

## CIS 226 – Intro to Java

### Object Encapsulation

---

---

---

---

---

---

---

---

### OOP - Encapsulation

- When we think of objects in real-life, we associate specific properties and behaviors with those objects.
- As programmers, we would like to develop systems that simulate the real life situation into an abstraction.
- Things like books, cars, students, and people.
- Object-Oriented programming tries to simulate real life situations, entities, and OBJECTS.

---

---

---

---

---

---

---

---

### Objects

- Java is an extensible language that allows us to create our own data types (objects through source code).
- These new data types contain the properties (attributes) and actions (behaviors) similar to their real-world counter parts.
- Objects encapsulate these properties and actions.
- Attributes and behaviors translate into *member variables* and *methods* in Java.
- Objects are created through a process known as instantiation. This is the process that reserves computer memory in order to store object's data.

---

---

---

---

---

---

---

---

## Object Examples

- Book Object Example – This is the class definition

```
public class Book {
    public String preface;           // member variable
    public String contents;         // (attributes)
    public String[] bodyPages;
    public String glossary;
    public Book() {                 // object constructor
        ...                         // this is the method
    }                                 // called to create a
}                                     // Book object
```

---

---

---

---

---

---

---

---

## Book Example

- The Book class has one method which uses the same name as the class for the method name and no return type. This is known as the class *constructor*.
- Constructors are methods used to initialize object data before the class is used by a client/programmer.
- If you don't specify a constructor, the compiler will provide a default constructor for you.
- Remember that the JVM will initialize all primitive ints to 0 and object to references to null if you don't initialize values for member variables in the constructors.

---

---

---

---

---

---

---

---

## Book Example continued...

- To construct our Book object we use the following code:  
`Book myBook = new Book();`
- We need to use the keyword "new" to allocate memory for our object.
- Once the object has been constructed, we can call methods of our new object or access member variables via the dot "." notation.

```
myBook.prelude;
```

---

---

---

---

---

---

---

---

## Access Modifiers

- Access modifiers determine the access that *member variables* will have and who will be able to change the data stored in these variables.
- In our Book class we declare the member variables to be "public". The public keyword states that everyone can access these variables. There aren't any restriction on the variables. Thus, any other class whether its in the same package or not can access these variables simply by using dot notation. Ex.

```
Book myBook = new Book( );
myBook.preface = "23";
```

- This code violates one of the key principles of OOP and encapsulation, data hiding.
- How do we ensure access to member variables or methods.

---

---

---

---

---

---

---

---

## Access modifiers continued...

- Java provides additional modifiers to restrict access.
- Java provides the following keywords as access modifiers:
  - public - everyone has access
  - protected - subclass and classes in the same package
  - private - only the same class
  - package - only classes in the same package

---

---

---

---

---

---

---

---

## Access modifiers continued...

Modifier	Accessible to?			
	Same class	Class in same package	Subclass in different package	Non-subclass in different package
public	YES	YES	YES	YES
protected	YES	YES	YES	No
package	YES	YES	NO	NO
private	YES	NO	NO	NO

---

---

---

---

---

---

---

---

## Refactored Book Example

```
public class Book {
    private String preface;           // restrict access to
    private String contents;         // only this class
    private String[] bodyPages;
    private String glossary;
    public Book() {                  // object constructor
        this.preface = null;
        this.bodyPages = null;
        this.glossary = null;
    }
    public Book(String[] bodyPages) { // overload the
        this.bodyPages = bodyPages; // constructor
    }
    public Book(String preface, String contents,
                 String[] pages, String glossary) {
        this.preface = preface;
        this.contents = contents;
        this.bodyPages = pages;
        this.glossary = glossary;
    }
}
```

---

---

---

---

---

---

---

---

## Refactored Book Example...

- We've now restricted access to the book's data; preface, bodyPages, and glossary. Only the Book class can access these variables
- Why?
- This hides the implementation of the class and allows the data to be validated.
- Eliminates common problems associated with procedural programs by limited access to these variables.

---

---

---

---

---

---

---

---

## Refactored Book Example...

- We've also added two additional constructors to our class.
- Now we can construct a Book object and set the data that will be stored in the Book.
- Since all member variables are restricted to the class itself (private), we can only set bodyPages, contents, and the glossary via the constructor provided.
- Note: the constructors do not have a return type and they use the Class name as their method name.

---

---

---

---

---

---

---

---

## Refactored Book Example

The refactored code introduced the use of the keyword **"this"**. The keyword **"this"** refers to the current object.

In the following code, we have two variables named **"bodyPages"**. One declared as a local variable for the method and one declared as a member variable of the class.

In order to distinguish the two variables, we make use of the **"this"** operator. Since, **"this"** refers to the current object. The code **this.bodyPages** refers to the member variable and the **bodyPages** variable refers to the method's local variable.

```
public Book(String[] bodyPages) {
    this.bodyPages = bodyPages;
}
```

---

---

---

---

---

---

---

---

## How do we construct a Book Object?

```
public static void main(String[] args) {
    Book book1 = new Book();//default constructor

    String[] nwPages = { "...", "...", "...", ...};
    Book book2 = new Book( nwPages );

    String newPreface = "...";
    String newGloss = "...";
    Book book3 = new Book( newPreface, nwPages,
                          newGloss );

    // book2.preface = newPreface; // ERROR!
    // HOW DO WE SET THE
    // PREFACE FOR BOOK2?
}
```

---

---

---

---

---

---

---

---

## How do we set or get the private variables?

- Problem: If the book's pages are restricted to private access. Other classes have no way of retrieving this data or changing the data.
- Solution: We can provide accessor and mutator methods to private data. This gives the programmer the ability to validate or display appropriate data.

---

---

---

---

---

---

---

---

## Accessors - Getters

- Accessors are methods that provide access to a class' member variables.
- We can add the following code to our Book class:

```
public String getPreface() {  
    return preface;  
}  
public String[] getBodyPages() {  
    return bodyPages;  
}  
public String getGlossary() {  
    return glossary;  
}
```

---

---

---

---

---

---

---

---

## Mutators - Setters

- Mutators are methods that provide the ability to change data stored in a class without receiving direct access to the class' data.
- We can add the following code to our Book class:

```
public void setPreface(String preface) {  
    this.preface = preface;  
}  
public void setBodyPages(String[] bodyPages) {  
    this.bodyPages = bodyPages;  
}  
public void setGlossary(String glossary) {  
    this.glossary = glossary;  
}
```

---

---

---

---

---

---

---

---

## Question about getters and setters?

- If the getters just return the data, then why do we need the getter and private data?
  - Since the data is restricted, the programmer writing the class can determine what to return to the user/client.
  - Limited access ensures that the data won't get set to bogus data by some arbitrary assignment.

---

---

---

---

---

---

---

---

## Question about getters and setters?

- If the setter just assigns the variable to the data the user specified, then why do we need the setter?
  - Since the data is restricted again, the programmer writing the class can determine what is valid data and what to set the data to if the user decides to enter bogus data.
  - Limited access ensures that the data won't get set to bogus data by some arbitrary assignment.

---

---

---

---

---

---

---

---

## Question about getters and setters?

Example:

```
public void setPreface(String pFace) {  
    if( pFace == null )  
        preface = "No preface is" +  
                " currently " +  
                " available";  
    else  
        preface = pFace;  
}
```

---

---

---

---

---

---

---

---

## Questions about getter and setters?

Example:

```
public String getPreface() {  
    if( preface == null )  
        return "Preface is unavailable!";  
    else  
        return preface;  
}
```

---

---

---

---

---

---

---

---

## Class Scope

- Class scope allows us to call any method or member variable from within the class it was defined in, simply by name.
- Assume we have a method named "indexBook" and a method called "addPage" in our Book class.
- Note: The method indexBook() is declared as having "private" access. Thus, only code from within the Book class can call the method.

```
private void indexBook() {  
    bodyPages ... // we can call  
                // variables by name  
    ...Do something...  
    addPage( newPage ); // as well as  
                       // methods  
}
```

---

---

---

---

---

---

---

---

## Constant Member Variables

- Java uses the keyword "final" to denote variables that cannot be changed once they are set to their initial value.
- Example, assume that we only want books of 500 pages or less.

```
public class Book {  
    public final int MAXPAGES = 500;  
    ...  
}
```

- We can use the MAXPAGES variable for validity checks when adding pages to our book. Once the book has 500 pages, we can refuse any additional request to add pages to the book.
- If we try to set MAXPAGES to another value, the compiler will complain.

---

---

---

---

---

---

---

---

## Static Variables

- We can use the keyword "static" to denote variables that belong to the class not just a particular object.
- Static variables eliminate redundant data from our classes.
- Since MAXPAGES is the same for each Book, 500. MAXPAGES is a perfect variable to be declared as static.

```
public class Book {  
    public static final int MAXPAGES = 500;  
    ...  
}
```

---

---

---

---

---

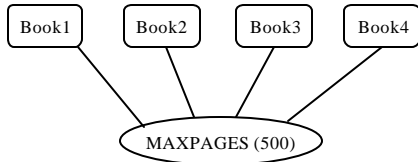
---

---

---

## What does this really look like?

- With multiple objects (instances) of the Book class



- There is only one variable MAXPAGES

---

---

---

---

---

---

---

---

## More with “static”

```
public class SimpleSurvey {
    public int MAXPAGES = 10;
    public static void main( String[] args ) {
        System.out.println( MAXPAGES );    // ERROR
    }
}
```

```
public class SimpleSurvey {
    public static int MAXPAGES = 10;
    public static void main( String[] args ) {
        System.out.println( MAXPAGES );    // OK
    }
}
```

This main belongs to the class “SimpleSurvey” not just one object. Therefore, all variables accessed from inside the main method that are not local variables must be declared “static”. Also, any methods called from within the main must also be declared as static. Stating that they belong to the class as well. Without the static keyword, the main method would not know which object’s data to refer to.

---

---

---

---

---

---

---

---

## Composition

- Classes can have references to other objects as member variables
- Example:

```
public class Book {
    private String preface; //composition
    ...
}
```

---

---

---

---

---

---

---

---

## Composition Continued...

```
■ Example 2:  
public class HardBackBook {  
    private Book mainBook;    // composted of a book object  
    private String CoverArt;  
    private String publisher;  
    ...  
public class PaperbackBook {  
    private Book mainBook;    // composted of a book object  
    private String publisher;  
    private String printer;  
    ...
```

---

---

---

---

---

---

---

---

## Shallow and Deep Copy

- Shallow Copy
    - Refers to a copy of an object's reference
    - Two or more object reference will point to the same memory location

```
Book book1 = new Book();  
Book book2 = book1;  
book1 and book2 point to the same book!
```
  - Deep Copy
    - Refers to constructing a new object in memory that has the same attributes and methods as the object being copied.

```
Book book1 = new Book();  
Book book2 = new Book( book1 );
```
- This code assumes we have a constructor that takes a book as a parameter and constructs a new Book object that is identical to "book1".

---

---

---

---

---

---

---

---

## Class vs. Object

- Keep in mind that a class is not an object
- An object is the actual constructed (instantiated object) in the computer's memory
- A class is a description of an object, but not the actual object.
- Think of classes as blueprints, just like houses have blueprints, and think of objects as the actual physical home after it has been built.
- Objects have behaviors and attributes
- Classes describe these behaviors and attributes

---

---

---

---

---

---

---

---

## **Inner Classes**

- Java provides the ability to declare a class from within another class.
- This is unlike C++.
- There will be more about Inner classes shortly.

---

---

---

---

---

---

---

---