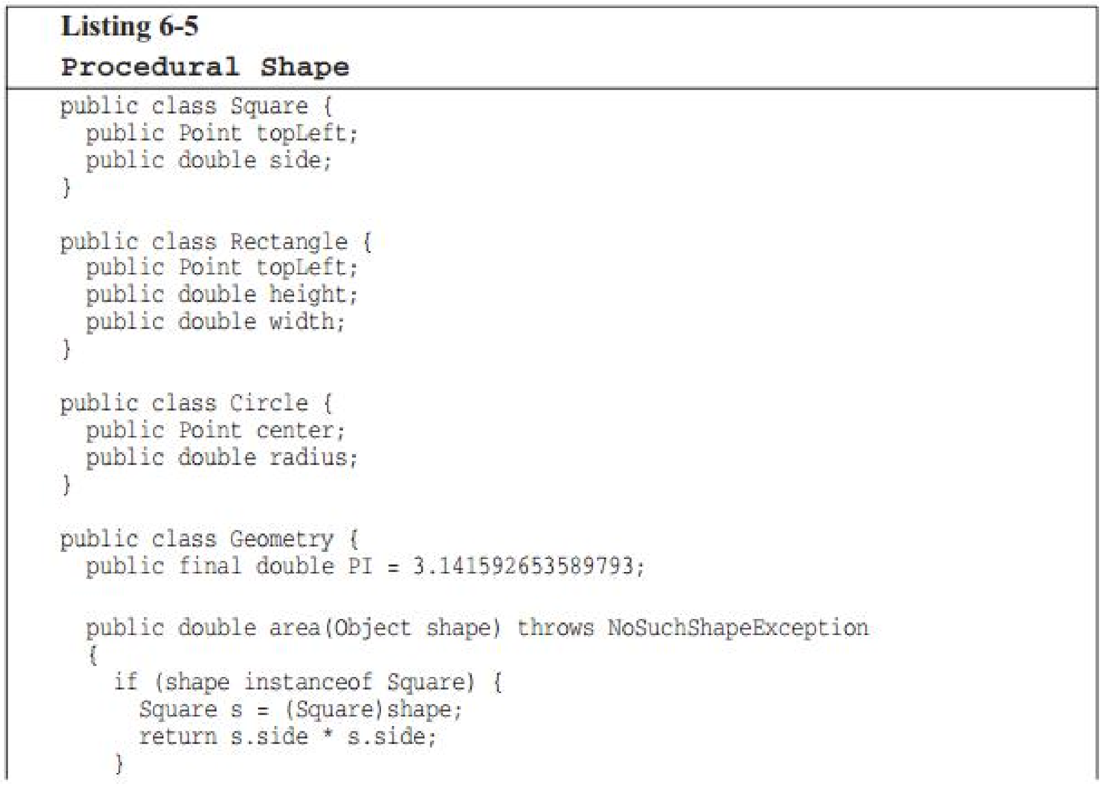
Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.

Data/Object Anti-Symmetry:

Objects hide their data behind abstractions and expose functions that operate on that data.
Data structure expose their data and have no meaningful functions.

Object & DS



Listing 6-5

Procedural Shape

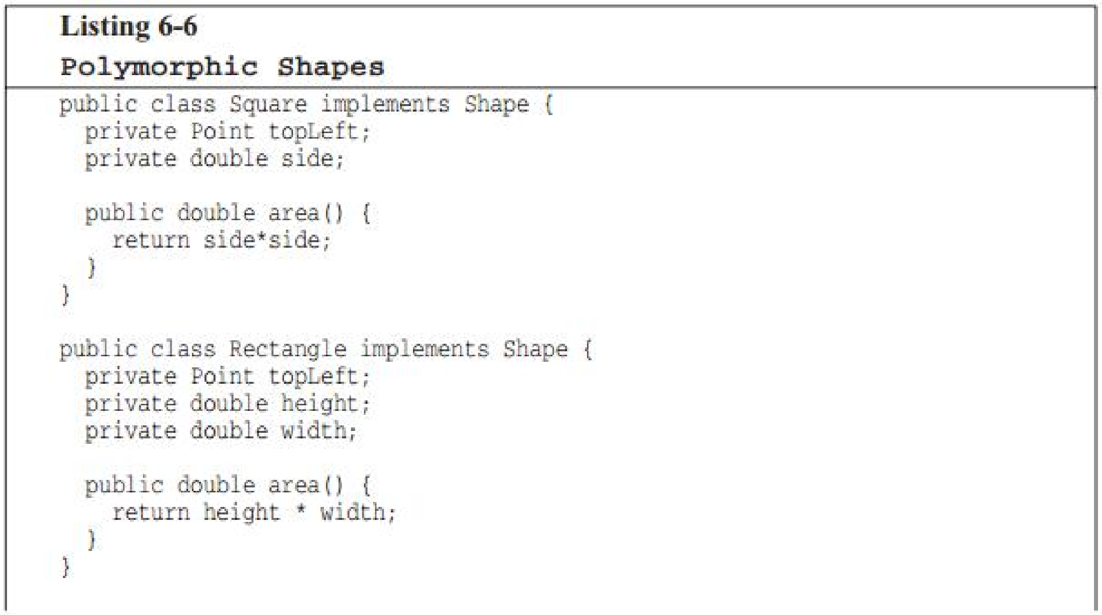
```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
```



Listing 6-6

Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

Object & DS

Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

If x an object, we should be telling it to do something;we should not be asking it about its internals.

Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.

hybrid structures that are half object and half data structure. They have functions that do significant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them.

The Law of Demeter:

An object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.
More precisely, the Law of Demeter says that a method f of a class C should only call the methods of these:

- C
- An object created by f
- An object passed as an argument to f
- An object held in an instance variable of C

The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.
The following code appears to violate the Law of Demeter (among other things) because it calls the getScratchDir() function on the return value of getOptions() and then callsgetAbsolutePath() on the return value of getScratchDir().

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

This kind of code is often called a train wreckbecause it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided. It is usually best to split them up as follows:

Object & DS

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Data Transfer Objects:

A class with public variables and no functions. This is sometimes called a data transfer object**(DTO)**.

Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit.

Active Records are special forms of DTOs. They are data structures with public (or bean accessed) variables; but they typically have navigational methods like save and find.

The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).

Conclusion:

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects.

Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

In any given system we will sometimes want the flexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the flexibility to add new behaviors, and so in that part of the system we prefer data types and procedures.

Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.