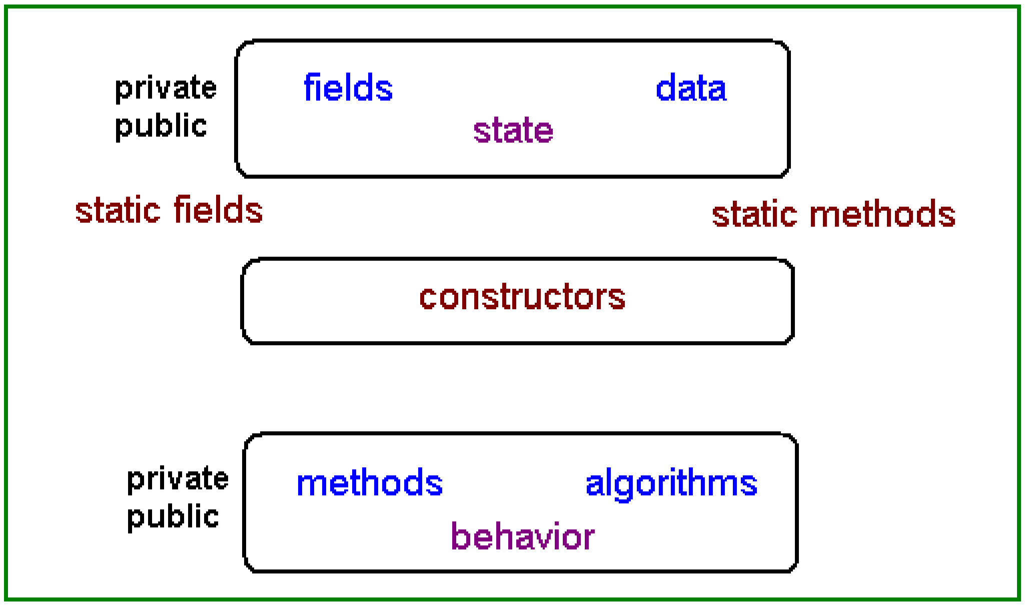
Class Design

Introduction

Consider a car manufacturing. A motor giant BMW does not make all parts for its new model of a car. Various pieces (wheels, doors, seats, breaks, sparks...) come from different manufacturers. This model keeps the cost down and let BMW response to the market needs in the most efficient way. The main BMW's responsibility is to design a car, and then to assemble it from already existent parts (a small amount of completely new details might be necessary). Object-oriented programming (OOP) springs from the same idea of using preassembled modules. The OOP provides the programmer with a natural way to divide an application into small reusable pieces, called objects. At the same time, the OOP provides an elegant way to construct applications in more efficient way by building up from a collection of reusable components.

All software objects have *state* and *behavior*. For example, a student as an object has a name, address, list of courses, and grades. All these are a state (or data). On the other hand, the student has behaviors (or methods), such as adding a new course, changing the address. Therefore,each object is a collection of data and methods (or algorithms) applied to its internal data.



An object should never allow an external manipulation (the client access) over the internal data or expose data to other objects. Why not give the client direct access to the fileds? Because this will create problems in the long run. For example, if you change the name of theat field, the client will no longer work. Also, the object should never expose details of an algorithm implementation. Clients need only know what your methods do, not how you implement them. In short,

- Data is private
- Implementation is hidden

Encapsulation is the idea of hiding data and methods implementations from the client. By encapsulation you reduce data dependency and maximize reusability.

How do we describe the objects? By designing *classes*. Think of a class as a template for objects. We can create several objects from one class. Such process is called an *instantiation*. An object, in effect, is an ainstantiated class. There are three ways to design classes: by *composition*, via *inheritance*, and via *interface*.

Composition (or aggregation) is achieved by using existing class as a part of a new class. For example, the ArrayStack class includes an array of objects.

Inheritance allows you to define a new class in terms of an old class. The new class automatically inherits all public members of the original class. Inheritance is useful for creating specialized objects that share comon behavior. We will see more on inheritance later in the course.

Interfaces act like inheritance in the way that they define a set of properties, methods,and events, but unlike classes they do not provide implementation. A class that implements the interface muct implement every method of that interface exactly as it is defined. We will see more on interfaces later in the course.

Class Declaration

Freely speaking, a Java program is a set of packages, where each package is a set of classes. A class can contain any number of fields and methods - they are called *members* of a class. The implementation design might suggest using private and public members. The private methods mostly used as helper methods to assist in the implementation of public methods. The data in a class is typically private, therefore you must ensure that a class implements public methods to access (and maybe modify) the data. We will call public methods *getters* or *accessors* if they are designed to only access the data. We will call public methods *setters* or *mutators* if they allow to modification of the internal data.

Constructor

All Java classes have constructors (one ore more) that are used to allocate memory for the object and to initialize the data fields. Constructors are not methods, though they look like a method. There are three differences between constructors and methods:

1. A constructor has the same name as the class.
2. A constructor has no return value.
3. A constructor is called with the `new` operator when a class is instantiated or by using a special reference *this()*.

When writing your own class, you don't have to provide constructors for it. The *default constructor* is automatically provided. The default constructor doesn't do anything. However, if you want to perform some initialization, you will have to write one ore more constructors for your class.

Method Overloading

Java enables several methods of the same name to be defined as long as these methods have a different set of parameters: the number of parameters, the types of the parameters and the order of the parameters. This is called method overloading. When an overloaded method is called, the Java compiler selects the proper method by examining the number, types and order of the arguments in the call. Here is an example of legal declarations:

```
public void methodA()
public void methodA(int z)
public void methodA(int z, String str)
public void methodA(String str, int z)
```

Method overloading is commonly used to create several methods with the same name that perform similar tasks, but on different data types. A method's name along with the number, type and order of its parameters is called the method *signatures*. If you attempt to specify two methods with identical signature, the compiler issues an error message.

Scope

The scope of an identifier for a variable, reference or method is the portion of the program in which the identifier can be referenced. A local variable or reference declared in a block can be used only in that block or in blocks nested within that block. The scopes for an identifier are class scope and block scope. Methods and instance variables of a class have *class scope*. Class scope begins at the opening left brace, {, of the class definition and terminates at the closing right brace, }, of the class definition. Class scope enables methods of a class to blockquoteectly invoke all methods defined in that same class and to blockquoteectly access all instance variables defined in the class.

Identifiers declared inside a block have *block scope*. Block scope begins at the identifier's declaration and ends at the terminating right brace } of the block. Local variables of a method have block scope as do method parameters, which are also local variables of the method.

For example, the following code fragment

```
int i = 1;
for(int sum = 0; i < 2; i++) sum += i;

while(i < 3) {
   sum += i;
   i++;
}
```

will result in the error "cannot resolve symbol sum". Since the variable *sum* is defined inside the for-loop it can be referenced only within that scope.

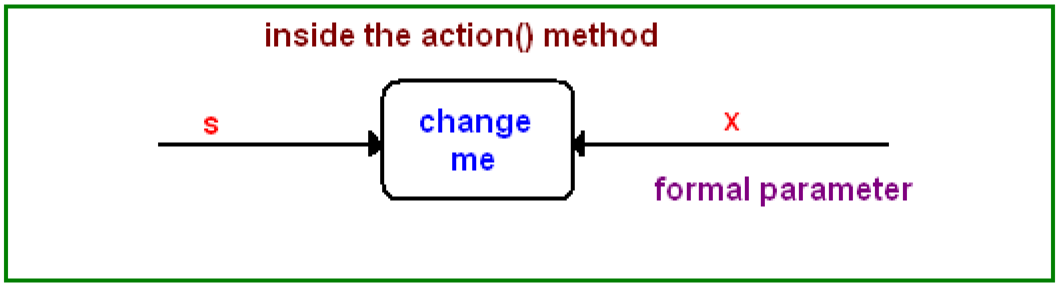
Pass by value

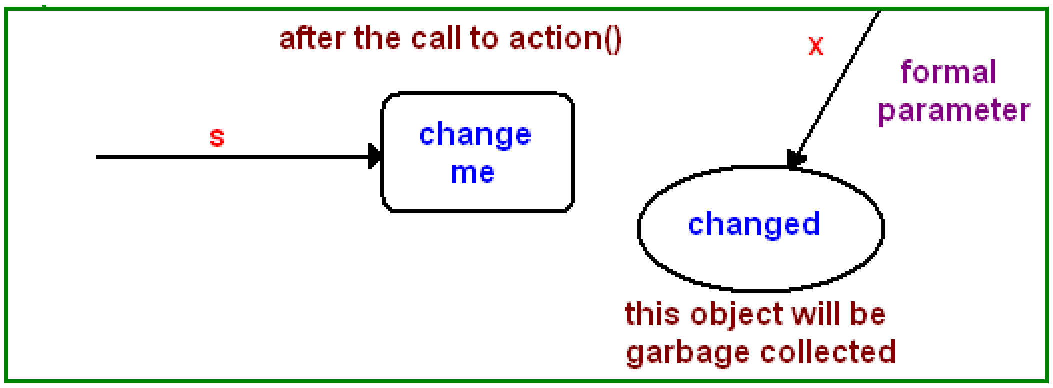
In Java objects are passed by value. Pass by value means that when an argument is passed to a method, the method receives a copy of the original value. In the example below, the string "change me" has two references **s** and **x** assigned to it.

```
public class Demo
{
   public static void main(String[] args)
   {
      String s = "change me";
      action(s);
      System.out.println(s);
   }

   public static void action(String x)
   {
      x = "changed";
   }
}
```

These two references live in different scopes. When we are back to the **main()** method, the formal parameter **x** from the **action()** method is garbag collected and the new object "changed" is destroyed. Therefore, the object "change me" passed into the method won't be actually changed. The picture below illustrates the internal mechanism.




after the call to action()
x
formal parameter
s
change me
changed
this object will be garbage collected

In other words, pass-by-value means that the method cannot change the whole object, but can invoke the object's methods and modify the parts within the object. If you want to actually change the whole object, you need to explicitly return it as in the code below

```
public class Demo
{
   public static void main(String[] args)
   {
      String s = "change me";
      s = action(s);
      System.out.println(s);
   }

   public static String action(String x)
   {
      x = "changed";
      return x;
   }
}
```

If the above data type was an array you would have an ability to internally modify the object. Consider the following code fragment:

```
public class Demo
{
   public static void main(String[] args)
   {
      int[] a = {1,2,3};
      action(a);
      System.out.println( Arrays.toString(a) );
   }

   public static void action(int[] x)
   {
      x[0] = 5;
   }
}
```

It demonstrates that the first element of the array has been changed.

Static vs. Non-Static

A class may contain *static* and *instance* members (fields and methods). The static members have a keyword `static` included in their declarations. They are called by using the class name.

Static members do not formally belong to an object; they exist before the object was created, Static members are created when you compile your program. In runtime, when you instantiate a class, only instance members are created. Moreover, every new object instantiated from the same class, has a new set of instance variables. In contrary, there is only one copy of static members, regardless how many object you created. Consider the following code fragment

```
   public class StaticDemo
   {
      public static void main(String[] args)
      {
         Demo obj1 = new Demo();
         Demo.number++;
         Demo obj2 = new Demo();
         System.out.println(obj2.getX());
      }
   }
   class Demo
   {
      private int x;
      static int number = 0;

      public Demo() {number++;}
      public int getX() {x = number; return x;}
   }
```

In this code example we created two objects ob1 and obj2 that share a static variable number. This variable was incremented three times: during instantiation of ob1, during instantiation of obj2 and by a direct call to it. The staic variable number leaves in a global context and can be invoked either using references obj1 and obj2 or using the class name Demo.

OO design principles

There are many heuristics associated with object oriented design. For example,

- all member variables should be private
- global variables should be avoided

In this chapter we briefly glimpse a design principle that is fundamental to these heuristics -- *the open-closed principle*. When a single change to a program results in a cascade of changes, the program becomes fragile and unpredictable. The open-closed principle says that you should design modules that never change.

Software modules should be open for Extension and closed for Modification.

To extend the behavior of the system, we do not modify old code that already works. - we add new code, How can we do this? Let us consider a code fragment. Given the Part class

```
public class Part
{
   private double price;

   public Part(double p)
   {
      price = p;
   }

   public double getPrice()
   {
      return price;
   }
}
```

and the totalPrice method of some other class:

```
public double totalPrice(Part[] parts)
{
   double total = 0.0
   for(int k = 0; k < parts.length; k++)
      total += parts[k].getPrice();

   return total
}
```

that totals the price of each part in the specified array of parts. The code looks quite innocent and you would not suspect a problem with it in the future. Now assume that the accounting department comes out with a new pricing policy for some parts, for example DVD_Drives are 20% off sale. To reflect this change we have to rewrite the totalPrice method

```
public double totalPrice(Part[] parts)
{
   double total = 0.0
   for(int k = 0; k < parts.lebgth; k++)
   {
      if(parts[k] instanceof DVD_Drive)
         total += 0.8 * parts[k].getPrice();
      else
         total += parts[k].getPrice();
   }

   return total
}
```

This is an example of bad design - every time the accounting department comes out with a new pricing policy, we have to modify the totalPrice method. The code is not closed for modification. What can we do? A better idea is not to fix the price but to introduce various pricing policies via a PricingPolicy class.

```
public class Part
{
   private double price;
   private PricingPolicy policy;

   public Part(double p)
   {
      price = p;
   }

   public double getPrice()
   {
      return price * policy.getPolicy();
   }

   public double setPricingPolicy(PricingPolicy newPolicy)
   {
      policy = newPolicy;
   }
}
```

With this solution we can dynamically set pricing policies for each modified part. The general idea is that we try to design and implement classes so that they use abstract classes and interfaces. That way, we can extend the functionality by simply adding new subclasses which add new behavior.

Victor S.Adamchik, CMU, 2009