Working with Wrapper Classes, Enums, and Records

Jim Wilson
MOBILE SOLUTIONS DEVELOPER & ARCHITECT

@hedgehogjim jwhh.com

Overview

- Primitive wrapper classes
- Role of enum types
- Conditional logic with enums
- Using class-based features of enums
- Records

Primitive types

- byte, short, int, long
- float, double
- char
- boolean

Primitive types represent data only

- Unable to provide methods for operating on that data

Primitive wrapper classes

- Can hold primitive data values
- Provide methods
- Enable compatibility with richer aspects of Java type system

Each primitive type has a wrapper class

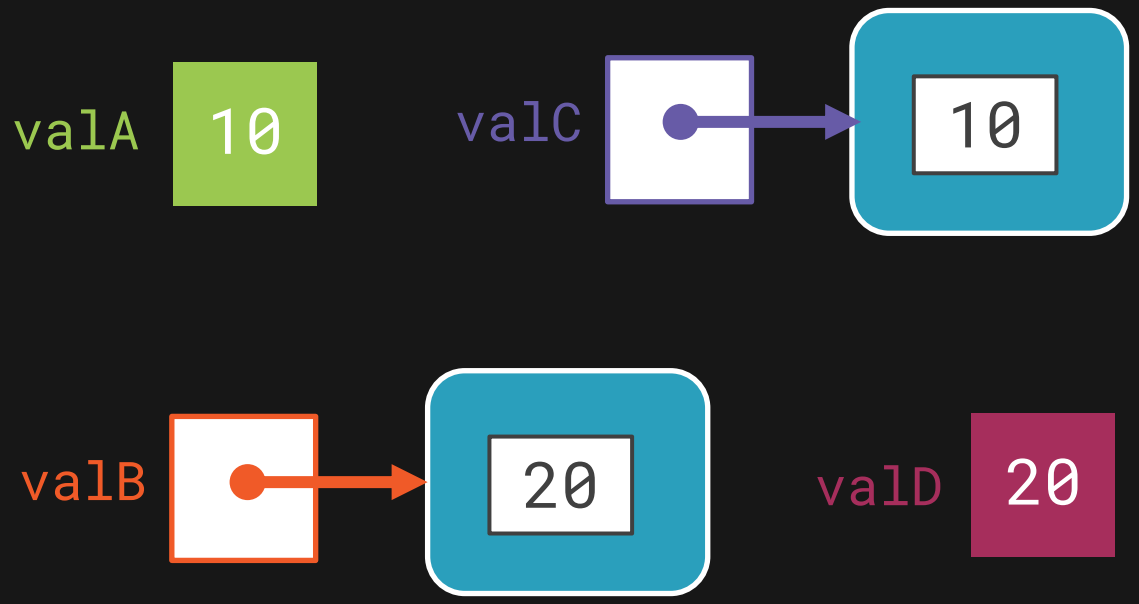
- Byte, Short, Integer, Long
- Float, Double
- Character
- Boolean

```
int valA = 10;

Integer valB = 20

Integer valC = valA;

int valD = valB;
```



Converting to and from a Wrapper Class

Boxing

- Converting from primitive type to a wrapper class

Unboxing

- Converting from a wrapper class to a primitive type

Methods handle common operations

- Converting to from other types
- Extracting values from strings
- Finding min/max values
- Many others

```
public enum FlightCrewJob {
  FLIGHT_ATTENDANT,
  COPILOT,
  PILOT
}
```

Enumeration Types

Useful for defining a type with a finite list of valid values

- Declare using the enum keyword
- Provide comma-separated value list
- By convention value names are all upper case

```
FlightCrewJob job1 = FlightCrewJob.PILOT;
FlightCrewJob job2 = FlightCrewJob.FLIGHT_ATTENDANT;
if(job1 == FlightCrewJob.PILOT)
    System.out.println("job1 is PILOT");
if(job1 != job2)
    System.out.println("job1 and job2 are different");
```

Conditional Logic

Enums support equality tests

- Can use == and != operators

Enums support switch statements

Main.java

```
void displayJobResponsibilities(FlightCrewJob job) {
  switch(job) {
    case FLIGHT_ATTENDANT:
      System.out.println("Assures passenger safety");
      break;
    case COPILOT:
      System.out.println("Assists in flying the plane");
      break;
    case PILOT:
      System.out.println("Flies the plane");
      break;
  }
}
```

Relative Comparisons

Values are ordered

- First value is lowest
- Last value is highest

Use compareTo for relative comparison

- Returns negative, zero, or positive value
- Indicates current instance's ordering relative to another value

Relative Comparisons

FlightCrewJob.java

```
public enum FlightCrewJob {
    FLIGHT_ATTENDANT,
    COPILOT,
    PILOT
}
```

CrewMember.java

```
class CrewMember {
  FlightCrewJob job;
  String name;
  CrewMember(FlightCrewJob job,
             String name) {
    this.job = job;
    this.name = name;
  }
} // other members elided
```

Main.java

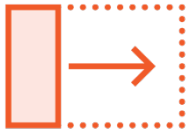
```
CrewMember geetha = new CrewMember(FlightCrewJob.PILOT, "Geetha");
CrewMember bob = new CrewMember(FlightCrewJob.FLIGHT_ATTENDANT, "Bob");
whoIsInCharge(geetha, bob);

void whoIsInCharge(CrewMember member1, CrewMember member2) {
  CrewMember theBoss
          member1 :         ;
  System.out.println(theBoss.getName() + " is boss");
}
```

Common Enum Methods

Method	Description

Enum Types Are Classes



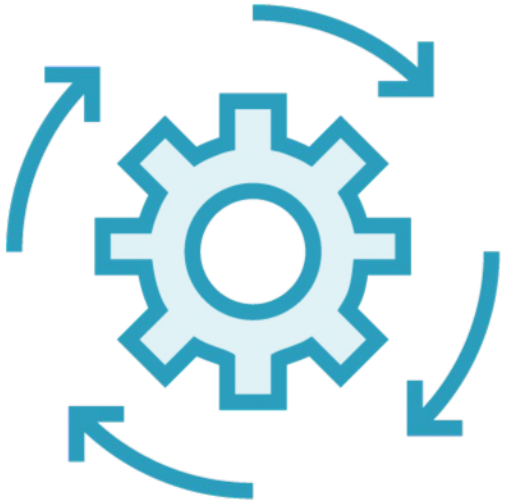
Implicitly inherit from Java's Enum class

Similar to other class in some ways

Have some special characteristics

Enum Types Can Have Members

Fields



Methods

Constructors

Enum Values

Each value is an instance of the enum type

Declaring the value creates the instance

Can leverage the enum type's constructor

FlightCrewJob.java

```
public enum FlightCrewJob {
  FLIGHT_ATTENDANT,
  COPILOT,
  PILOT

  private String title;
  public String getTitle() { return title; }
  private FlightCrewJob(String title) {
    this.title = title;
  }
}
```

FlightCrewJob.java

```
public enum FlightCrewJob {
  FLIGHT_ATTENDANT,
  COPILOT,
  PILOT("Captain");

  private String title;
  public String getTitle() { return title; }
  private FlightCrewJob(String title) {
    this.title = title;
  }
}
```

FlightCrewJob.java

```
public enum FlightCrewJob {
  FLIGHT_ATTENDANT,
  COPILOT("First Officer"),
  PILOT("Captain");

  private String title;

  public String getTitle() { return title; }

  private FlightCrewJob(String title) {
    this.title = title;
  }
}
```

FlightCrewJob.java

```
public enum FlightCrewJob {
  FLIGHT_ATTENDANT("Flight Attendant"),
  COPILOT("First Officer"),
  PILOT("Captain");

  private String title;
  public String getTitle() { return title; }
  private FlightCrewJob(String title) {
    this.title = title;
  }
}
```

Main.java

```
CrewMember geetha = new CrewMember(FlightCrewJob.PILOT, "Geetha");
CrewMember bob = new CrewMember(FlightCrewJob.FLIGHT_ATTENDANT, "Bob");
whoIsInCharge(geetha, bob);


void whoIsInCharge(CrewMember member1, CrewMember member2)
  CrewMember theBoss = member1.getJob().compareTo(member2.getJob()) > 0
          member1  : member2;
  System.out.println(
        theBoss.getName() + " is boss");
}
```

Main.java

```
CrewMember geetha = new CrewMember(FlightCrewJob.PILOT, "Geetha");
CrewMember bob = new CrewMember(FlightCrewJob.FLIGHT_ATTENDANT, "Bob");
whoIsInCharge(geetha, bob);


void whoIsInCharge(CrewMember member1, CrewMember member2)
  CrewMember theBoss = member1.getJob().compareTo(member2.getJob()) > 0
          member1 : member2;
  System.out.println(theBoss.getJob().getTitle() + " " +
        theBoss.getName() + " is boss"); // Captain Geetha is boss
}
```

Data-only Classes

Some classes serve only as data carriers

- Initialized with required data values
- Those values never change

Often involves a lot of "boilerplate" code

- Constructor to initialize instance fields
- Getters for each instance field
- Common methods such as equals, hashCode, and toString

Passenger.java

```
public class Passenger {
  private String name;
  private int checkedBags;

  public Passenger(String name, int checkedBags) {
    this.name = name;
    this.checkedBags = checkedBags;
  }
```

Passenger.java

```
  public String getName() { return name; }
  public int getCheckedBags() { return checkedBags; }

  public boolean equals(Object o) { /* compare members for equality */ }
  public int hashCode() { /* compute hash code for members */ }
  public String toString() { /* return string containing members */ }
}
```

Record

Simplifies creating data-only classes

- Declared using the record keyword
- Created class is immutable

Automatically generates common code

- Constructor to initialize instance fields
- Getters for each instance field
- equals, hashCode, and toString methods

Passenger.java

```
public record Passenger
```

Main.java

```
Passenger p1 =                          ;
String n = p1.name();
int b = p1.checkedBags();

Passenger p2 = new Passenger("Maria", 1);

if (p1.equals(p2))
  // do something
```

Summary

Primitive wrapper classes

- Can hold primitive data values
- Provide methods
- Automatic conversion between primitive types and wrapper classes

Summary

Enumeration types

- Define a finite list of valid values

Support conditional logic

- Can perform equality tests
- Work well with switch statements

Enum values are ordered

- First value is lowest
- Last value is highest
- Can perform order-based comparisons with compareTo

Summary

Enum types are classes

- Inherit from Java's Enum class
- Can define members

Enum values

- Are instances of the enum type
- Declaring a value creates the instance
- Can leverage constructors

Summary

Records

- Simplifies creating data-only classes
- Created class is immutable
- Automatically generates commonly required code