More About Data Types

Jim Wilson

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

What to Expect in This Module

- String class
- StringBuilder class
- Primitive wrapper classes
- Final fields
- Enumeration types

String Class

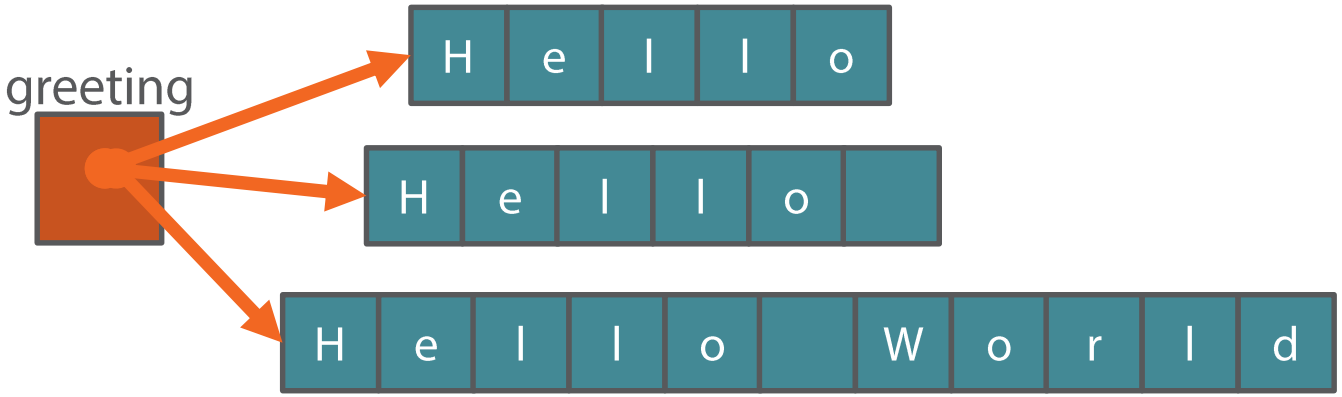
The String class stores a sequence of Unicode characters

Stored using UTF-16 encoding

- Literals are enclosed in double quotes (“ “)
- Values can be concatenated using + and +=
- String objects are immutable

```
String name = "Jim";
System.out.println("Hi " + name);
String greeting = "Hello";
greeting += " ";
greeting += "World";
```

Hi Jim



Select String Class Methods

Operation	Methods
Length	length
String for non-string	valueOf
Create new string(s) from existing	concat, replace, toLowerCase, toUpperCase, trim, split
Formatting	format
Extract substring	charAt, substring
Test substring	contains, endsWith, startsWith, indexOf, lastIndexOf
Comparison	compareTo, compareToIgnoreCase, isEmpty, equals, equalsIgnoreCase

String class documentation: http://bit.ly/javastringclass

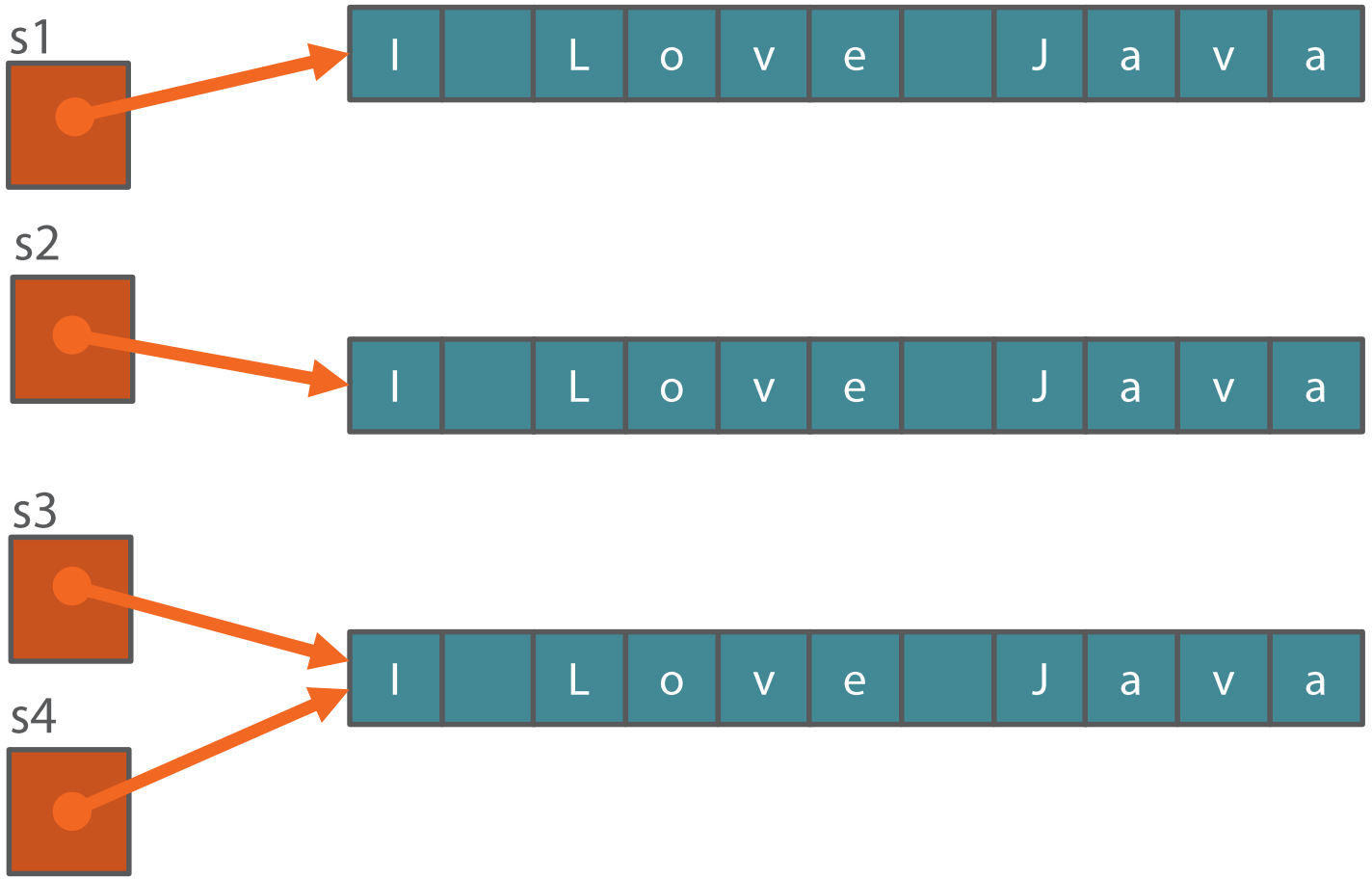
String Equality

```
String s1 = "I Love";
s1 += " Java";

String s2 = "I";
s2 += " Love Java";

if(s1 == s2)
   // do something

if(s1.equals(s2))
   // do something

String s3 = s1.intern();
String s4 = s2.intern();

if(s3 == s4)
   // do something
```

FALSE

TRUE

TRUE



Converting Non-string Types to Strings

We often need to convert non-string types into strings

- String.valueOf provides overrides to handle most types
- Conversions often happen implicitly
- Class conversions controlled by the class' toString method

```
int iVal = 100;
String sVal = String.valueOf(iVal);
```

```
int i = 2, j = 3;
int result = i * j;
System.out.println(
   i + " * " + j + " = " + result);
```

Implementing the toString Method

```
public class Flight {
  int flightNumber;
  char flightClass;
  // other members elided for clarity
  @Overrride
  public String toString() {
    if(flightNumber > 0)
      return "Flight #" + flightNumber;
    else
      return "Flight Class " + flightClass;
  }
}
```

```
Flight myFlight = new Flight(175);
System.out.println(
  "My flight is " + myFlight);
```

StringBuilder

- StringBuilder provides mutable string buffer
  - For best performance pre-size buffer
    - Will grow automatically if needed
  - Most common methods: append and insert
  - Use toString to extract resulting string

sb

```
I flew to Florida on Flight #175
```

sb.capacity() → 40

```
StringBuilder sb = new StringBuilder(40);
Flight myFlight = new Flight(175);
String location = “Florida”;

sb.append(“I flew to “);
sb.append(location);
sb.append(“ on “);
sb.append(myFlight);

int time = 9;

int pos = sb.length() - “ on “.length()
    - myFlight.toString().length();

sb.insert(pos, “ at “);
sb.insert(pos + 4, time);

String message = sb.toString();
```

StringBuilder

- StringBuilder provides mutable string buffer
  - For best performance pre-size buffer
    - Will grow automatically if needed
  - Most common methods: append and insert

sb

```
I flew to Florida on Flight #175
```

sb.capacity() → 40

```
StringBuilder sb = new StringBuilder(40);
Flight myFlight = new Flight(175);
String location = "Florida";

sb.append("I flew to ");
sb.append(location);
sb.append(" on ");
sb.append(myFlight);

int time = 9;

int pos = sb.length() - " on ".length()
   - myFlight.toString().length();
```

StringBuilder

- StringBuilder provides mutable string buffer
  - For best performance pre-size buffer
    - Will grow automatically if needed
  - Most common methods: append and insert
  - Use toString to extract resulting string

sb

```
I flew to Florida at 9 on Flight #175
```

sb.capacity() → 40

```
StringBuilder sb = new StringBuilder(40);
Flight myFlight = new Flight(175);
String location = "Florida";

sb.append("I flew to ");
sb.append(location);
sb.append(" on ");
sb.append(myFlight);

int time = 9;

int pos = sb.length() - " on ".length()
    - myFlight.toString().length();

sb.insert(pos, " at ");
sb.insert(pos + 4, time);

String message = sb.toString();
```

Classes vs. Primitives

Classes provide convenience

- ✓ Common interaction through Object class
- ✓ Fields and methods specific to the type
- Incurs an overhead cost

Primitives provide efficiency

- Cannot be treated as Object
- Cannot expose fields or methods
- ✓ Lightweight

Classes vs. Primitives

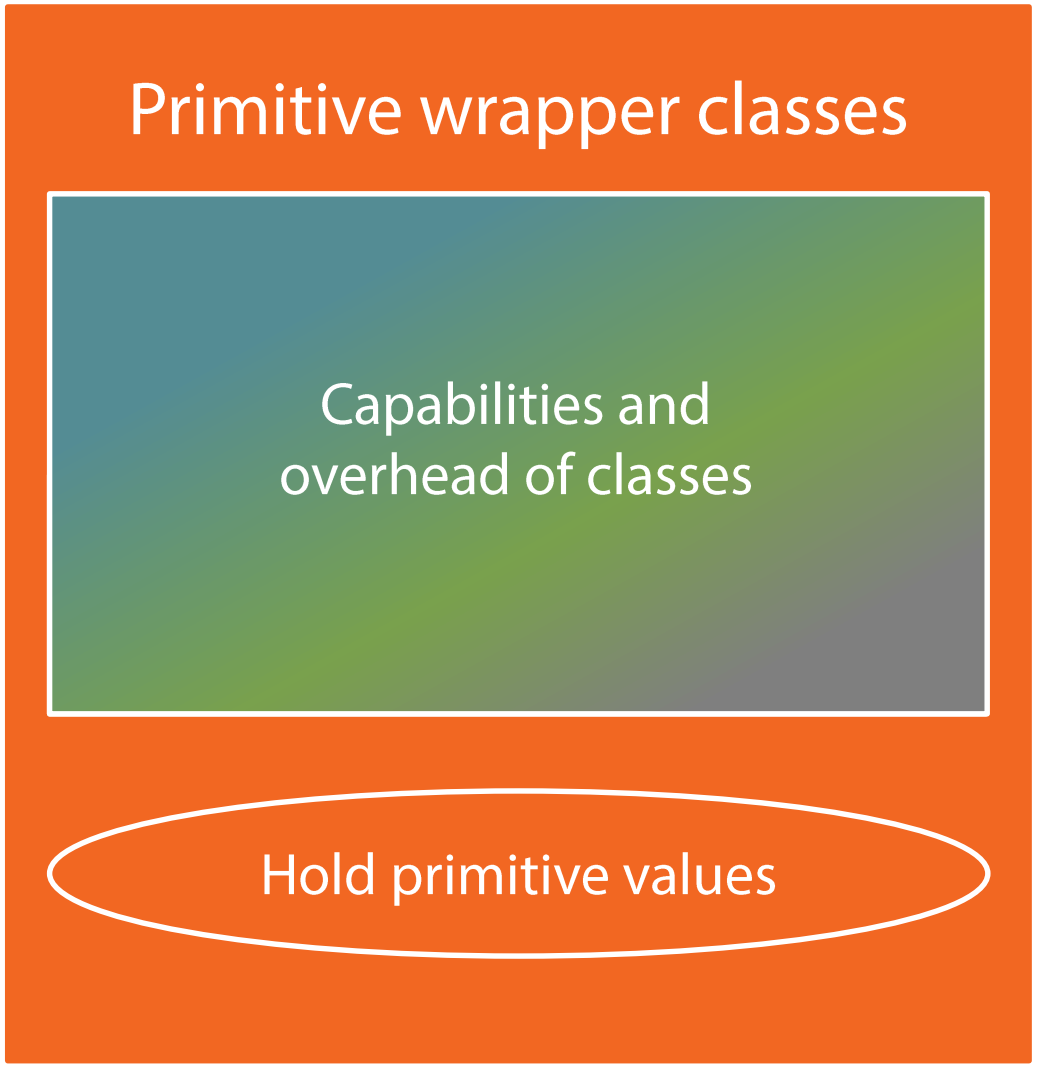
Classes provide convenience

- Common interaction through Object class
- Fields and methods specific to the type
- Incurs an overhead cost

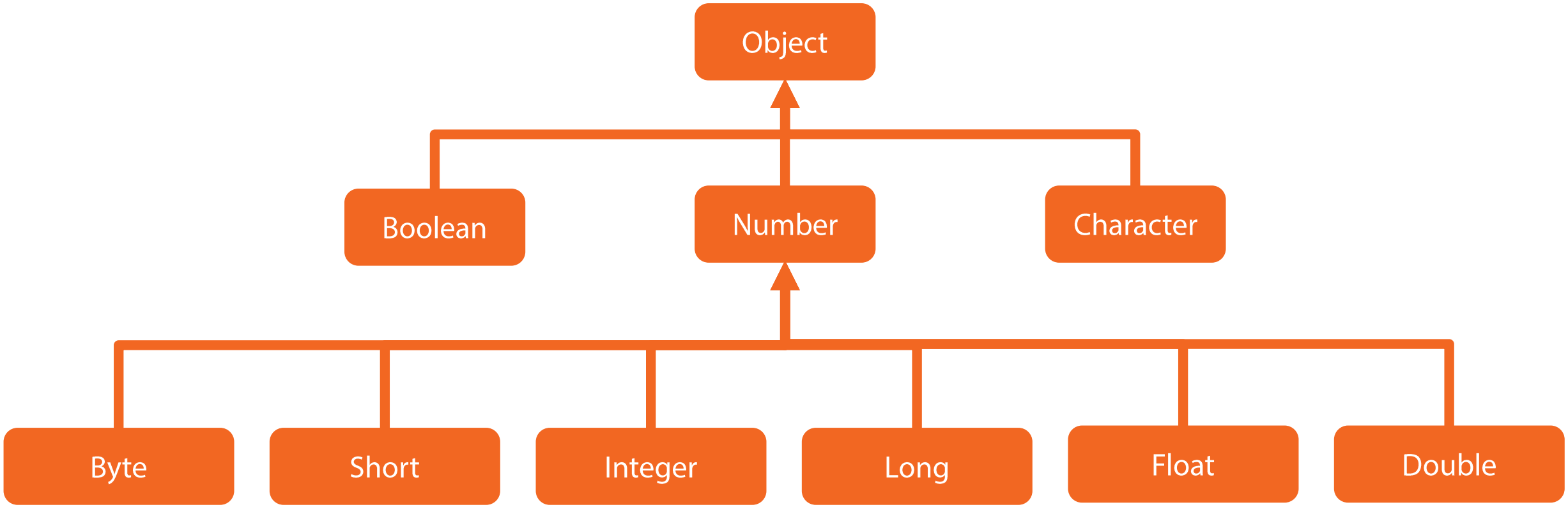
Primitives provide efficiency

- Cannot be treated as Object
- Cannot expose fields or methods
- Lightweight

Primitives Wrapper Classes



Primitive Wrapper Class Hierarchy



All wrapper class instances are immutable

Wrapper Class & Primitive Conversion

Java provides a number of ways to handle conversions

Common conversions handled automatically

Wrapper Class & Primitive Conversion

```
Integer a = 100;
int b = a;
Integer c = b;
```

Wrapper Class & Primitive Conversion

Java provides a number of ways to handle conversions

Common conversions handled automatically

Wrapper classes provide methods for explicit conversions

Primitive to wrapper
valueOf

Known as boxing

Wrapper to primitive
xxxValue

Known as unboxing

Wrapper Class & Primitive Conversion

```
Integer a = 100;
int b = a;
Integer c = b;

Integer d = Integer.valueOf(100);
int e = d.intValue();
Integer f = Integer.valueOf(e);

Float g = Float.valueOf(18.125f);
float h = g.floatValue();
```

Wrapper Class & Primitive Conversion

Java provides a number of ways to handle conversions

Common conversions handled automatically

Wrapper classes provide methods for explicit conversions

Primitive to wrapper
valueOf

Known as boxing

Wrapper to primitive
xxxValue

Known as unboxing

String to primitive
parseXxx

String to wrapper
valueOf

Wrapper Class & Primitive Conversion

```
Integer a = 100;
int b = a;
Integer c = b;

Integer d = Integer.valueOf(100);
int e = d.intValue();
Integer f = Integer.valueOf(e);

Float g = Float.valueOf(18.125f);
float h = g.floatValue();
```

```
String s = “87.44”;
double s1 = Double.parseDouble(s);
Double s2 = Double.valueOf(s);
```

Using Wrapper Classes

Treat as Object

```
Object[] stuff = new Object[3];
stuff[0] = new Flight();
stuff[1] = new Passenger(0, 2);
stuff[2] = 100;
```

Null references

```
public class Flight {
  Integer flightNumber;
  Character flightClass;
  // other members elided for clarity

  @Overrride
  public String toString() {
    if(flightNumber != null)
      return "Flight #" + flightNumber;
    else if(flightClass != null)
      return "Flight Class " + flightClass;
    else
      return "Flight identity not set ";
  }
}
```

Wrapper Class Members

Class	Select Members	Documentation
Byte Short Integer Long	MIN_VALUE, MAX_VALUE, bitCount, toBinaryString	http://bit.ly/javabyte http://bit.ly/javashort http://bit.ly/javainteger http://bit.ly/javalong
Float Double	MIN_VALUE, MAX_VALUE, isInfinite, isNaN	http://bit.ly/javafloat http://bit.ly/javadouble
Character	MIN_VALUE, MAX_VALUE, isDigit, isLetter	http://bit.ly/javacharacter
Boolean	TRUE, FALSE	http://bit.ly/javaboolean

Wrapper Class Equality

```
Integer i1000A = 10 * 10 * 10;
Integer i1000B = 100 * 10;

if(i1000A == i1000B)
   // do something

if(i1000A.equals(i1000B))
   // do something

Integer i8A = 4 * 2;
Integer i8B = 2 * 2 * 2;

if(i8A == i8B)
   // do something
```

FALSE

TRUE

TRUE

Boxing conversions that always return the same wrapper class instance	
Primitive Type	**Values**
int	-128 to 127
short	-128 to 127
byte	-128 to 127
char	'\u0000' to '\u00ff'
boolean	true, false

Final Fields

Marking a field as final prevents it from being changed once assigned

A simple final field must be set during creation of an object instance

Can be set with field initializer, initialization block, or constructor

Final Fields

```
public class Passenger {
  private final int freeBags;
  // other members elided for clarity

  public Passenger(int freeBags) {
     this.freeBags = freeBags;
  }

}
```

Final Fields

Marking a field as final prevents it from being changed once assigned

A simple final field must be set during creation of an object instance

Can be set with field initializer, initialization block, or constructor

Adding the static modifier makes a final field a named constant

Cannot be set by an object instance

Final Fields

```
public class Passenger {
  private final int freeBags;
  // other members elided for clarity

  public Passenger(int freeBags) {
     this.freeBags = freeBags;
  }

}
```

```
public class Flight {
  static final int MAX_FAA_SEATS = 550;
  private int seats;
  // other members elided for clarity

  public void setSeats(int seats) {
     if(seats <= MAX_FAA_SEATS)
       this.seats = seats;
     else
       // handle error
  }

}
```

Enumeration Types

- Enumeration types useful for defining a type with a finite list of valid values
  - Declare with enum keyword
  - Provide a comma-separated value list

```
public enum FlightCrewJob {
  Pilot,
  CoPilot,
  FlightAttendant,
  AirMarshal
}
```

```
public class CrewMember {
  private FlightCrewJob job;
  // other members elided for clarity

  public CrewMember(FlightCrewJob job) {
    this.job = job;
  }
  public void setJob(FlightCrewJob job) {
    this.job = job;
  }
}
```

```
CrewMember judy =
  new CrewMember(FlightCrewJob.CoPilot);

// . . .

Judy.setJob(FlightCrewJob.Pilot);
```

Demo
CalcEngine with More Data Type Capabilities

Summary

- String class stores an immutable sequence of Unicode characters
  - Implement toString method to provide conversion to a string
- StringBuilder class provides an efficient way to manipulate string values
- Primitive wrapper classes bring class capabilities to primitive values
  - Wrapper classes much less efficient than primitive types
- Final fields prevent a value from being changed once assigned
  - Simple final fields must be set during object instance creation
  - Static final fields act as named constants
- Enumeration types useful for defining a type with a finite list of values