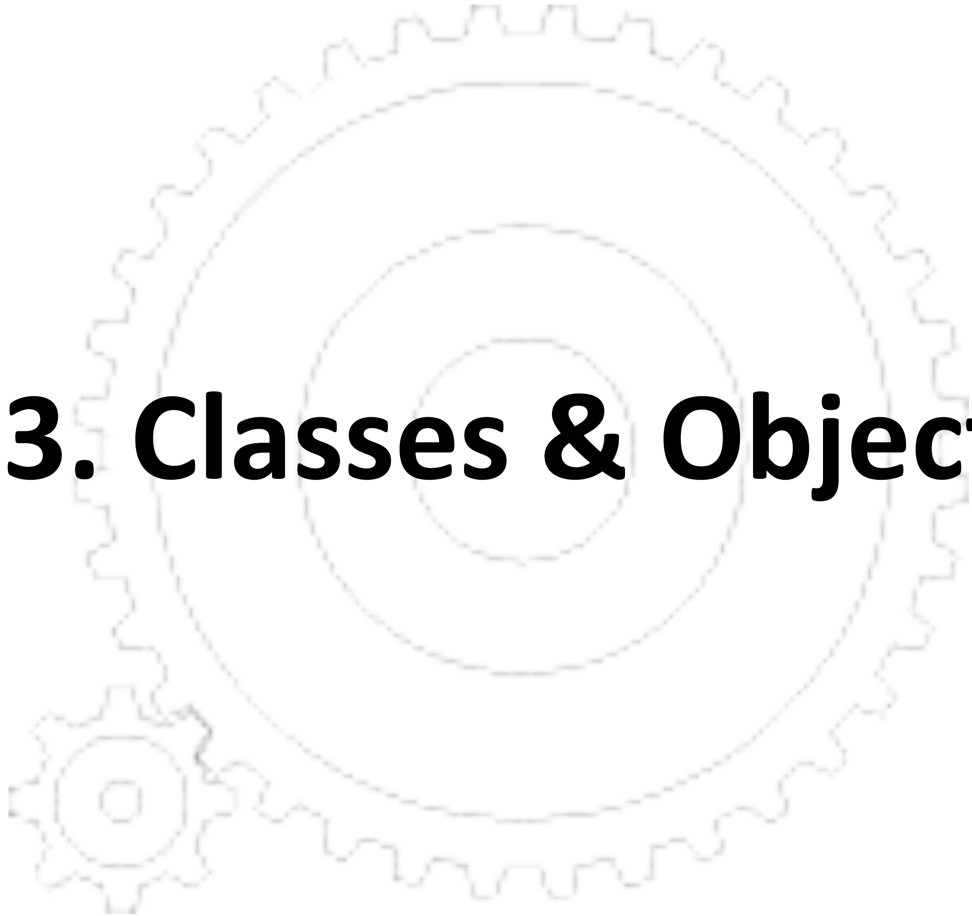
G51OOP



3. Classes & Objects

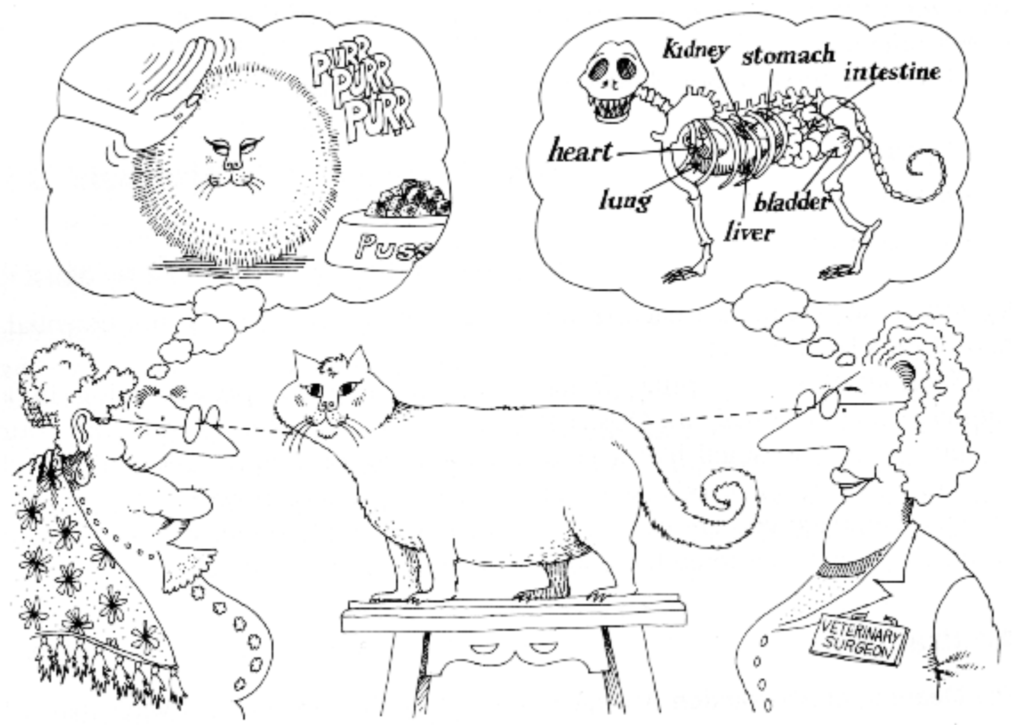
Object Oriented Programming

Colin Higgins

Overview

- Abstraction
  - Procedural
  - Data Type
- Object Orientation
- Class Libraries - Importing
- The Car Class Example
  - Constructing
  - Member Access
  - Using
- Member Methods
- Invoking Methods
- Constructors
- Access Protection
- Examples
  - Counter
  - Money

Abstract Data Types - Procedural Abstraction



- The key to building anything but the smallest systems is abstraction.
- The history of programming languages is one of increasing abstraction.
- Abstraction allows us to form a view of a problem which considers only necessary info and hides away unnecessary detail.
- Abstraction can also be used to give different people different views of the same problem.

Procedural Abstraction

- Imagine that I write a Java method called power which raises one number to the power of another. I might show you the method signature with suitable comments describing what the method does. You would then know how to use it in your own program without having to bother with its implementation.

```
// Raise the number n to the power p, return the answer.
// Works for floating point n and positive and negative
// integer p.
public static float power(float n, int p );
```

- Now, you have an abstract view of the method.

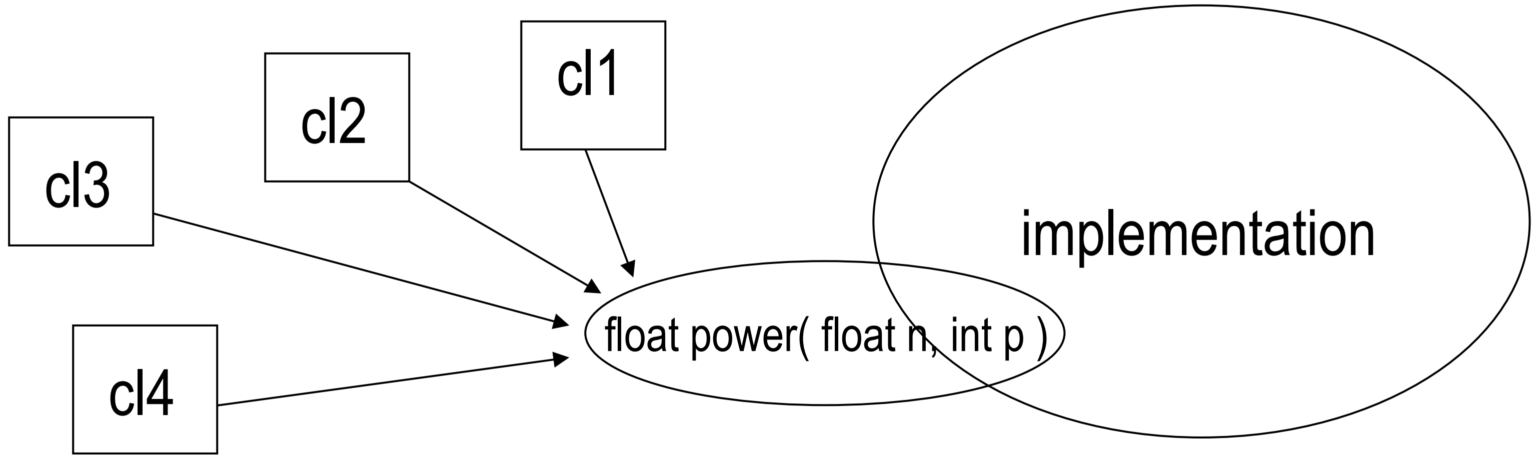
Procedural Abstraction

- Below are two different ones.

```
// implementation using a for loop
float power( float n, int p ) {
      float res = 1f;
      // test for positive or negative power
      if ( p >= 0 ) {
         // multiply 1 by n p times
         for( int i = 0; i < p; i++) {
            res = res * n;
         }
      } else {
         // divide 1 by n p times
         for( int i = p; i < 0; i++ ) {
            res = res / n;
         }
      } // end if p
      return res;
} // end power
```

```
// implementation using a while loop
float power( float n, int p ) {
      float res = 1f;
      // test for positive or negative power
      if ( p >= 0 ) {
         // multiply 1 by n p times
         while( p-- ) {
               res = res * n;
         }
      } else {
         // divide 1 by n p times
         while( p++ ) {
              res = res / n;
         }
      } // end if p
      return res;
} // end power
```

Procedural Abstraction



- Note that you are unaware of which implementation is used.
- Even more importantly, we can change the implementation - perhaps make it more efficient - and the code that invokes it won't have to be changed.
- We call this **procedural abstraction.**
- Put more formally, procedural abstraction is used to define an interface between the user and the implementor of a method resulting in more modular code.

Data Abstraction

- Data Abstraction takes the idea of abstraction much further.
- Instead of defining a method, a whole new data type is defined.
- Each language provides a basic set of types (e.g. float, int, char in Java) and gives the user operations for manipulating variables of these types (e.g. + - * etc).
- Data Abstraction allows the definition of new types, complete with a set of operations or methods, which appear as if they were part of the language. We call these Abstract Data Types (ADTs).
- The idea is that a new ADT is defined so that its users see an interface which provides an abstract view of the type and only its implementers see its internal details.

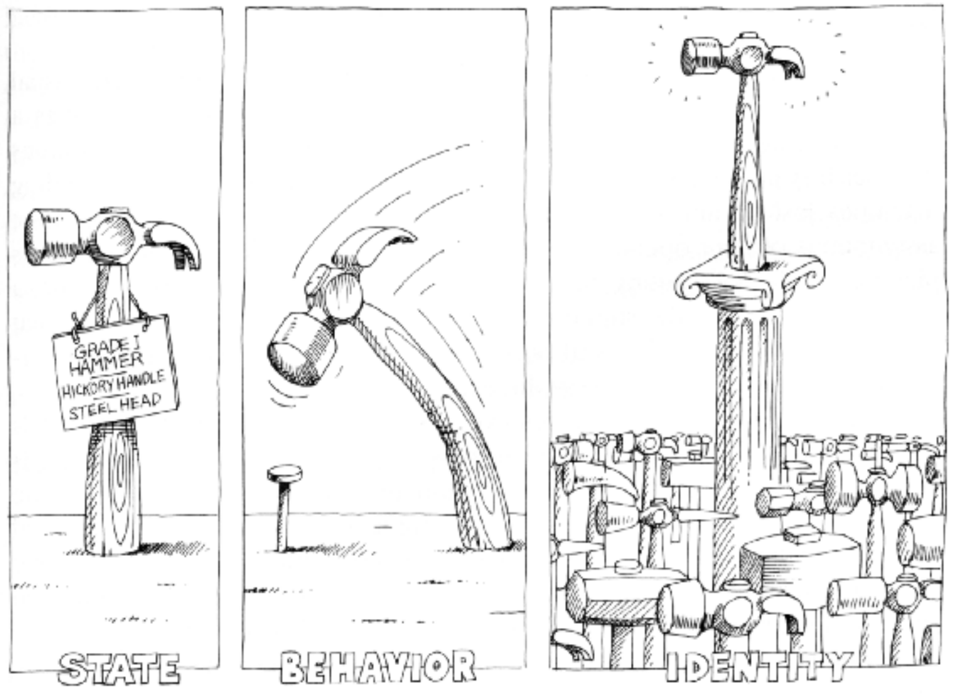
Data Abstraction

- We talk about each ADT having a user and an implementor. These may represent different members of a programming team. Thus ADTs give us a way of breaking a program into a modular structure.
- The methods exactly specify the interface to the type (i.e. how it can be used).
- The internal details (e.g. any data) shouldn’t be accessed apart from through methods. In other words, the methods protect the ADT from being tampered with, as well as protecting the user from having to know internal details. We call this encapsulation.
- Encapsulation reduces the chances of bugs occurring and makes them easier to locate when they do.
- If we define a general enough set of methods, an ADT can be re-used in many different situations.

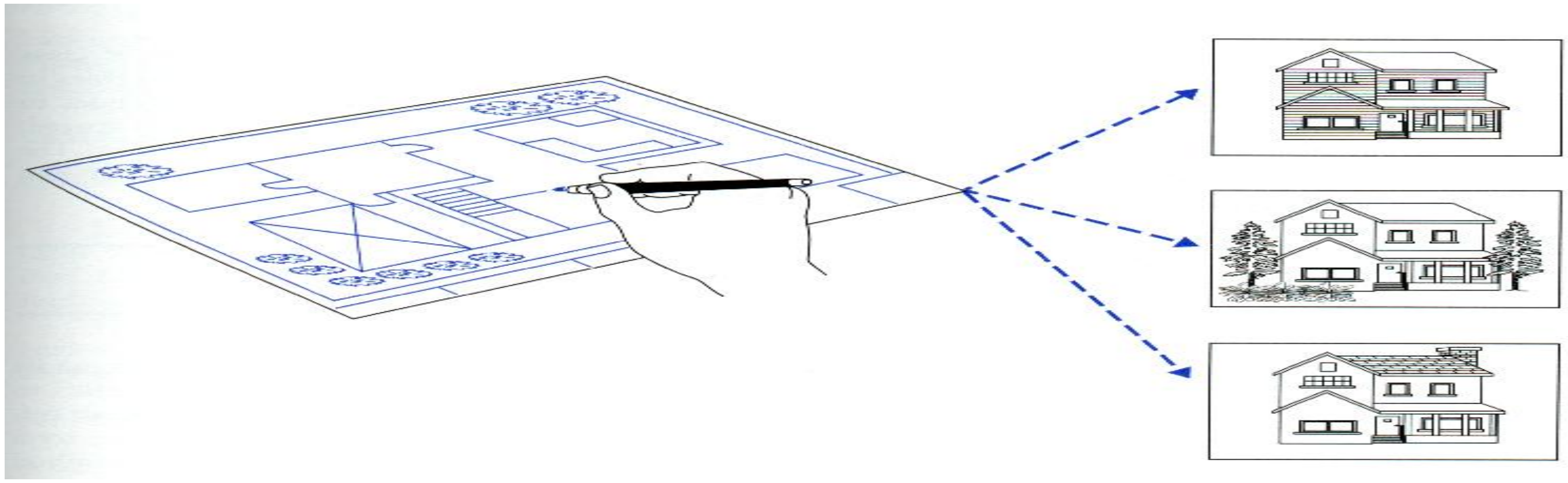
Object Orientation

- OO Systems are viewed as compositions of domain specific object abstractions, rather than data and functions.
- Objects associate data and behaviour.
- Objects in a System intercommunicate by sending messages to each other.

- An Object is a domain concept that has :
  - State
  - Behaviour
  - Identity



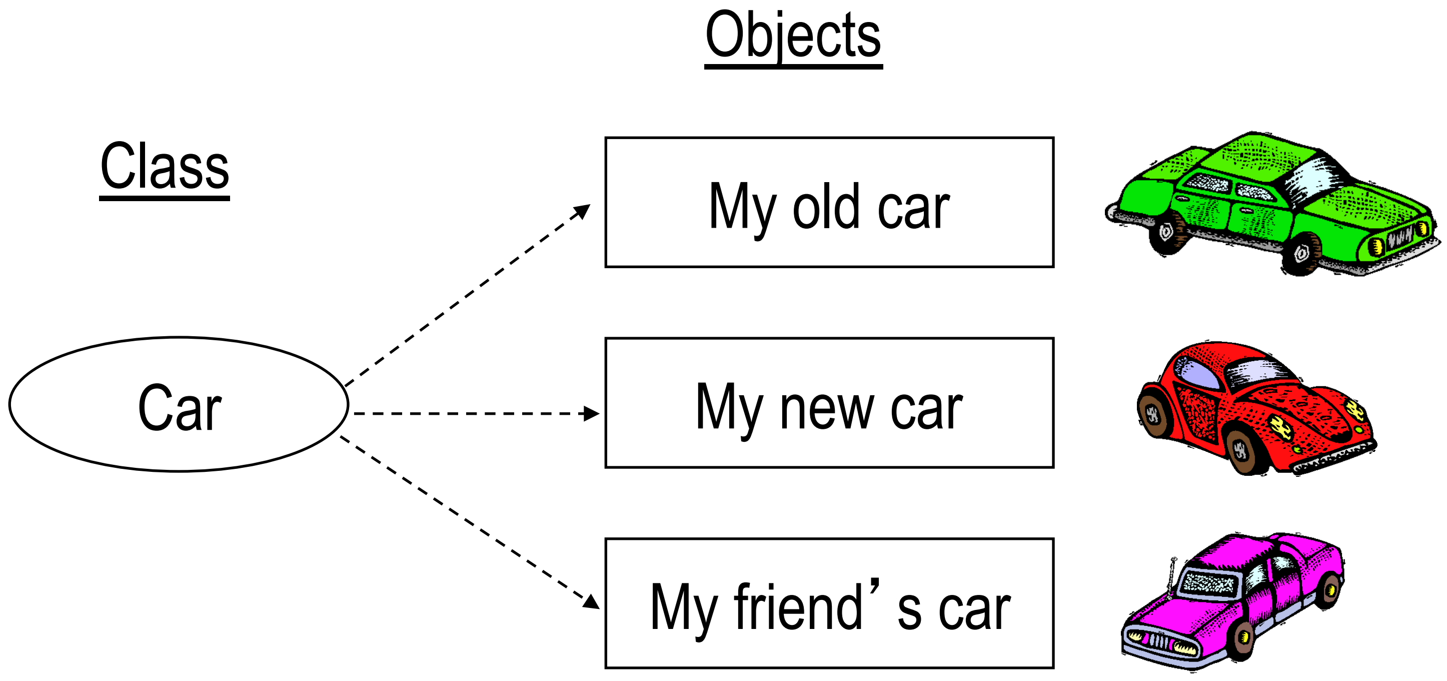
Object-Oriented Programming



- Java is *object-oriented language*
- Programs are made from software parts, classes and objects
- An *object* contains data and methods
- An object is defined by a *class*
- Multiple objects can be created from the same class

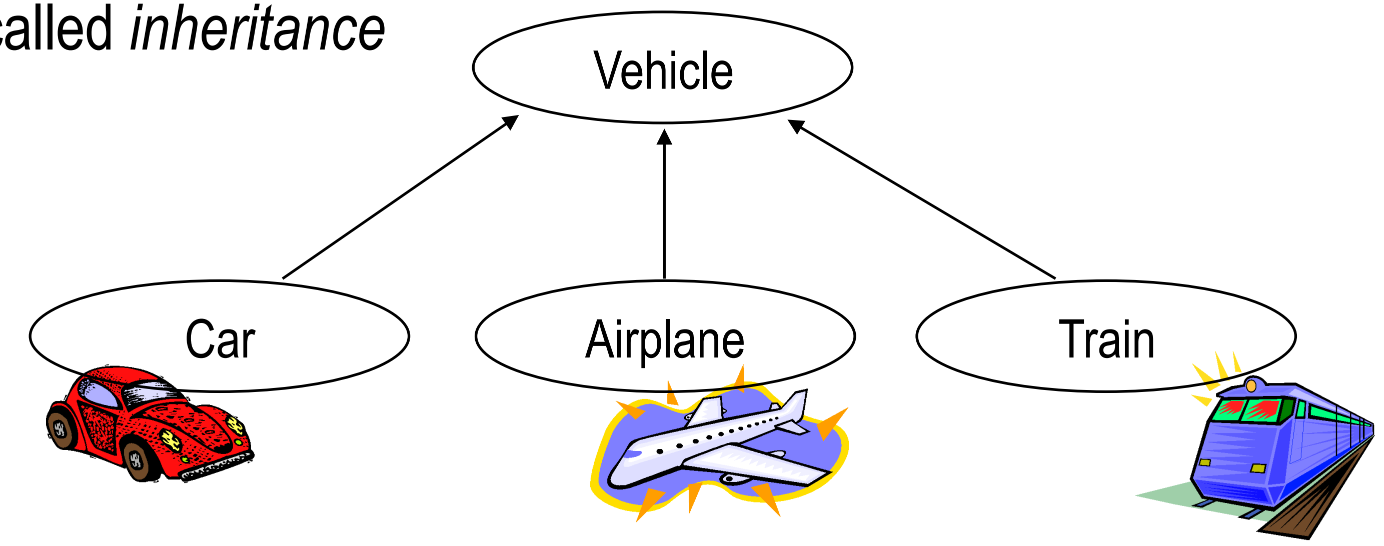
Object-Oriented Programming

- A class represents a concept and an object represents the realisation of that concept



Object-Oriented Programming

- Objects can also be derived from each other using a relationship called *inheritance*



- Objects and classes related with inheritance will be discussed in greater detail later.

Class Libraries

- The Java API is a *class library*, a group of classes that support program development
- Classes in a class hierarchy are often related by inheritance
- The classes in the Java API is separated into *packages*
- The `System` class, for example, is in package `java.lang`
- Each package contains a set of classes that relate in some way

Importing Classes from Packages

- Using a class from the Java API can be accomplished by using its fully qualified name:

```
java.lang.System.out.println ();
```

- Or, the package can be imported using an *import statement*, which has two forms:

```
import java.applet.*;
import java.util.Random;
```

- The `java.lang` package is automatically imported into every Java program

Introductory Example - The Car Class

- Suppose you need to write a traffic simulation program that watches cars going past an intersection.
- Each car has data such as **speed**, a **maximum speed**, and a **number plate**
- In traditional programming languages you'd have a structure of two floating point and one string variable for each car. With a class you combine these into one a Car object.

```
class Car {
    public String numberPlate;
    public double speed;
    public double maxSpeed;
}
```

Introductory Example - The Car Class

- These variables (numberPlate, speed and maxSpeed) are called
  - member variables,
  - instance variables, or
  - fields.
- Fields tell you what a class is and what its properties are.
- An object is a specific instance of a class with particular values for the fields.
- While a class is a blueprint for objects, an instance is a particular object.

Constructing objects with new

- To instantiate an object in Java, use the keyword `new` followed by a call to the class's constructor. Here's how you'd create a new Car variable called c.

```
Car c;
c = new Car();
```

- The first word, Car, declares the type of the variable c. Classes are types and variables of a class type need to be declared just like simple datatypes.
- Notice the Car() method. The parentheses tell you this is a method
- This is a **constructor;** a method that creates a new instance of a class.
- You could declare and instantiate an object of a class in one statement:

```
Car c = new Car();
```

Instantiating Car objects

```
class Car {
    public String numberPlate;
    public double speed;
    public double maxSpeed;
}

public class CarTestExample {

   public static void main(String[] argv) {
       Car myNewCar = new Car();
       Car myOldCar = new Car();
   }
}
```

- Notice that when you compile the above program, you get two .class files (Car.class and CarTestExample.class)

The Member Access Separator

- To access the fields of the car you use the . separator.
- The Car class, has three public fields :

  numberPlate

  speed

  maxSpeed
- Therefore if myNewCar is a new Car object, it has three new fields as well:

```
myNewCar.numberPlate
myNewCar.speed
myNewCar.maxSpeed
```

The Member Access Separator

- You use these just like you'd use any other variables of the same type. For instance:

```
class CarTestExample {
   public static void main(String args[]) {
     Car c = new Car();
     c.numberPlate = "P 123 XYZ";
     c.speed = 70.0;
     c.maxSpeed = 123.45;
     System.out.println(c.numberPlate + " is moving
           at " + c.speed + "kilometers per hour.");
   } // end main()
} // end CarTest
```

The . separator selects a specific member of a Car object by name.

Using a Car object in a different class

- You can have more than one description of a class in one source file.
- However, only one can be public, the one that gives its name to the source file.
- It is customary in Java to put every class in its own file.

CarTestExample.java

```
public class CarTestExample {
  public static void main(String[] argv) {
  Car myNewCar = new Car();
  Car myOldCar = new Car();
  }
}
```

Car.java

```
public class Car {
   public String numberPlate;
   public double speed;
   public double maxSpeed;
}
```

Using a Car object in a different class

- To do this for the previous example :
  - put the Car class in a file called Car.java;
  - put the CarTest class in a file called CarTest.java;
  - put both these files in the same directory;
  - compile both files in the usual way;
  - run CarTestExample.

- Note that Car does not have a main() method so you cannot run it. It can exist only when called by other programs that do have main() methods.
- Many of the programs you will write will use multiple classes. Next Semester, you'll learn how to use packages to organise your commonly used classes in different directories. For now you can keep all your .java source code and .class byte code files in one directory (as done by Course Master).

Methods

- Objects can provide behaviour defined by the non-static methods of their class.
- The non-static methods are called instance or object methods and they have access to the fields of an object.
- The fields and methods of a class are collectively referred to as the members of the class.For instance the Car class might have a method to make the car go as fast as it can.

```
class Car {
   public String numberPlate; // e.g. "New York A456 324"
   public double speed; // kilometers per hour
   public double maxSpeed; // kilometers per hour

   // accelerate to maximum speed
   // put the pedal to the metal
   public void floorIt() {
      this.speed = this.maxSpeed;
   }
} // class Car
```

Invoking Instance Methods

- Outside the Car class, you call the `floorIt()` method just like you reference fields, using the name of the object you want to accelerate to maximum and the .separator.

```
class CarTest2 {
  public static void main(String args[]) {
    Car c = new Car();
    c.numberPlate = "P 123 XYZ";
    c.speed = 0.0;
    c.maxSpeed = 123.45;
    System.out.println(c.numberPlate + " is moving at
    " + c.speed + " kilometers per hour.");
    c.floorIt();
    System.out.println(c.numberPlate + " is moving at
    " + c.speed + " kilometers per hour.");
  } // main()
} // class CarTest2
```

Invoking Instance Methods

- The output is:

```
P 123 XYZ is moving at 0.0 kilometers per hour.
P 123 XYZ is moving at 123.45 kilometers per hour.
```

- The `floorIt()` method is completely enclosed within the Car class. Every method in a Java program must belong to a class.
- Unlike C++ and other languages programs in Java cannot have a method hanging around in global space.
- Within the Car class, you don't need to prefix the field names with `this`. Just numberPlate is sufficient rather than `this.numberPlate`. The `this.` may be implied.

Static and non-Static members of a Class

- A class can only contain definitions for :
  - Instance methods
  - Instance fields
  - Static methods
  - Static fields
- Instance methods and instance fields are part of every instantiated object.
- Static methods and fields are not parts of the objects that get instantiated from a class.
- This is reflected to the mechanism of invoking static methods or accessing static fields.

Accessing static fields and methods of a Class

- To access a static member of a class from a different class you need to precede the reference to it with the class name in which it resides.
- The same happens with static methods.

```
public class Circle {
    public static final float PI = 3.14f;
    public static float getPI() { return PI; }
}

public class Example {
    public static void main(String[] argv) {
        System.out.println("PI" + Circle.PI);
        System.out.println("PI" + Circle.getPI());
    }
}
```

Access restrictions on fields

- It's generally a bad idea to access fields directly.Instead it is considered good object oriented practice to access the fields only through methods.
- This allows you to change the implementation of a class without changing its interface towards its clients.
- This also allows you to enforce constraints on the values of the fields.
- To do this you need to be able to send information into the Car class.
- This is done by creating suitable (access) methods and passing arguments to them.
- For example, to allow other objects to change the value of the speed field in a Car object, the Car class could provide an `accelerate()` method.
- This method does not allow the car to exceed its maximum speed, or to go slower than 0 kph.

Access restrictions on fields through methods

```
public void accelerate(double deltaV) {
    speed = speed + deltaV;
    if ( speed > maxSpeed ) {
        speed = maxSpeed;
    }
    if ( speed < 0.0 ) {
        speed = 0.0;
    }
} // end accelarate()
```

- The first line of the method is known as its signature. The signature
  `void accelerate(double deltaV)`
  indicates that `accelerate()` returns no value and takes a single argument, a double which will be referred to as deltaV inside the method.

Constructor Methods

- A constructor method is called on creation of a new instance of the class. It initialises all the member variables and does any work necessary to prepare the class to be used.
- In the line `Car c = new Car();` Car() is the constructor.
- A constructor has the same name as the class.
- If no constructor exists Java provides a default one that takes no arguments.
- You make a constructor by writing a method that has the same name as the class.
- Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.

Constructors

- The following method is a constructor that initialises numberPlate to an empty string, speed to zero, and maximum speed to 100.0.

```
      public Car() {
         numberPlate = "";
         speed = 0.0;
         maxSpeed = 100.0;
      }
```

- Better yet, you can create a constructor that accepts three arguments and use those to initialise the fields as below.

```
public Car(String numberPlate, double speed, double maxSpeed)
{
   this.numberPlate = numberPlate;
   this.speed = speed;
   this.maxSpeed = maxSpeed;
}
```

- Better still, you can use both methods since overloading works with constructors as with any method.

Constructors

- You can call a constructor from whithin a constructor using this.

```
class Car {
     //field declarations
    ……
    public Car() {
       numberPlate = "";
       speed = 0.0;
       maxSpeed = 100.0;
    }
    public Car(Color c) {
       color = c;
       this();
    }
    public Car(int r, int g, int b) {
       this(new Color(r,g,b));
    }
} //What about a private constructor?
```

Access Protection

- Variables that can be accessed from anywhere are a classic source of bugs in most programming languages. Some function can change the value of a variable when the programmer isn't expecting it to change. This plays all sorts of havoc.
- Most OOP languages including Java allow you to protect variables from external modification. This allows you to guarantee that your class remains consistent with what you think it is -as long as the methods of the class themselves are bug-free-.
- For example, inside the Car class we'd like to make sure that no block of code in some other class is allowed to make the speed greater than the maximum speed.
- We want a way to make the following illegal:

```
Car c = new Car("P 123 XYZ", 100.0);
c.speed = 150.0; // Illegal
```

- This code violates the constraints we've placed on the class. We want to allow the compiler to enforce these constraints.
- To achieve this we can specify who will be allowed to access which parts of the class.

Examples of Access Protection

- This is how the Car class would be written in practice. Notice that all the fields are now declared private, and they are accessed only through public methods. This is the normal pattern for all but the simplest classes.

```
public class Car {
  private String numberPlate;// e.g. "New York A456 324"
  private double speed; // kilometers per hour
  private double maxSpeed;     // kilometers per hour
   .........
```

Examples of Access Protection

```
// constructor
public Car(String numberPlate, double maxSpeed) {
   this.numberPlate = numberPlate;
   this.speed = 0.0;
   if (maxSpeed >= 0.0) {
        this.maxSpeed = maxSpeed;
   } else {
        maxSpeed = 0.0;
   }
} // constructor

// getter (accessor) methods
public String getNumberPlate() { return numberPlate; }
public double getMaxSpeed() { return speed; }
public double getSpeed() { return maxSpeed; }
```

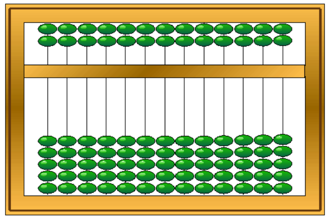
Examples of Access Protection

```
  // setter method for the number plate property
  public void setNumberPlate(String numberPlate) {
    this.numberPlate = numberPlate;
  }
  // accelerate to maximum speed
  public void floorIt() {
    speed = maxSpeed;
  } // floorit()
  public void accelerate(double deltaV) {
    speed = speed + deltaV;
    if ( speed > maxSpeed ) {
        speed = maxSpeed;
    }
    if ( speed < 0.0 ) {
        speed = 0.0;
    }
  } // end accelerate()
} // end class Car
```

The Three Benefits of Access Protection

- Access protection has three main benefits:
  - It allows you to enforce constraints on an object's state.
  - It provides a simpler client interface. Client programmers don't need to know everything that's in the class, only the public parts.
  - It separates interface from implementation, allowing them to vary independently. For instance consider making the numberPlate field of Car an instance of a new NumberPlate class instead of a String.

The Counter Class Example



- The class is a concept which can be used to introduce new Java types for our programs. As our first example we will see a generic Counter which counts the number of occurrences of some event in a program.
- Effectively, a Counter is an object which starts with an initial value of zero, and whose value can be increased one step at a time. At any point the value can be read by the program, tested whether it is zero or compared against another counter. The value can also be reset to zero at any time.
- Our goal is to build a general Counter class that can be used in many different programs. Bear in mind that there are two kinds of people involved with the Counter. Implementors, who design and build it, and its user(s) who use it in their programs. We shall look at each role in turn. Overall, there are three stages to building the class.
  - 1. Definition - the implementors (and maybe users) decide what the class should do and define its interface.
  - 2. Implementation - the implementors implement the class.
  - 3. Use - users use the class in their programs.
  - Stages 2 and 3 can occur in parallel.

Definition of a Counter

- Definition of the Counter means exactly specifying what services it provides and how it behaves.

```
public class Counter {
  private int count;               // stores the current count value

  public Counter();                // initialises counter to 0
  public void reset();             // resets counter to 0
  public int read();               // returns current count
  public void inc();               // increments count
  public boolean iszero();         // test whether 0
  public boolean equals(Counter c);  // test equivalence
}
```

Definition of Counter

- The public part defines the interface to the class in terms of a set of method declarations. Each declaration gives the name of a method along with its argument and result types.
- This list of methods define all the actions that users can take on variables of the class. Thus, the `reset` method simply sets the value to zero (hence no argument or result); the `read` method returns the current value as an integer; the `inc` method adds one to the current value; the `iszero` method tests whether the current value is 0 and the `equals` method tests against another counter (hence one argument of type Counter).
- The last two methods return boolean results.
- Note that the constructor does not return anything (common hard-to-find error :void on constructors)

Definition of Counter

- Good comments are an essential guide to how to use the class.
- The private fields define the data that is needed to implement the class. In the case of the Counter we need only to remember the current count value.
- This requires a single integer variable. Being modified as private means that users of the class are prohibited from having direct access. Private members can only be accessed by the class that defines them.
- It is quite common to see private methods (which can only be used by other methods). It is very rare (and probably very bad practice) to see public data members.
- It is a convention that the private section is written before the public section.

Implementation of the Counter Class

- Now let's play the role of the implementer. Once it has been defined, the Counter class needs to be implemented. This means implementing all of the methods specified in its definition.

```
public class Counter {
  private int count;

  // construct and initialise to 0
  public Counter() { count = 0; }
  // reset data member count to 0
  public void reset() { count = 0; }
  // return current value of data member count
  public int read() { return count; }
  // increment the value of the data member count
  public void inc() { count++; }

  // test whether the value of the data member count is zero
  public boolean iszero() { return (count == 0) }
  // test whether the value of count is equal to that of c
  public boolean equals(Counter c) { return (count == c.
  count) }
}
```

Implementation of the Counter Class

- Note the following points:
- There is no method `main()` so this is not a complete program.
- Each method can refer directly to private data members of the class. Hence the statement `count = 0`; in the reset method.
- The methods can also access the data members of other objects of the same Class. To do this they specify the object name. Thus the equals method compares its own data member against that of its argument "c" via the statement :

```
    if (count == c.count)
```

Using the Counter Class

- Now let's assume the users perspective. The user wants to employ the class to solve a specific problem. The following program reads in a series of words a word at a time. It counts the number of occurrences of the letter e (lower or uppercase) as the first letter of the word and prints out the result.

```
// test  the counter class
// count all occurrences of the letter 'e' in a sentence
public TestCounter {
   public static void main(String[] argv) {
        Counter myCounter; char c;
        while ((c = G51OOPInput.readChar())!=‘.’) {
              if (c==‘e’ || c==‘E’) myCounter.inc();
        }
    System.out.println(“There were “ + myCounter.read());
   }
```

- `Counter myCounter;` declares an object (variable) of Class (type) Counter, called `myCounter`. This statement automatically calls the constructor method which initialises its internal data member to zero.

Using the Counter Class

- Inside the while loop, we see the statement `myCounter.inc()`. This says invoke the `inc()` method on the object count. Its effect is to increase the internal value of count by one.
- The statement `myCounter.read()` invokes the `read()` method on `myCounter` which returns its current value ready to be printed out.
- Notice how easy it is to use the Counter class. Notice also how the user is oblivious of its internal implementation (and also how they cannot tamper with internal details!).
- The program doesn't use the equals method. However, the following fragment of code shows it might be used.

```
Counter a=new Counter(),b=new Counter();
if ( a.equals(b) )
  System.out.println("a & b contain the same value");
```

- Notice how `equals`, a binary operation between two objects, is expressed; it invokes the method ON one object and passes the other as an argument. This may be a little confusing at first and is important to take some time to understand.

Second example - the Money class

- Our second example is rather more complex and will highlight some of the issues involved in designing good classes. Our goal is to define and implement a general purpose Money class which can be used to manipulate pounds and pence values. This class might be useful for a variety of applications.

Defining the Money class

- Bearing in mind that we want a generally useful class, our biggest problem is how to design the right interface. Unfortunately there is no easy formula to be applied. Good class design is a matter of experience and trial-and-error.
- Design is an iterative process. This means that there might be many revisions before the definition is satisfactory.
- First, think what are the general properties of the Money class? Well, it stores pounds and pence values. You can add and subtract money. Subtraction implies that we can have negative values. You can compare money values (<, >, == etc). You might want to change an overall value or just inspect the pounds or pence components. You might want to print out money values in a special format.

Defining the Money class

- Think about methods which fall into five general categories: -

  1. Constructors - how should objects of the class be initialised?

  2. Combination - methods which combine existing objects into new ones (e.g. addition).

  3. Access - methods which return of alter internal data members of an object.

  4. Tests - methods which test or compare objects in some way.

  5. Input/Output - methods which deal with input and output of objects. Should Money be responsible for input/output? Depends on design and usage.

Defining the Money Class

- After several iterations :

  1. Constructors: initialise to given pounds and pence values.

  2. Combination: add, subtract.

  3. Access: negate, return pounds, return pence, set pounds and pence, increase and decrease by another Money value.

  4. Tests: ==, <, >, <=, >=, !=, is it zero?, is it positive or negative?

  5. None in this design

- These translate into the following Java class definition :

Defining the Money class 1/3

// Java class definition for money objects, adapted by azt’ 2000, original by sdb’ 1991

```
class Money {
   private int fpence; // total number of pennies

   // 1. constructors
   Money(int pounds, int pence);    // construct with supplied values
                                    // 0 <= pence <= 99 and 0 <= pounds

   // 2. combination methods

   // add two money values
   public Money add(Money m);

   // subtract two money values
   public Money subtract(Money m);
```

Defining the Money class 2/3

```
// 3. access methods
public void negate(); // change sign of value (+ve <-> -ve)

public int pounds(); // return number of pounds

public int pence(); // return number of pence

public void set(int pounds, int pence); // set to new value

public void increase(Money m); // increase current value by m

public void decrease(Money m); // decrease current value by m
```

Defining the Money class 3/3

```
// 4. testing methods
   boolean equals(Money m); // equal to?

   boolean less(Money m);    // less than?

   boolean greater(Money m);  // greater than?

   boolean less_equals(Money m);  // less than or equal to?

   boolean greater_equals(Money m);  // greater than or equal to?

   boolean is_zero();  // has zero value?

   boolean is_positive();  // is the value +ve or –ve

}
```

Defining the Money class

- A design decision had to be taken to model negative Money values. At first, It seems that the user can supply negative values of pounds in the constructor (e.g. Money a(-6,40)). However, initialising a sum such as -40 pence would be impossible this way because the computer interprets -0 as 0. One solution, sees the constructor building only positive values and forcing the user to subsequently use the negate() method. This is clumsy but consistent.

Implementing the Money class

- There are several possible implementations of the money class.
- We could use two internal integer data members called pounds and pence. Each method would then manipulate these as appropriate. For example, adding would involve adding the pounds and pence values from two objects.
- However, because the pence data member should have a maximum value of 99, this would involve more elaborate calculations in all combination and test operations (particularly for subtraction involving negative values).
- A second approach is to use only one data member to represent the total number of pence. This simplifies most calculations. The only extra work would be in the `pounds()` and `pence()` methods (and `print()` if provided). This kind of trade off is common to a variety of problems.

Implementation of the Money class 1/6

```
public class Money {
private int pnce;

// constructor - check and initialise values
// NOTE: can only initialise to positive value
// negative values must be obtained via the negate method
public Money(int pounds, int pence) {
    // check for pounds out of range
    if ( (pounds < 0) || ( (pence < 0) || (pence > 99) ) ) {
            System.out.println( "Error initialising Money: pounds out of range”);
    }
    // set data value to total number of pennies
    pnse = (pounds * 100) + pence;
}
```

Implementation of the Money class 2/6

```
// add two Money objects
public Money add(Money m) {
    // declare result object
    Money res = new Money(0,0);
    res.pnce = pnce + m.pnce;
    return res;
}
// subtract two money objects
public Money subtract(Money m) {
    // declare result object
    Money res = new Money(0,0);
    res.pnce = pnce - m.pnce;
    return res;
}
// negate current value
public void negate() { pnce = -pnce; }
```

Implementation of the Money class 3/6

```
// return number of pounds (absolute value)
public int pounds() { return Math.abs(pnce / 100); }

// return number of pence (absolute value)
public int pence() { return abs(pnce % 100); }

// is the value positive
public boolean is_positive() { return (pnce >= 0); }

// set to a new value
public void set(int pounds, int pence) {
    // check for pounds and pence out of range
    if ((pounds < 0) || ( (pence < 0) || (pence > 99) )) {
            System.out.println( "Error initialising Money: pounds out of range“);
    }
    // set data value to total number of pennies
    pnce = (pounds * 100) + pence;
}
```

Implementation of the Money class 4/6

```
// increase current value
public void increase(Money m) { pnce = pnce + m.pnce; }

//decrease current value
public void decrease(Money m) { pnce = pnce - m.pnce; }

// are two values equal?
public boolean equals(Money m) { return (pnce == m.pnce);}

// is one value less than another?
public boolean less(Money m) { return (pnce < m.pnce); }
```

Implementation of the Money class 5/6

```
    // is one value greater than another?
    public boolean greater(Money m) { return (pnce > m.pnce) }

    // is one value less than or equal to another?
    public boolean less_equals(Money m) { return ( pnce <= m.pnce); }

    // is one value greater than or equal to another?
    public boolean greater_equals(Money m) { return (pnce >= m.pnce); }

    // is a money value zero
    public boolean is_zero() { return (pnce == 0); }

} // end Money class
```

Note the use of the Math.abs method to get absolute (i.e. positive) values and the remainder (%) operation in the print() and pence() methods

Implementation of the Money class 6/6

Note a print a money value method might look like this….

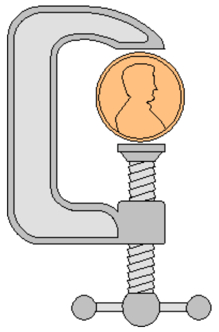
```
public void print() {
     int p;
     // display sign and money symbol
     if (!is_positive())
          System.out.println( "- “);
     // calculate and print pounds and pence values
               p = Math.abs(pnce % 100);
      System.out.print( Math.abs(pnce / 100) + “.”+ p );

 }
```

Testing the Money class 1/2

The following user program shows how Money objects might be built, added and subtracted. The program does nothing useful other than show the syntax of manipulating Money objects.

```
public static void main(String[] argv) {
    int Pounds, Pence; char ch;          // read in first money value
    System.out.print(“enter a - pounds: “);
    Pounds = G51OOPInput.readInt();
    System.out. print("pence: “);
    Pence= G51OOPInput.readInt();
    Money a= new Money(Pounds, Pence);
    System.out. print("negate? “);
    if (G51OOPInput.readChar() == 'y')
            a.negate();
```



Testing the Money class 2/2

```
    // read in second money value
    System.out. print(" enter b - pounds: “);
    Pounds = G51OOPInput.readInt();
    System.out. print("pence: “);
    Pence= G51OOPInput.readInt();
    Money b= new Money(Pounds, Pence);
    System.out. print("negate? “);
    if (G51OOPInput.readChar() == 'y')
            b.negate();
    Money c= new Money(0, 0);
    System.out. print("a + b is “);
    c = a.add(b); c.print();                    // test add method
    System.out. print("a - b is “);
    c = a.subtract(b); c.print();               // test subtract method
    if (a.less_equals(b))                       // test a comparison operator
            System.out. print("a <= b\n“);
    else System.out. print(“a > b\n“);
}
```