# CMPS 251

Read Section 8.8

## Lecture 07

# Composition

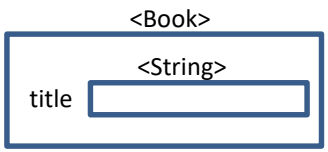
# Summary of Lecture 06

- Declaration of single and multi dimensional array
- How to assign and retrieve value from array using index.
- Basic built-in methods of array such as .length
- Array of user-defined object type
- **ArrayList**: declaration, built in method such as .size, .add, .get, set, etc.
- **Reference variables** and memory mapping
- Non-reference variables with primitive data types.
- Exception handling using **try, catch**, and **finally**

# Composition – Nested Classes

- In Java, it is also possible to nest classes (a class within a class).
- The purpose of nested classes is to group classes that belong together
- Sometimes you want an attribute of one class to be an object
- Embedding **one class** inside **another** is called composition
- We can also call this "the *has a* relationship"
- Examples:
  - **Car** has an **Engine** (**Car** is a class; **Engine** is also a class)
  - **Account** has an **Owner**
  - **BookStore** has a **Book** (or books)

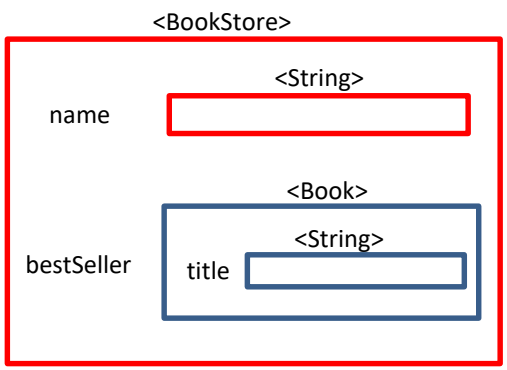
# Composition Class Definition




```
public class Book {
        private String title;

        public Book(String title) {
                this.title = title;
        }

        // getters and setters
        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }
}
```



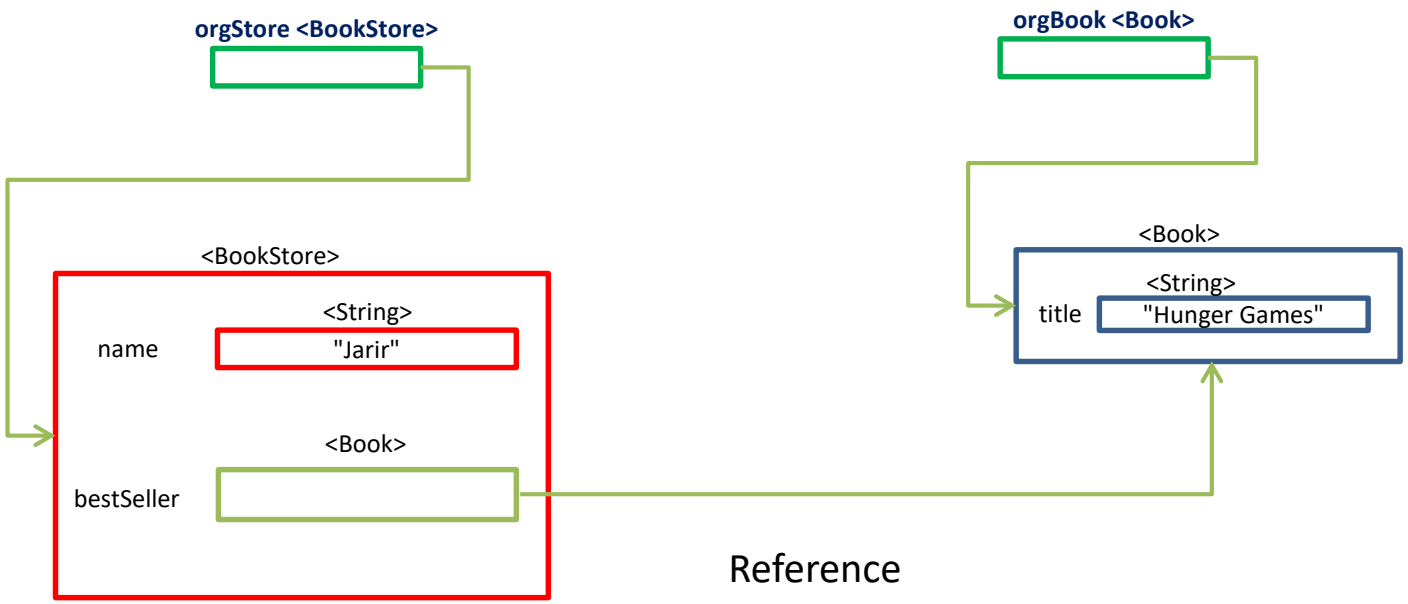

```
public class BookStore {
        private String name;
        private Book bestSeller;

        public BookStore(String name, Book bestSeller) {
                this.name = name;
                this.bestSeller = bestSeller;
        }
}
```

Composition!

# Referencing After Instantiation

```
public class App {
    public App() {
        Book orgBook = new Book("Hunger Games");
        BookStore orgStore = new BookStore("Jarir", orgBook);
    }
}
```



# Objects and References

- Once a class is defined, you can declare variables (object reference) of that type

```
Book book1, book2;
BookStore store1;
Author author1;
```

- Object references are initially **`null`**
  - The **`null`** is a special value in Java indicating that the object is NOT created yet

- The ***new*** operator is required to create the object

```
ClassName variableName = new ClassName();
```

# Comparing objects (using ==)

```
public class App {
    public App() {
        Book orgBook = new Book("Hunger Games");
        BookStore orgStore = new BookStore("Jarir", orgBook);

        Book copybook = new Book(“Hunger Games”);
        BookStore copyStore = new BookStore(“Jarir”, copybook);

        if (orgStore == copyStore)
        {
                System.out.println(“They have the same values”);
        }
    }
}
```

Will this line work?

# The solution to the == problem?

## Define an `equalsTo` method

```
public class Book {
…..
    public boolean equalsTo(Book book) {

        return this.title.equals(book.title);

    }
}
```

Two Books are equal if their *title* values are the same.

# Code Deconstructed <equalsTo method>

```
Book book1 = new Book(“Alf Laila wa Laila”);

Book book2 = new Book(“Alf Laila wa Laila”);


if (book1 == book2)
   System.out.println("Both variables refer to the same object");
else
   System.out.println("Each variable refers to a different object");


if ( book1.equalsTo(book2) )
   System.out.println("Both objects have the same value");
else
   System.out.println("Each object has a different value");
```

Can we use book2.equalsTo(book1) instead?

# Question

- What if we want to compare BookStores?

```
public class BookStore {
…..
    public boolean equalsTo(BookStore bookStore) {


       return    this.name.equals(bookStore.name) &&
                 this.bestseller.equalsTo(bookStore.bestSeller);

    }
}
```

# Using ArrayList in composition

```
public class Book
{
private String title;
private String author;

Book(String title, String author){
  this.title = title;
  this.author = author;
}

public String getTitle() {
  return title;
}

public void setTitle(String title) {
  this.title = title;
}

public String getAuthor() {
  return author;
}

public void setAuthor(String author) {
  this.author = author;
}

public String toString() {
  return String.format("Title : %s - Author : %s", getTitle(), getAuthor());
}
}
```

# Using ArrayList in composition

```
import java.util.ArrayList;

public class Library
{
    private ArrayList<Book> books;

    Library () {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book) {
         books.add(book);
    }

    public Book getBook(String bookTitle) {
                  for(Book book : books)
                  {
                           if (book.getTitle().equalsIgnoreCase(bookTitle))
                                    return book;
                  }
                  return null;
         }

    public ArrayList<Book> getBooks(){
       return books;
    }
}
```

# Using ArrayList in composition

```
import java.util.ArrayList;

public class LibraryUI {
    public static void main (String[] args)
    {
    Library library = new Library();

        // Creating the Objects of Book class.
        Book book = new Book("Effective Java", "Joshua Bloch");
        library.addBook(book);

        book = new Book("Thinking in Java", "Bruce Eckel");
        library.addBook(book);

        book = new Book("Java: The Complete Reference", "Herbert Schildt");
        library.addBook(book);

        Book abook = library.getBook("Thinking in Java");
        System.out.printf("%s is authored by %s \n", abook.getTitle(),
                          abook.getAuthor());

        ArrayList<Book> books = library.getBooks();
        for(Book b : books){
            System.out.println(b);
        }  } }
```

# Java Inner Classes

- In Java, it is also possible to nest classes (a class within a class).
- The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- One advantage of inner classes, is that they can access attributes and methods of the outer class

# Java Inner Classes

- To access the inner class, create an object of the outer class, and then create an object of the inner class:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

# Private Inner Class

- Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

```java
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

If you try to access a private inner class from an outside class (MyMainClass), an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in
OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
              ^
```

# Static Inner Class

- An inner class can also be `static`, which means that you can access it without creating an object of the outer class
- Just like `static` attributes and methods, a `static` inner class does not have access to members of the outer class.
- If you try to access a private inner class from an outside class (MyMainClass), an error occurs

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

# Access Outer Class From Inner Class

- One advantage of inner classes, is that they can access attributes and methods of the outer class:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```