COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

Admin: Lab overview

- Lab organization
- Website for lab manual
- Review guidelines
- Show list of labs
- Quick demo of the provided Feed Me game
- Pre-lab rules
- New multiple choice site demo
  - Bonus rule

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?
  - Classes and objects

what will be involved

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?
  - Classes and objects
  - Identifying attributes

what will be involved

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?
  - Classes and objects   what will be involved
  - Identifying attributes   what objects store
  - Class responsibilities

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?
  - Classes and objects what will be involved
  - Identifying attributes what objects store
  - Class responsibilities who interacts with whom
  - Encapsulation

Recap: Object-Oriented Design

- Process of building software based on a series of objects that interact together to solve a problem
- Object-oriented programming (OOP)
  - Set of programming techniques to support this design
- OOP examples from COSC 111?

– Classes and objects	what will be involved
– Identifying attributes	what objects store
– Class responsibilities	who interacts with whom
– Encapsulation	how they communicate

Class Relationships (Ch 7.4)

- Dependency ("Uses"): calls a method
  - A class uses another class
    - Ex: The Dog class uses the Scanner class
  - An object of one class uses another object of the same class
    - Ex: A Dog object shares snacks with another Dog
- Aggregation ("Has-A"):
  - A class has objects of another class
    - Ex: A Library has Book objects

Inheritance (Ch 9)

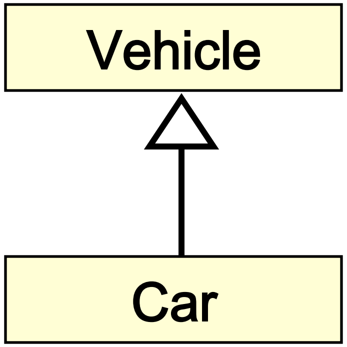
- Another OOP technique
- Purpose:
  - Organize “related” classes together
  - Maximize reusable classes
- What is reusability and its advantages?
  - Defined class once, don’t define it again
  - Defined methods once, don’t define them again
  - Changes isolated to one place
  - Bugs isolated to one place

Relationship

- Inheritance relates two classes to each other
- Conceptual examples:
  - Children inherit physical traits from their parents
  - Humans inherit biological traits from Animals
- Terminology:
  - A child class inherits from a parent class
  - A subclass inherits from a superclass
  - A child class is derived from a parent class
  - A subclass is derived from a superclass

Visually in UML

- Use a box to represent a class
- Use an upward arrow to point to the parent class



A car is a vehicle

- Depicts an IS-A relationship
- Text: each box has lots of details – ignore for this class

Benefits of Inheritance

- Inherit methods and attributes from parent class
- Can add new methods and attributes to child class
- Can modify inherited method definitions inside child class
- All to maximize software reusability

How it’s done

- Use a reserved word `extends` to indicate the relationship
- Template:

```
public class Child extends Parent
{
    // class contents
}
```

- Example:

```
public class Car extends Vehicle
{
    // class contents
}
```

Examples

- Partial code:

```
public class Animal { … }
public class Mammal extends Animal { … }
public class Reptile extends Animal { … }
public class Dog extends Mammal { … }
```

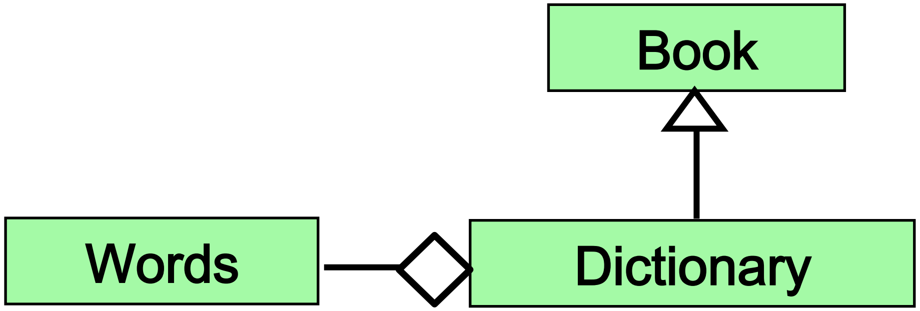
- List the *four* IS-A relationships that are defined by this code

Longer Example

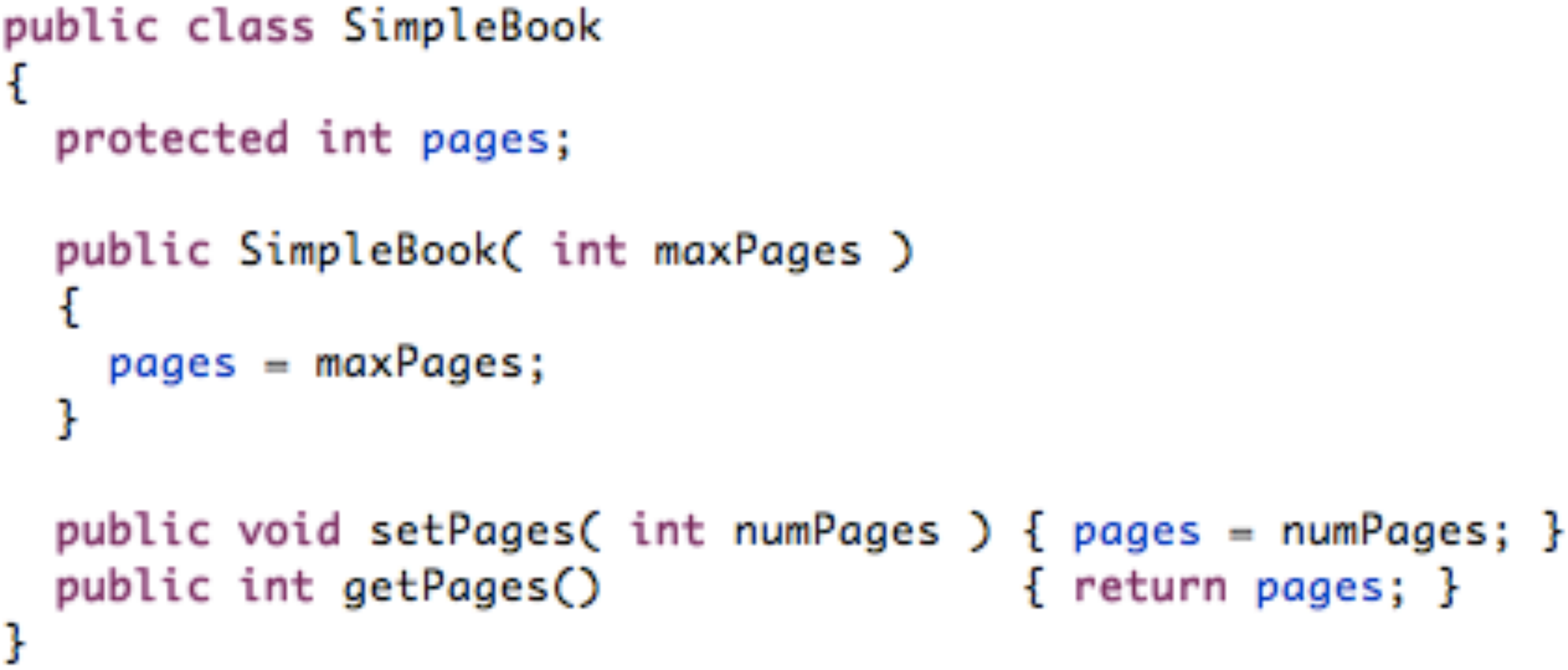
- Client says:
  - I want a software program that lets me look up word definitions easily. After that, I might also want to extend the program to give me more complicated entries, like an encyclopedia.
- What classes do we need to model?
- How are they related?

Longer Example (cont.)

- Sample solution:
  - A Dictionary is a Book
  - A Dictionary has Words

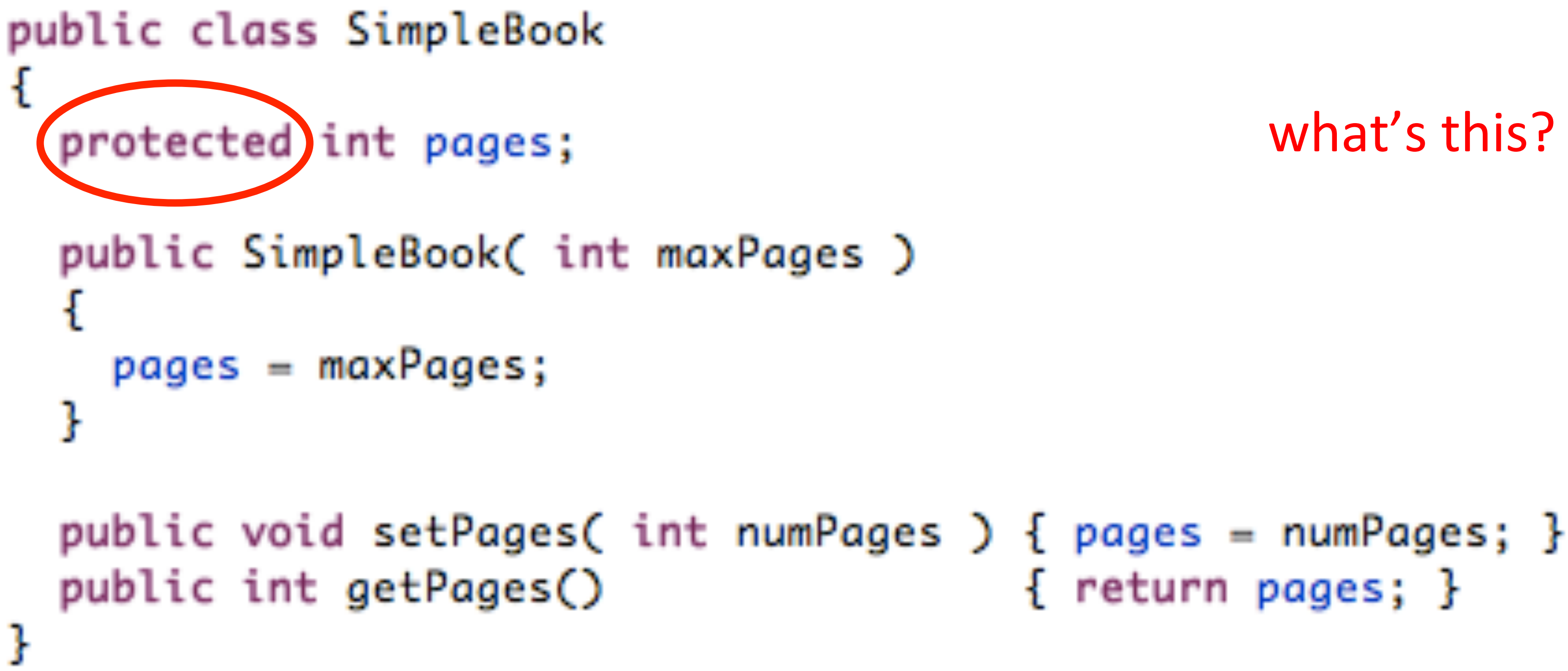


A Very Simple Book Class



```
public class SimpleBook
{
  protected int pages;

  public SimpleBook( int maxPages )
  {
    pages = maxPages;
  }

  public void setPages( int numPages ) { pages = numPages; }
  public int getPages()                { return pages; }
}
```

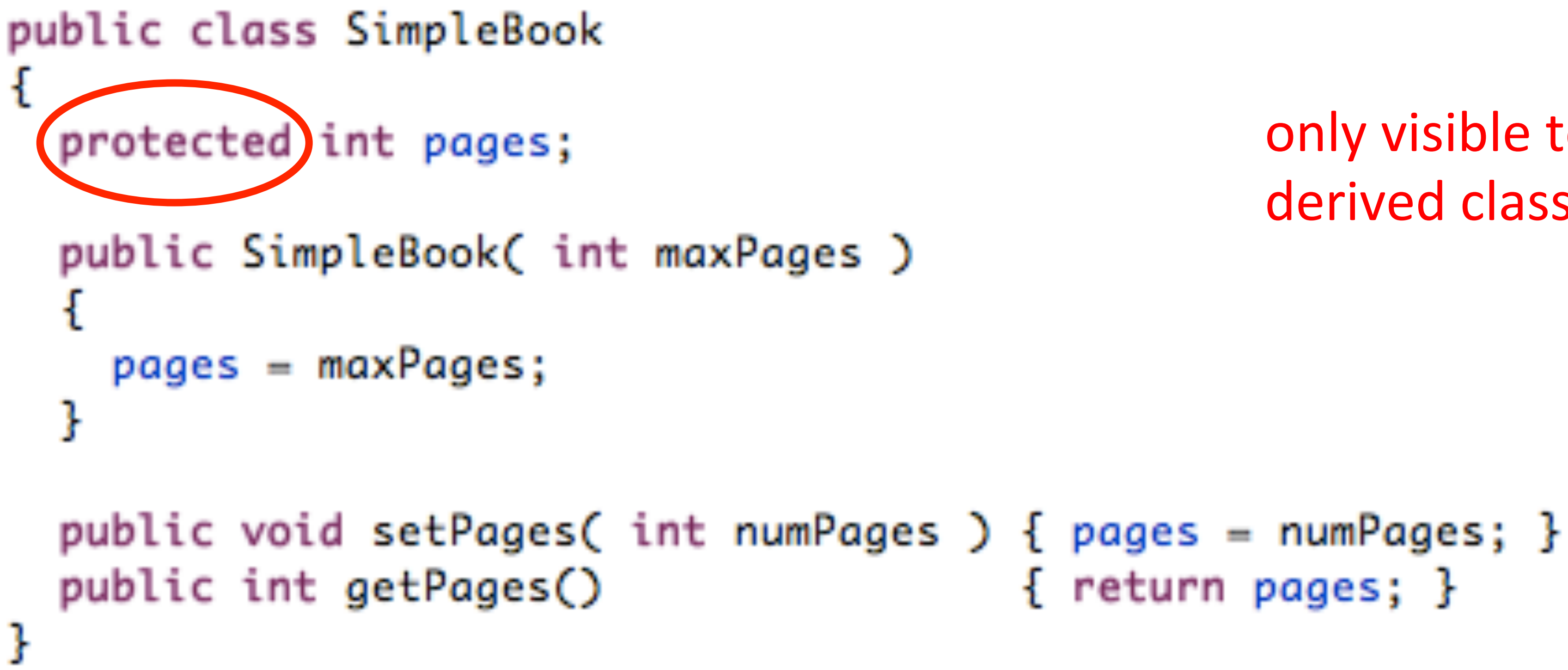
A Very Simple Book Class



```
public class SimpleBook
{
  protected int pages;

  public SimpleBook( int maxPages )
  {
    pages = maxPages;
  }

  public void setPages( int numPages ) { pages = numPages; }
  public int getPages()                { return pages; }
}
```

what’s this?

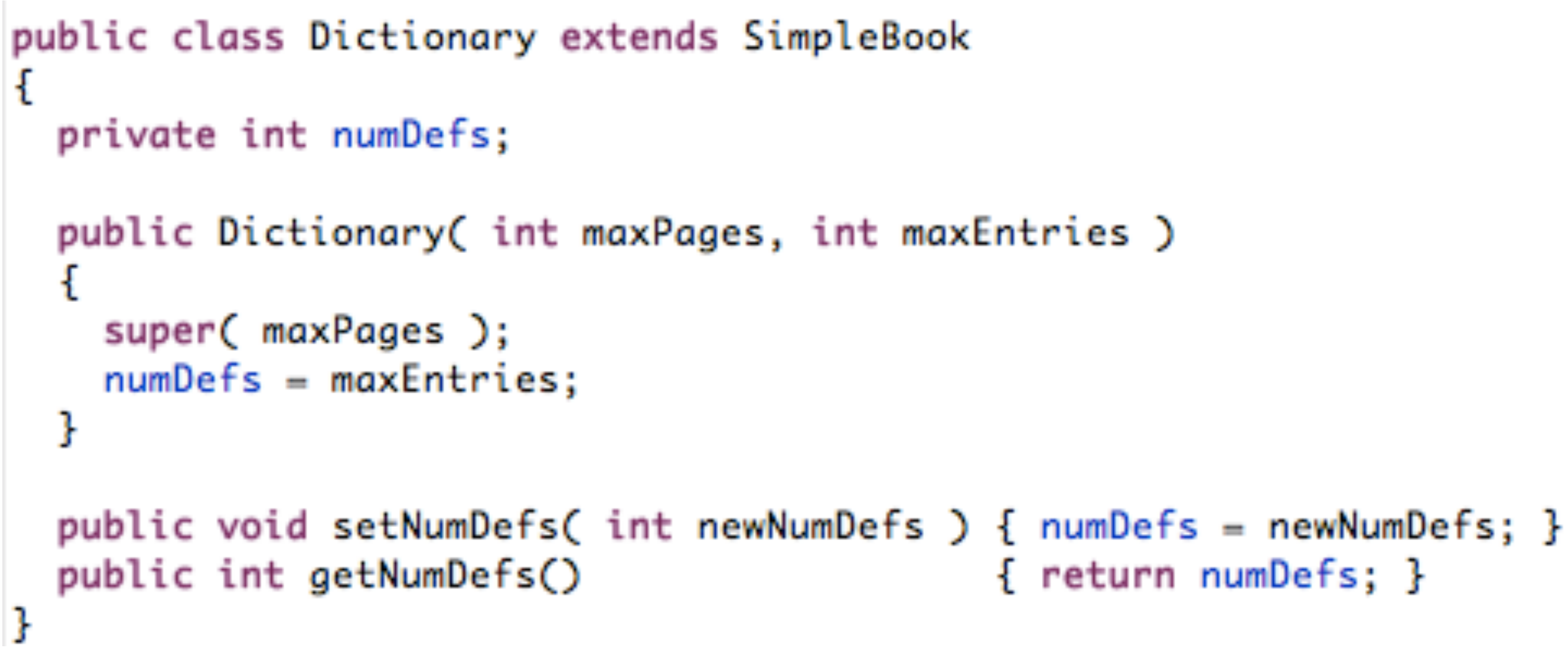
A Very Simple Book Class



```
public class SimpleBook
{
  protected int pages;

  public SimpleBook( int maxPages )
  {
    pages = maxPages;
  }

  public void setPages( int numPages ) { pages = numPages; }
  public int getPages()                { return pages; }
}
```

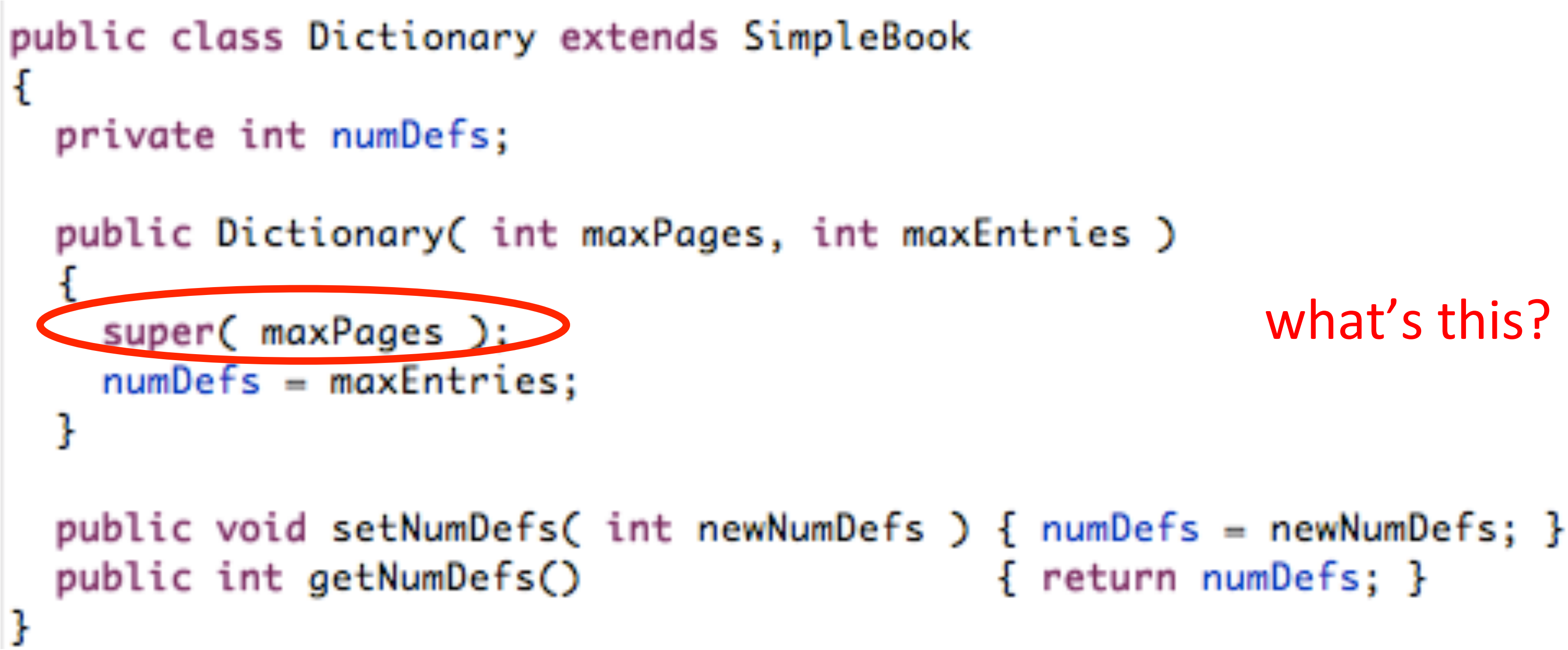
only visible to derived classes

An Initial Dictionary Subclass



```java
public class Dictionary extends SimpleBook
{
  private int numDefs;

  public Dictionary( int maxPages, int maxEntries )
  {
    super( maxPages );
    numDefs = maxEntries;
  }

  public void setNumDefs( int newNumDefs ) { numDefs = newNumDefs; }
  public int getNumDefs()                  { return numDefs; }
}
```

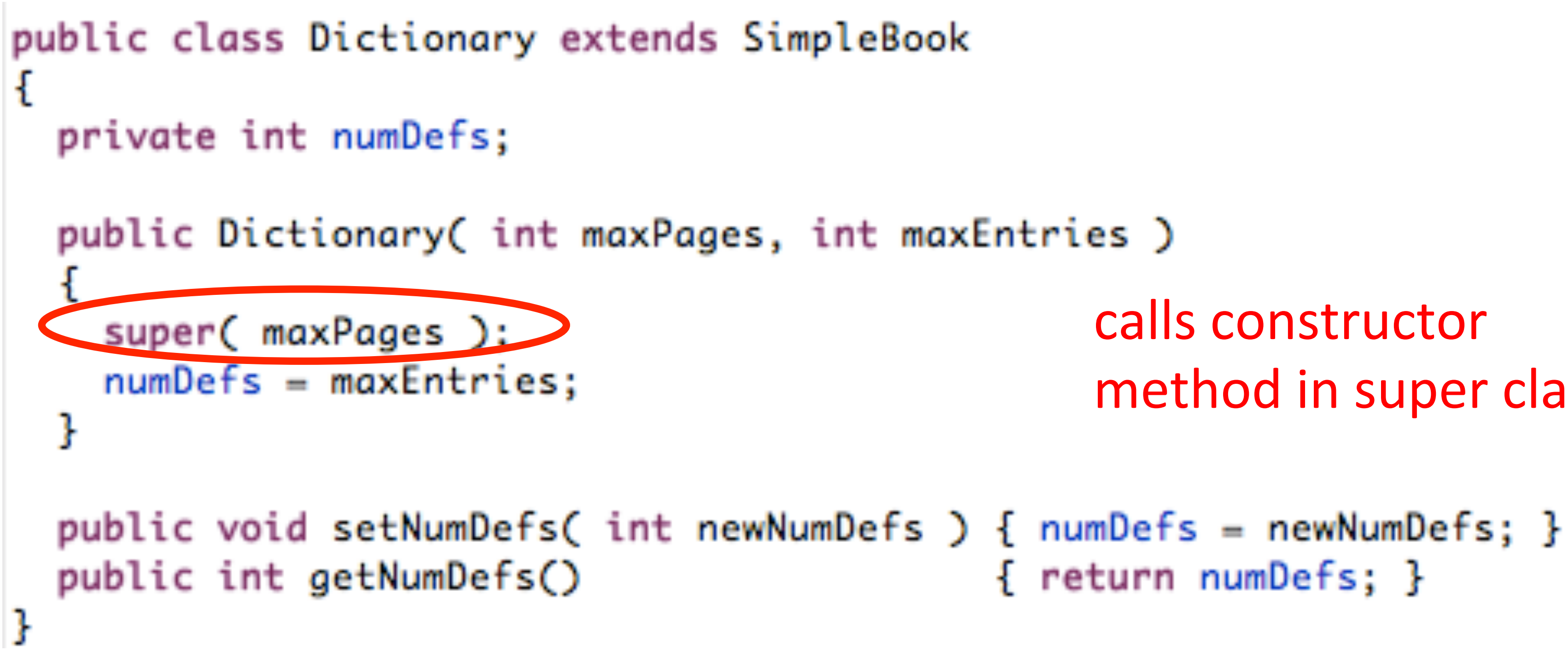
An Initial Dictionary Subclass



```
public class Dictionary extends SimpleBook
{
  private int numDefs;

  public Dictionary( int maxPages, int maxEntries )
  {
    super( maxPages );
    numDefs = maxEntries;
  }

  public void setNumDefs( int newNumDefs ) { numDefs = newNumDefs; }
  public int getNumDefs()                  { return numDefs; }
}
```

what's this?

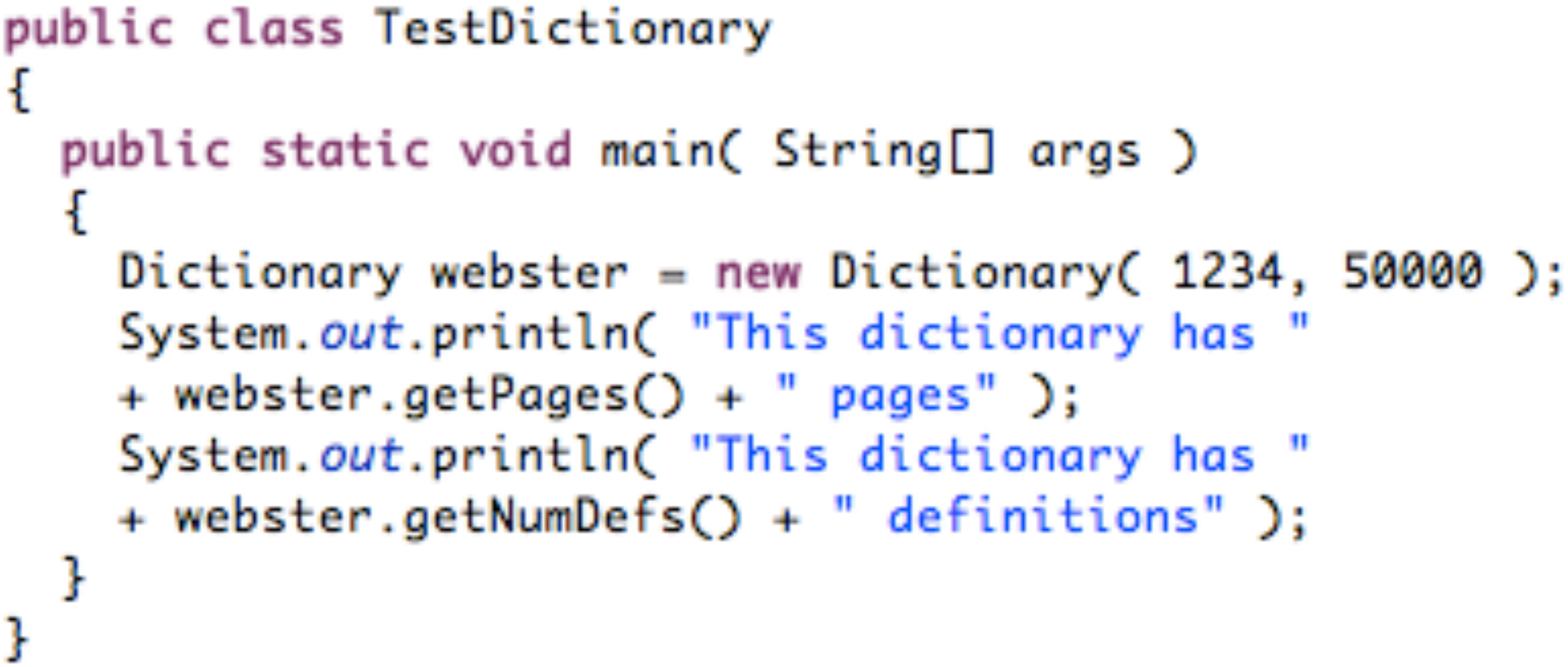
An Initial Dictionary Subclass



```java
public class Dictionary extends SimpleBook
{
  private int numDefs;

  public Dictionary( int maxPages, int maxEntries )
  {
    super( maxPages );
    numDefs = maxEntries;
  }

  public void setNumDefs( int newNumDefs ) { numDefs = newNumDefs; }
  public int getNumDefs()                  { return numDefs; }
}
```

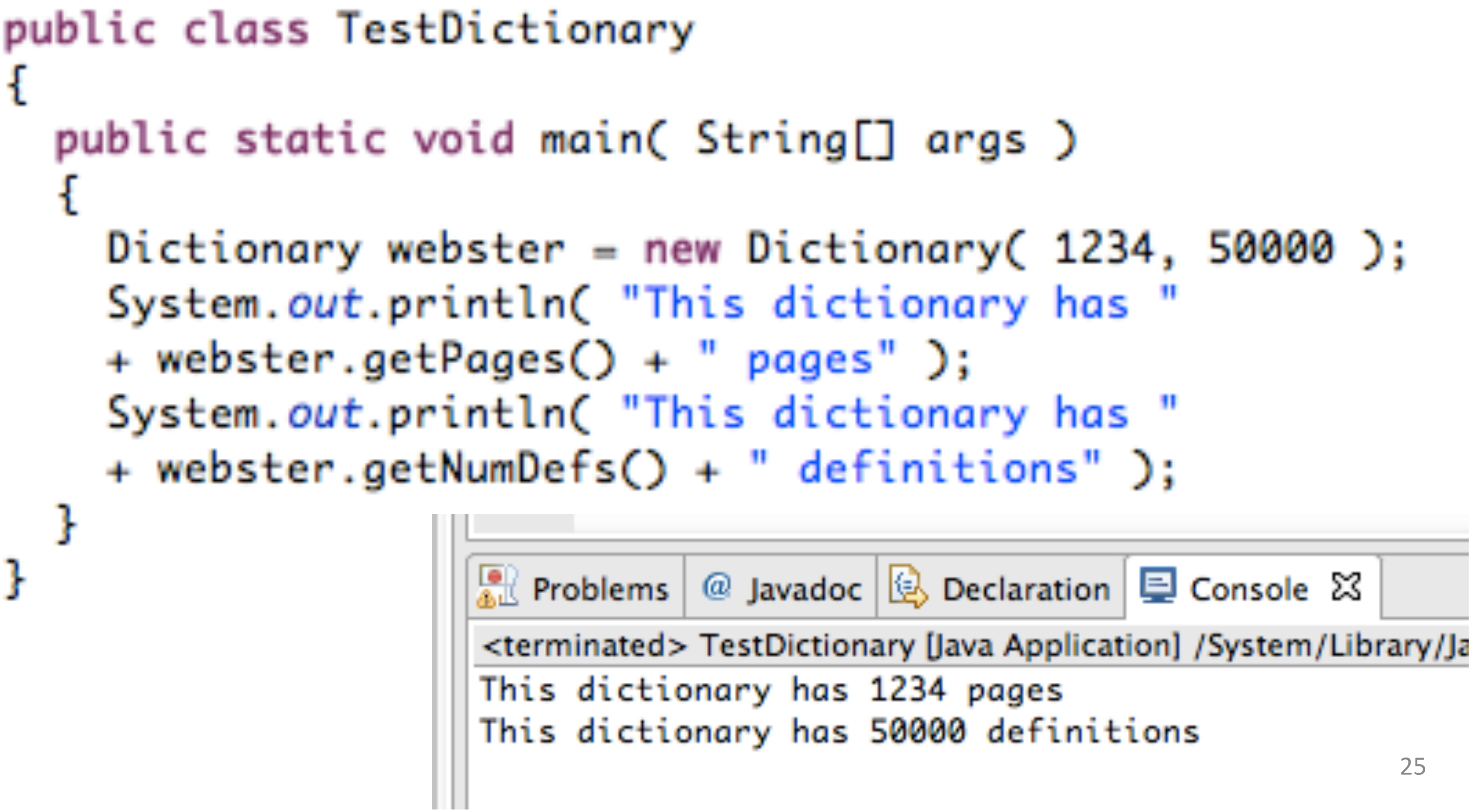
calls constructor method in super class

A Test Class



```
public class TestDictionary
{
  public static void main( String[] args )
  {
    Dictionary webster = new Dictionary( 1234, 50000 );
    System.out.println( "This dictionary has "
    + webster.getPages() + " pages" );
    System.out.println( "This dictionary has "
    + webster.getNumDefs() + " definitions" );
  }
}
```

A Test Class



```
public class TestDictionary
{
  public static void main( String[] args )
  {
    Dictionary webster = new Dictionary( 1234, 50000 );
    System.out.println( "This dictionary has "
    + webster.getPages() + " pages" );
    System.out.println( "This dictionary has "
    + webster.getNumDefs() + " definitions" );
  }
}
```

<terminated> TestDictionary [Java Application] /System/Library/Ja

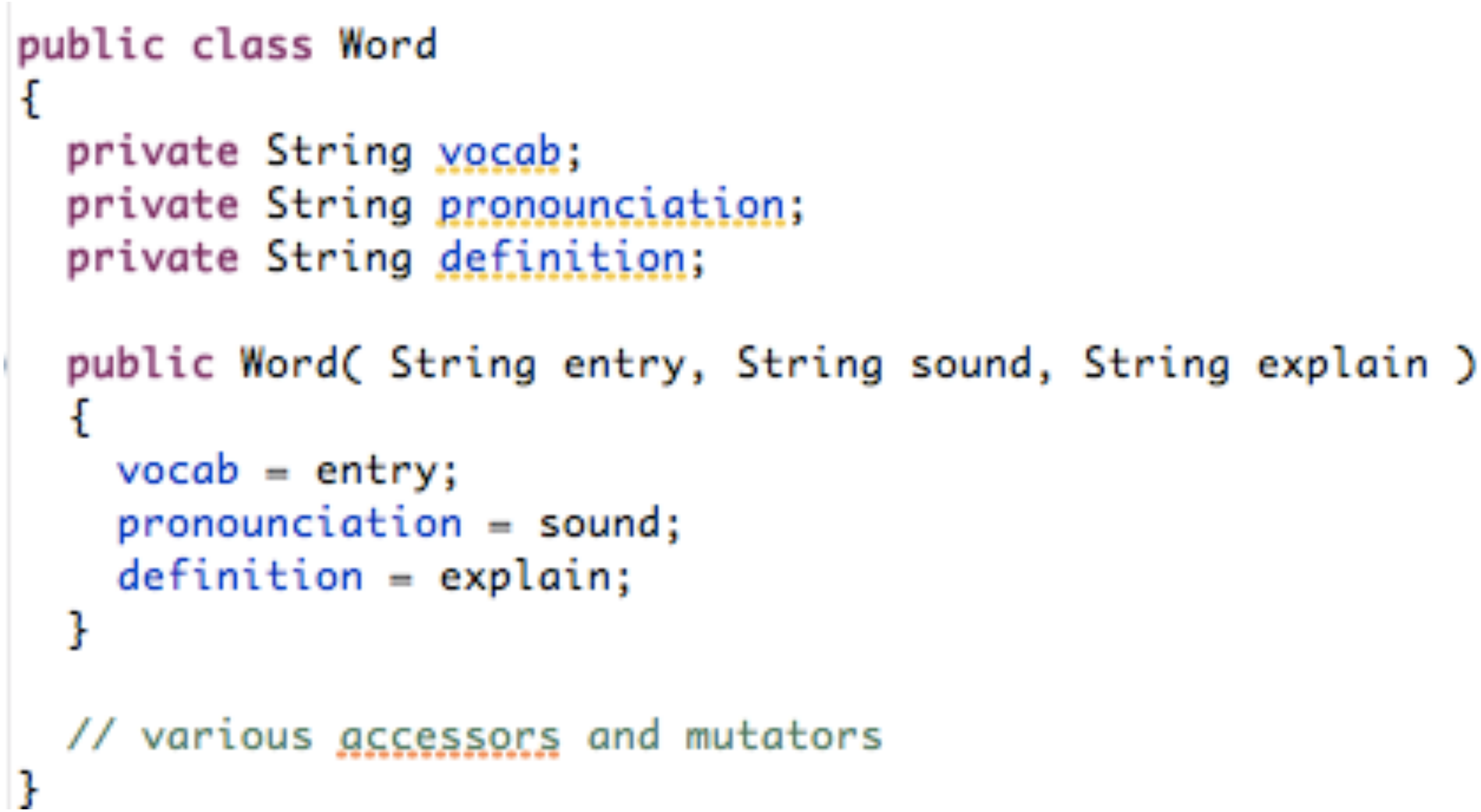
```
This dictionary has 1234 pages
This dictionary has 50000 definitions
```

What is inherited?

- All attributes from the parent class
  - Even private ones
  - How to access them? (See Section 9.4)

- All methods from the parent class
  - Except: constructors are not inherited
  - Why not?

What about Word?

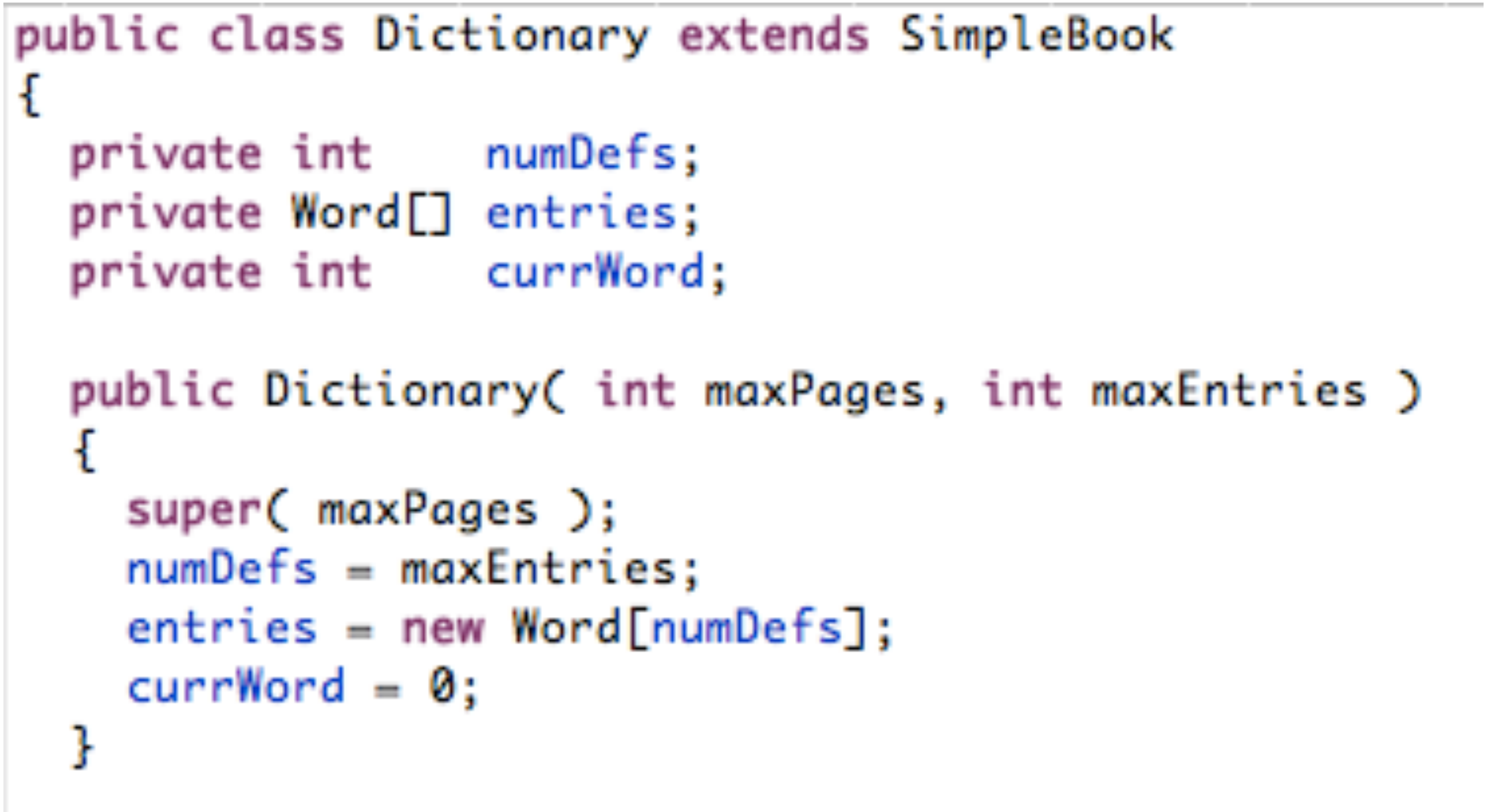


```
public class Word
{
  private String vocab;
  private String pronounciation;
  private String definition;

  public Word( String entry, String sound, String explain )
  {
    vocab = entry;
    pronounciation = sound;
    definition = explain;
  }

  // various accessors and mutators
}
```

Changing Dictionary Class

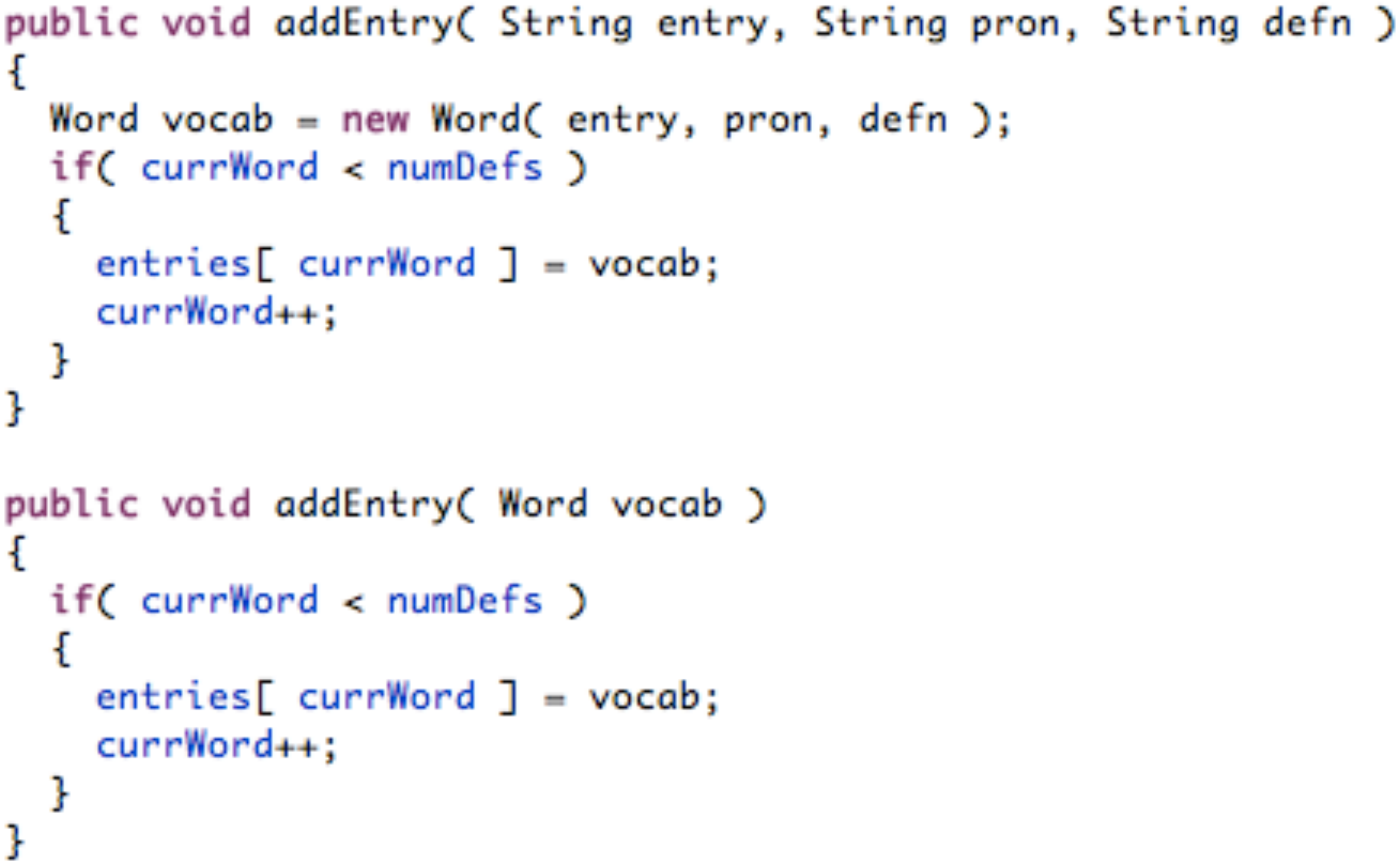
- How to keep track of Word objects?
- How to define a new method for `addEntry()`?
- What input parameters should it take?
- How to test your new changes in TestDictionary?

Sample Solution



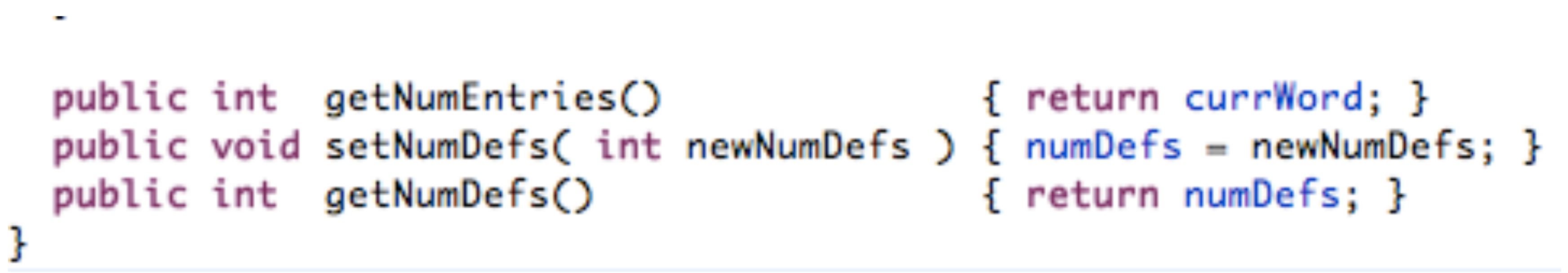
```
public class Dictionary extends SimpleBook
{
  private int     numDefs;
  private Word[] entries;
  private int     currWord;

  public Dictionary( int maxPages, int maxEntries )
  {
    super( maxPages );
    numDefs = maxEntries;
    entries = new Word[numDefs];
    currWord = 0;
  }
```

Sample Solution (cont.)



```
public void addEntry( String entry, String pron, String defn )
{
  Word vocab = new Word( entry, pron, defn );
  if( currWord < numDefs )
  {
    entries[ currWord ] = vocab;
    currWord++;
  }
}

public void addEntry( Word vocab )
{
  if( currWord < numDefs )
  {
    entries[ currWord ] = vocab;
    currWord++;
  }
}
```

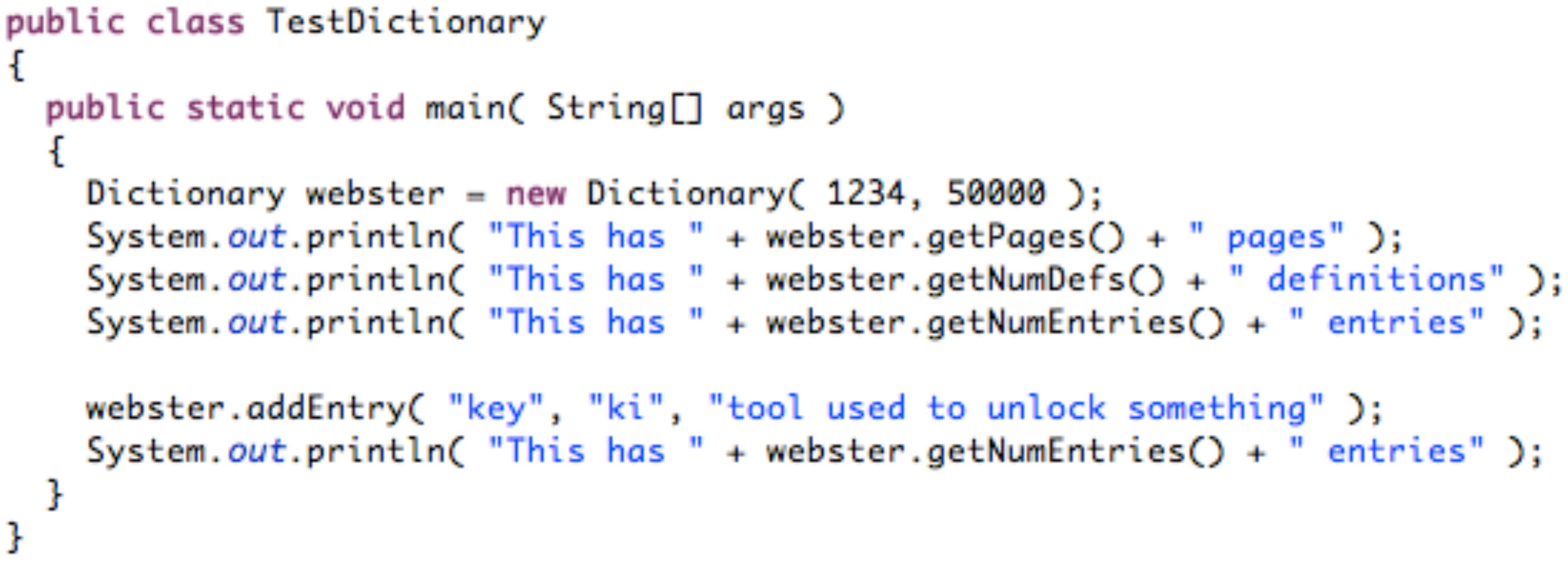
Sample Solution (cont.)



```
  public int  getNumEntries()              { return currWord; }
  public void setNumDefs( int newNumDefs ) { numDefs = newNumDefs; }
  public int  getNumDefs()                 { return numDefs; }
}
```

Sample Solution (cont.)

- Testing:
  - Call the methods you created
  - Check outputs before and after



```
public class TestDictionary
{
  public static void main( String[] args )
  {
    Dictionary webster = new Dictionary( 1234, 50000 );
    System.out.println( "This has " + webster.getPages() + " pages" );
    System.out.println( "This has " + webster.getNumDefs() + " definitions" );
    System.out.println( "This has " + webster.getNumEntries() + " entries" );

    webster.addEntry( "key", "ki", "tool used to unlock something" );
    System.out.println( "This has " + webster.getNumEntries() + " entries" );
  }
}
```

Visibility Modifiers Revisited

- Previously, you saw:
  - No visibility modifier (called “default”)
  - public
  - private
- Recall encapsulation rules
  - Don’t ever leave anything default
  - Unless there’s a reason, classes are public
  - All class attributes are private; access and changes must be done via accessors and mutators
  - Only methods that are to be called by other classes should be public, all other methods (“helpers” within the class) are private

New: `protected`

- Allows a child class to access an attribute or method from the parent class
  - Like granting special access to child classes
  - Trusted classes can see more of the parent class
  - Unrelated classes won't be able to see the info

- Note: protected info is also visible to any class in the same package (not part of this course)

The `super` Reference

- Constructors are not inherited
  - Even though they have public visibility
- Recall purpose of constructors: to set up attributes
- In many cases, we still want to reuse the parent class's setup
- Solution: call `super()` as if you were calling the parent constructor directly
  - Pass in the same input as you would

More on `super`

- Aside from the constructor, you can use super to call other methods and attributes in the parent class
- Examples in Dictionary.java:

```
super.setPages( 5000 );
super.pages = 2;
```

- Be careful not to break encapsulation rules!
  - Use accessors and mutators when possible

Multiple Inheritance

- This means a class is derived from two or more classes
- Example:

```
PickupTruck extends Truck
PickupTruck extends Car
```

- Problem:
  - Collisions – different parents may have the same attributes and/or method signatures
- Java only supports single inheritance; multiple inheritance is not allowed

Example

- A motorcycle inherits properties from both a bicycle and a car
  - Motorcycles and Bikes are two-wheeled vehicles
  - Motorcycles and Cars have engines, gas, fuel, similar speeds
- Java does not allow multiple inheritance
- How would you implement Motorcycle as a child class?

Summary of New Concepts

- Inheritance models IS-A relationship
- Visibility modifier: protected
- Use of super() calls parent's constructor
- A class cannot inherit from more than one class

- Next class: continue on inheritance