In the Beginning...

• Goal: Build a search engine!
• What do we need?
  – Robot that crawls all web pages
  – A retrieval engine that finds the best matches for a query.
  – A web server that gets keyword queries from the user and presents search results.

→ Break problem down into modules.

Modularity

• Examples:
  – Tires in a car (standard size, many vendors)
  – External keyboard for computer
  – Course at Cornell
  – ...
• Delegate responsibility for individual modules

How does Java support modularity?

• Classes and Objects
  – Contain data
  – Contain methods for accessing data
  – Inheritance avoids duplication of effort
• Interfaces
  – Standardization across multiple classes
• Packages
  – Collections of classes and interfaces

Information Hiding

• Modules hide internal design decisions!
• Modules provide a well-defined external interface.

```
class Set {
  ...
  public void add(Object o) ...
  public boolean contains(Object o) ...
  public int size() ...
}
```

Encapsulation

• By hiding code and data behind its interface, a class encapsulates its “inner workings”
• Why is that good?
  • Lets us change the implementation later without invalidating the code that uses the class

```
class LineSegment {
  private Point2D _p1, _p2;
  ...
  public double length() {
    return _p1.distance(_p2);
  }
}
```
Encapsulation

• Why is that good? (continued)
  – Sometimes, we want a few different classes to implement some shared functionality
  – For example, the “iterator” construct:

    ```java
    Iterator it = collection.iterator();
    while (it.hasNext()) {
        Object next = it.next();
        doSomething(next);
    }
    ```

    Ensures there are methods hasNext(), next()...

    Can be list, set, tree, ...

    • To support iteration, a class simply needs to implement the **Iterable** interface

Degenerate Interfaces

• Public fields are usually a **Bad Thing**:

  ```java
  class Set {
      public int _count = 0;
      public void add(Object o) ... 
      public boolean contains(Object o) ... 
      public int size() ... 
  }
  ```

  • Anybody can change them; the class has no control

Interfaces vs. Implementations

• This says “I need this specific implementation”:

  ```java
  public void doSomething(LinkedList items) ... 
  ```

• This says “I can operate on anything that supports the **Iterable** interface”

  ```java
  public void doSomething(Iterable items) ... 
  ```

• Interfaces represent higher levels of abstraction (they focus on “what” and leave out the “how”)

Use of encapsulation and interfaces?

• Support of team work and modularity!
  – Rebecca agrees to implement web robot
  – Tom will implements the ranking algorithm
  – Willy is responsible for the user interface

  → By agreeing on the interfaces between their respective modules, they can all work on the program simultaneously

• Can use work of others (later) without having to understand internals!
  – Faster development of code.
  – Use of components that are already well tested.