8

Object Life Cycle and Inner Classes

Objectives

After completing this lesson, you should be able to do the following:

- Provide two or more methods with the same name in a class
- Provide one or more constructors for a class
- Use initializers to initialize both instance and class variables
- Describe the class loading and initializing process and the object life cycle
- Define and use inner classes

Overloading Methods

- Several methods in a class can have the same name.
- The methods must have different signatures.

```
public class Movie {
  public void setPrice() {
    price = 3.50F;
  }
  public void setPrice(float newPrice) {
    price = newPrice;
  } …
}
```

```
Movie mov1 = new Movie();
mov1.setPrice();
mov1.setPrice(3.25F);
```

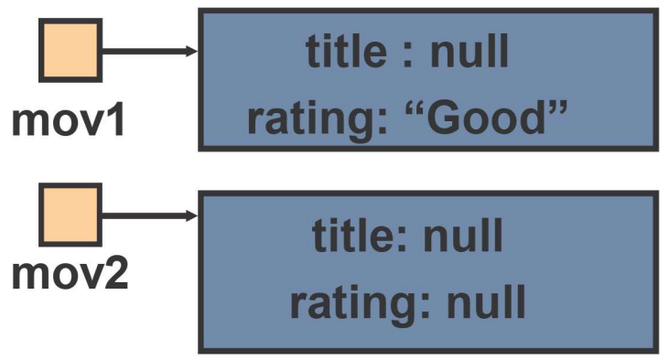
Using the `this` Reference

Instance methods receive an argument called `this`, which refers to the current object.

```
public class Movie {
  public void setRating(String newRating) {
    this.rating = newRating;
  }
```

this

```
void anyMethod() {
 Movie mov1 = new Movie();
 Movie mov2 = new Movie();
 mov1.setRating("Good"); …
```



Initializing Instance Variables

- Instance variables can be explicitly initialized at declaration.
- Initialization happens at object creation.

```
public class Movie {
  private String title;
  private String rating = "Good";
  private int numOfOscars = 0;
```

- All instance variables are initialized implicitly to default values depending on data type.
- More complex initialization must be placed in a constructor.

Constructors

- For proper initialization, a class must provide a constructor.
- A constructor is called automatically when an object is created:
  - It is usually declared `public`.
  - It has the same name as the class.
  - It must not specify a return type.
- The compiler supplies a `no-arg` constructor if and only if a constructor is not explicitly provided.
  - If a constructor is explicitly provided, the compiler does not generate the `no-arg` constructor.

Defining and Overloading Constructors

```
public class Movie {
  private String title;
  private String rating = "PG";

  public Movie() {
    title = "Last Action …";
  }
  public Movie(String newTitle) {
    title = newTitle;
  }
}
```

The Movie class now provides two constructors.

```
Movie mov1 = new Movie();
Movie mov2 = new Movie("Gone …");
Movie mov3 = new Movie("The Good …");
```

Sharing Code Between Constructors

```
Movie mov2 = new Movie();
```

A constructor can call another constructor by using this().

What happens here?

```
public class Movie {
  private String title;
  private String rating;

  public Movie() {
    this("G");
  }
  public Movie(String newRating) {
    rating = newRating;
  }
}
```

`final` Variables, Methods, and Classes

- A `final` variable is a constant and cannot be modified.
  - It must therefore be initialized.
  - It is often declared `public static` for external use.
- A `final` method cannot be overridden by a subclass.
- A `final` class cannot be subclassed (extended).

```
public final class Color {
  public final static Color black=new Color(0,0,0);
  …
}
```

Reclaiming Memory

- When all references to an object are lost, the object is marked for garbage collection.
- Garbage collection reclaims memory that is used by the object.
- Garbage collection is automatic.
- There is no need for the programmer to do anything, but the programmer can give a hint to `System.gc();`.

`finalize()` Method

- If an object holds a resource such as a file, the object should be able to clean it up.
- You can provide a `finalize()` method in that class.
- The `finalize()` method is called just before garbage collection.

```
public class Movie {
  …
  public void finalize() {
    System.out.println("Goodbye");
  }
}
```

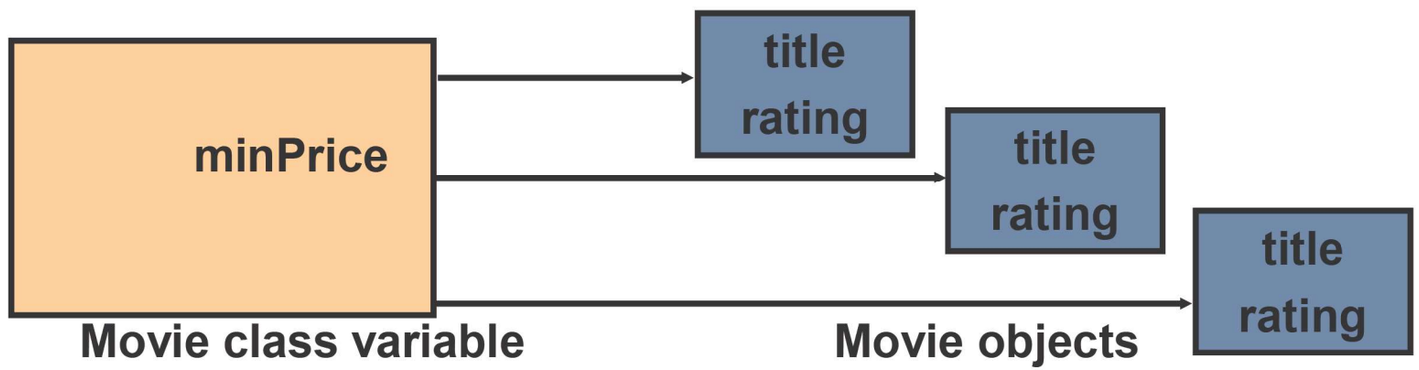
Any problems?

Class Variables

Class variables:

- Belong to a class and are common to all instances of that class
- Are declared as static in class definitions

```
public class Movie {
  private static double minPrice;   // class var
  private String title, rating;     // inst vars
```



Initializing Class Variables

- Class variables can be initialized at declaration.
- Initialization takes place when the class is loaded.
- Complex initialization of class variables is performed in a static initializer block.
- All class variables are initialized implicitly to default values depending on the data type.

```
public class Movie {
  private static double minPrice = 1.29;
  private String title, rating;
  private int length = 0;
```

Class Methods

Class methods are:

- Shared by all instances
- Useful for manipulating class variables
- Declared as static

```
public static void increaseMinPrice(double inc) {
  minPrice += inc;
}
```

A class method is called using the name of the class or an object reference.

```
Movie.increaseMinPrice(.50);
mov1.increaseMinPrice(.50);
```

Guided Practice: Class Methods

```
public class Movie {

 private static float price = 3.50f;
 private String rating;
  …
 public static void setPrice(float newPrice) {
    price = newPrice;
 }
 public String getRating() {
    return rating;
 }
}
```

Legal or not?

```
Movie.setPrice(3.98f);
Movie mov1 = new Movie(…);
mov1.setPrice(3.98f);
String a = Movie.getRating();
String b = mov1.getRating();
```

Examples of Static Methods in Java

Examples of static methods:

- `main()`
- `Math.sqrt()`
- `System.out.println()`

```
public class MyClass {

 public static void main(String[] args) {
  double num, root;
  …
  root = Math.sqrt(num);
  System.out.println("Root is " + root);
 } …
```

Inner Classes

- Inner classes are nested classes that are defined in a class or method.
- Inner classes enforce a relationship between two classes.
- There are four types of inner classes:
  - Static
  - Member
  - Local
  - Anonymous

```
public class Outer { …
        class Inner { …
  }
}
```

Enclosing class

Static Inner Class

- A Static inner class behaves like any outer class.
- Creating an instance of the inner class does not require an instance of the outer class to exist.
- Creating an instance of the outer class does not create any instances of the inner class.

```
public class Outer {
  …
  public static class Inner {
            int x;
            void f(){
      ...}
  }
}
```

Member Inner Class

- The Member inner class is declared within another class.
- Nesting is allowed.
- It can access variables within its own class and any outer classes.
- It can declare only `final` static methods.

```
public class CalendarPopup {
  ...
  class MonthSelector {
            class DayOfMonth{...};
            DayOfMonth[] NumberOfDaysInMonth
      ...
  }
}
```

Local Inner Class

- The Local inner class is declared within a code block (inside a method).
- All `final` variables or parameters that are declared in the block are accessible by the methods of the inner class.

```
public class CalendarPopup {
  ...
  public void handlerMethod(){
        class DateHandler{…};
        DateHandler sc = new DateHandler();
        ...
  }
}
```

Anonymous Inner Class

- The Anonymous inner class is defined at the method level.
- It is declared within a code block.
- It lacks the `class`, `extends`, and `implements` keywords.
- It cannot have a constructor.

```
public class Outer {
  ...
  public void outerMethod(){
            ...
            myObject.myAnonymous(new SomeOtherClass(){
       ...
            } );
   }
}
```

Calendar Class

- The `Calendar` class converts a date object to a set of integer fields.
- It represents a specific moment in time.
- Subclasses interpret a date according to the specific calendar system.

```
import java.util.Calendar;
public class Order {
  ...
  public void String getShipDate(){
            ...
            Calendar c = Calendar.getInstance();
            c.setTime(orderDate);
            ...
  }
}
```

Summary

In this lesson, you should have learned the following:

- Methods can be overloaded in Java.
- Instance methods receive a `this` reference to the current object.
- Most classes provide one or more constructors to initialize new objects.
- Class variables and class methods can be defined for classwide properties and behaviors.
- Classes can be defined in various ways within a class.

Practice : Overview

This practice covers the following topics:

- Defining and using overloaded methods
- Providing a `no-arg` constructor for a class
- Providing additional constructors for a class
- Defining static variables and static methods for classwide behavior
- Using static methods