Representing Complex Types with Classes

Jim Wilson

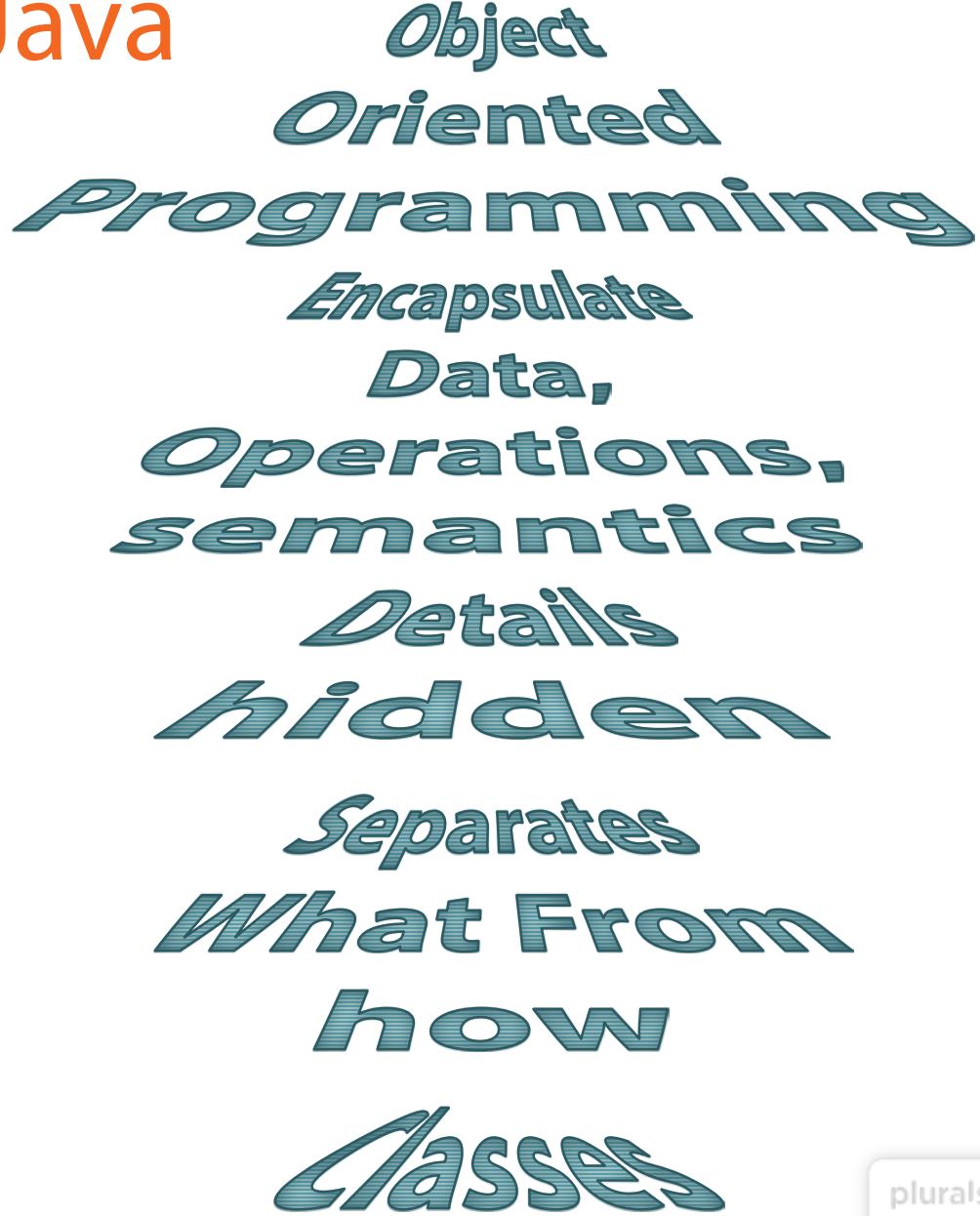
@hedgehogjim | blog.jwhh.com | jimw@jwhh.com

What to Expect in This Module

- Classes
- Using classes
- Classes as reference types
- Encapsulation and access modifiers
- Method basics
- Field accessors and mutators

Classes in Java

- Java is an object-oriented language
- Objects encapsulates data, operations, and usage semantics
  - Allows storage and manipulation details to be hidden
  - Separates "what" is to be done from "how" it is done
- Classes provide a structure for describing and creating objects



Classes

- A class is a template for creating an object
  - Declared with the class keyword followed by the class name
  - Java source file name normally has same name as the class
    - We'll talk more about this shortly
  - Body of the class is contained within brackets

Flight.java

```
class Flight {



}
```

Classes

- A class is made up of both state and executable code
  - Fields
    - Store object state
  - Methods
    - Executable code that manipulates state and performs operations
  - Constructors
    - Executable code used during object creation to set the initial state

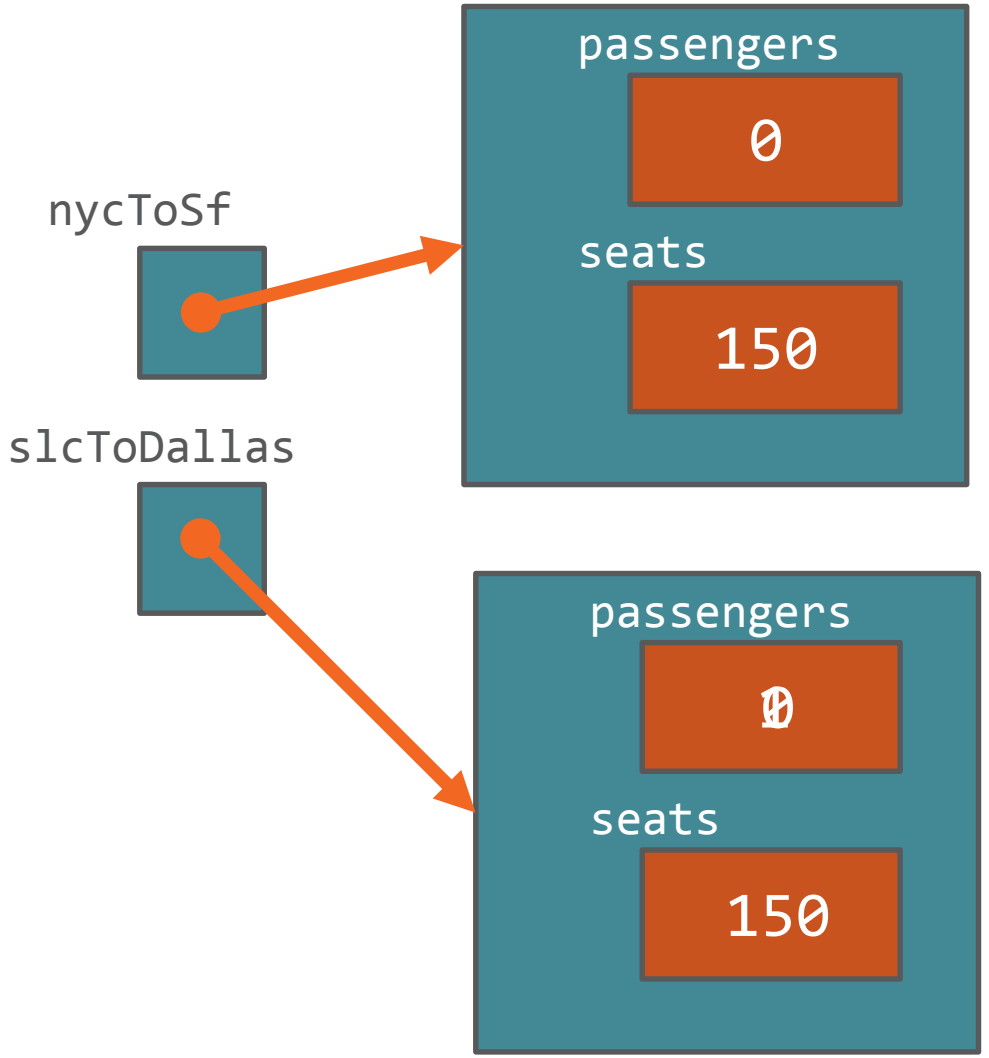
Flight.java

```
class Flight {
   int passengers;
   int seats;

   Flight() {
     seats = 150;
     passengers = 0;
   }

   void add1Passenger() {
     if(passengers < seats)
        passengers += 1;
   }
}
```

Using Classes

- Use the new keyword to create a class instance (a.k.a. an object)
  - Allocates the memory described by the class
  - Returns a reference to the allocated memory

```
Flight nycToSf;
nycToSf = new Flight();
Flight slcToDallas = new Flight();

slcToDallas.add1Passenger();
```



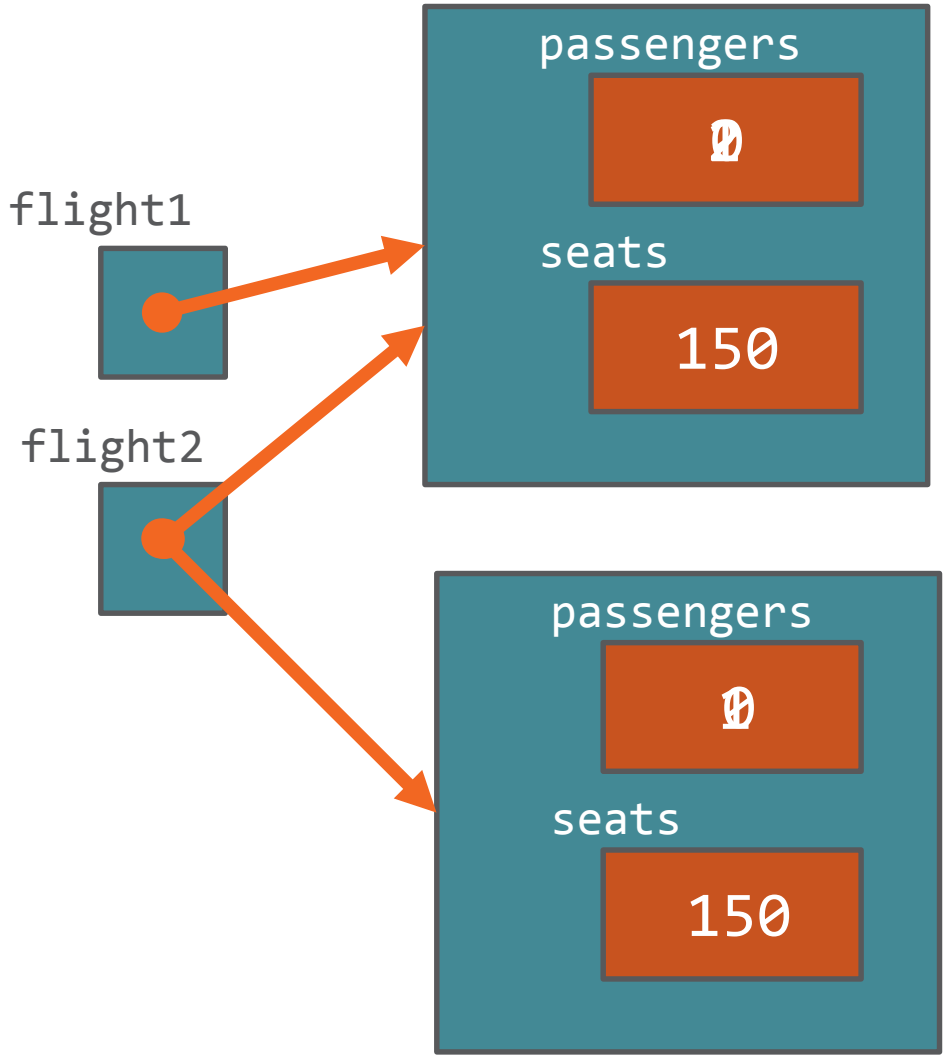
Classes Are Reference Types

```
Flight flight1 = new Flight();
Flight flight2 = new Flight();

flight2.add1Passenger();
System.out.println(flight2.passengers);

flight2 = flight1;
System.out.println(flight2.passengers);

flight1.add1Passenger();
flight1.add1Passenger();

System.out.println(flight2.passengers);
```

1

0

2



Encapsulation and Access Modifiers

The internal representation of an object is generally hidden

This concept is known as *encapsulation*

Java uses *access modifiers* to achieve encapsulation

Basic Access Modifiers

Modifier	Visibility	Usable on Classes	Usable on Members
no access modifier	Only within its own package (a.k.a. package private)	Y	Y
public	Everywhere	Y	Y
private	Only within its own class	N *	Y

* As private applies to top-level classes; private is available to nested-classes

Applying Access Modifiers

```
Flight flight1 = new Flight();

System.out.println(flight1.passengers);

flight1.add1Passenger();

flight1.handleTooMany();
```

```
public class Flight  {
   private int passengers;
   private int seats;

   public  Flight() {
              seats = 150;
              passengers = 0;
           }

   public  void add1Passenger() {
              if(passengers < seats)
                 passengers += 1;

              else
                 handleTooMany();

           }

   private void handleTooMany() {
              System.out.println("Too many");
           }
    }
```

Flight.java

Naming Classes

- Class names follow the same rules as variable names
- Class name conventions are similar to variables with some differences
  - Use only letters and numbers
  - First character is always a letter
  - Follow the style often referred to as "Pascal Case"
    - Start of each word, including the first, is upper case
    - All other letters are lower case
  - Use simple, descriptive nouns
  - Avoid abbreviations unless abbreviation's use is more common than full name

```
class BankAccount { ... }
class Person { ... }
class TrainingVideo { ... }
class URL { ... }
```

Method Basics

- Executable code that manipulates state and performs operations
  - Name
    - Same rules and conventions as variables
    - Should be a verb or action
  - Return type
    - Use void when no value returned
  - Typed parameter list
    - Can be empty
  - Body contained with brackets

```
return-type  name ( typed-parameter-list ) {
    statements
}
```

```
void showSum(float x, float y, int count) {
    float sum = x + y;
    for(int i = 0; i < count; i++)
        System.out.println(sum);
}
```

Method Basics

- Executable code that manipulates state and performs operations
  - Name
    - Same rules and conventions as variables
    - Should be a verb or action
  - Return type
    - Use void when no value returned
  - Typed parameter list
    - Can be empty
  - Body contained with brackets

```
MyClass m = new MyClass();
m.showSum(7.5, 1.4, 3);
```

```
8.9
8.9
8.9
```

```
public class MyClass {

    public void showSum (float x , float y, int count) {
        float sum = x + y;
        for(int i = 0; i < count; i++)
            System.out.println(sum);
    }

}
```

Exiting from a Method

- A method exits for one of three reasons
  - The end of the method is reached
  - A return statement is encountered
  - An error occurs
- Unless there's an error, control returns to the method caller

```
MyClass m = new MyClass();
m.showSum(7.5, 1.4, 3);
System.out.println("I'm back");
```

```
8.9
8.9
8.9
I'm back
```

```
void showSum(float x, float y, int count) {
    float sum = x + y;
    for(int i = 0; i < count; i++)
        System.out.println(sum);
    return;
}
```

Exiting from a Method

- A method exits for one of three reasons
  - The end of the method is reached
  - A return statement is encountered
  - An error occurs
- Unless there's an error, control returns to the method caller

```
MyClass m = new MyClass();
m.showSum(7.5, 1.4, 0);
System.out.println("I'm back");
```

```
void showSum(float x, float y, int count) {
    if(count < 1)
        return;
    float sum = x + y;
    for(int i = 0; i < count; i++)
        System.out.println(sum);
    return;
}
```

Method Return Values

- A method returns a single value
  - A primitive value
  - A reference to an object
  - A reference to an array
    - Arrays are objects

```
public class Flight {
  private int passengers;
  private int seats;
  // constructor and other methods elided for clarity

  public boolean hasRoom(Flight f2) {
     int total = passengers + f2.passengers;
     if (total <= seats)
        return true;
     else
        return false;
  }

}
```

Method Return Values

- A method returns a single value
  - A primitive value
  - A reference to an object
  - A reference to an array
    - Arrays are objects

```
public class Flight {
  private int passengers;
  private int seats;
  // constructor and other methods elided for clarity

  public boolean hasRoom(Flight f2) {
     int total = passengers + f2.passengers;
     return total <= seats;



  }




}
```

Method Return Values

- A method returns a single value
  - A primitive value
  - A reference to an object
  - A reference to an array
    - Arrays are objects

```
public class Flight {
  private int passengers;
  private int seats;
  // constructor and other methods elided for clarity

  public boolean hasRoom(Flight f2) {
     int total = passengers + f2.passengers;
     return total <= seats;
  }

  public Flight createNewWithBoth(Flight f2) {
     Flight newFlight = new Flight();
     newFlight.seats = seats;
     newFlight.passengers = passengers + f2.passengers;
     return newFlight;
  }
}
```

Method Return Values

```
Flight lax1 = new Flight();
Flight lax2 = new Flight();
// add passengers to both flights

Flight lax3;
if(lax1.hasRoom(lax2))
    lax3 =
       lax1.createNewWithBoth(lax2);
```

```
public class Flight {
  private int passengers;
  private int seats;
  // constructor and other methods elided for clarity

  public boolean hasRoom(Flight f2) {
     int total = passengers + f2.passengers;
     return total <= seats;
  }

  public Flight createNewWithBoth(Flight f2) {
     Flight newFlight = new Flight();
     newFlight.seats = seats;
     newFlight.passengers = passengers + f2.passengers;
     return newFlight;
  }
}
```

Special References: this and null

- Java provides special references with predefined meanings
  - *this* is an implicit reference to the current object
    - Useful for reducing ambiguity
    - Allows an object to pass itself as a parameter
  - *null* is a reference literal
    - Represents an uncreated object
    - Can be assigned to any reference variable

```
public class Flight {
  private int passengers;
  private int seats;
  // constructor and other methods elided for clarity

  public boolean hasRoom(Flight f2) {
     int total = this.passengers + f2.passengers;
     return total <= seats;
  }
}
```

Special References: this and null

- Java provides special references with predefined meanings
  - *this* is an implicit reference to the current object
    - Useful for reducing ambiguity
    - Allows an object to pass itself as a parameter
  - *null* is a reference literal
    - Represents an uncreated object
    - Can be assigned to any reference variable

```
Flight lax1 = new Flight();
Flight lax2 = new Flight();
// add passengers to both flights

Flight lax3;= null;
if(lax1.hasRoom(lax2))
   lax3 =
      lax1.createNewWithBoth(lax2);

// do some other work

if(lax3 != null)
   System.out.println("Flights combined");
```

Demo
CalcEngine with Classes and Methods

Field Encapsulation

In most cases, a class' fields should not be directly accessible outside of the class

Helps to hide implementation details

Use methods to control field access

Accessors and Mutators

- Use the accessor/mutator pattern to control field access
  - Accessor retrieves field value
    - Also called getter
    - Method name: get*FieldName*
  - Mutator modifies field value
    - Also called setter
    - Method name: set*FieldName*

```
public class Flight {
  private int passengers;
  private int seats;
  // other members elided for clarity

  public int getSeats() {
     return seats;
  }

  public void setSeats(int seats) {
     this.seats = seats;
  }
}
```

Accessors and Mutators

```
Flight slcToNyc = new Flight();
slcToNyc.setSeats(150);
System.out.println(slcToNyc.getSeats());
```

150

```
public class Flight {
  private int passengers;
  private int seats;
  // other members elided for clarity

  public int getSeats() {
     return seats;
  }

  public void setSeats(int seats)
  {  this.seats = seats;
  }

}
```

Demo

CalcEngine with Accessors and Mutators

Summary

- A class is a template for creating an object
  - Declared with class keyword
  - Class instances (a.k.a. objects) allocated with new keyword
- Classes are reference types
- Use access modifiers to control encapsulation
- Methods manipulate state and perform operations
  - Use return keyword to exit and/or return a value
- Fields store object state
  - Interaction normally controlled through accessors(getters) and mutators(setters)