CompSci 230
Software Construction

Java Implementation: Part 3 S1 2015

Agenda

- Topics:
  - Enum Types
  - Object: a superclass
  - Memory allocation
  - An OO description of Java's type system
- Reading, in The Java Tutorials:
  - Enum Types and Nested Classes pages, in the Classes and Objects Lesson.
  - Object as a Superclass page, in the Interface and Inheritance Lesson.

Enum Types

- “An *enum type* is a special data type that enables for [sic] a variable to be a set of predefined constants.
  - The variable must be equal to one of the values that have been predefined for it.
  - Common examples include
    - compass directions (values of NORTH, SOUTH, EAST, and WEST) and
    - the days of the week.
- “Because they are constants, the names of an enum type's fields are in uppercase letters.
- “… you define an enum type by using the `enum` keyword.
  - For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY
}
```

- “You should use enum types any time you need to represent a fixed set of constants.
  - That includes natural enum types such as the planets in our solar system and
  - data sets where you know all possible values at compile time—for example,
    - the choices on a menu,
    - command line flags, and so on.”

Example of **enum** usage; **switch** syntax

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) { this.day = day; }

    public void tellItLikeItIs() {
        switch (day) {
        case MONDAY:
            System.out.println("Mondays are bad.");
            break;
        case FRIDAY:
            System.out.println("Fridays are better.");
            break;
        case SATURDAY:
        case SUNDAY:
            System.out.println("Weekends are best.");
            break;
        default:
            System.out.println("Midweek days are so-so.");
            break;
        }
    }
```

Remainder of the **EnumTest** class

```
  public static void main(String[] args) {
      EnumTest firstDay = new EnumTest(Day.MONDAY);
      firstDay.tellItLikeItIs();
      EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
      thirdDay.tellItLikeItIs();
      EnumTest fifthDay = new EnumTest(Day.FRIDAY);
      fifthDay.tellItLikeItIs();
      EnumTest sixthDay = new EnumTest(Day.SATURDAY);
      sixthDay.tellItLikeItIs();
      EnumTest seventhDay = new EnumTest(Day.SUNDAY);
      seventhDay.tellItLikeItIs();
   }
}
```

Output:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

Importing static members of a class

- Importing the static members of an enum may significantly reduce "code clutter", because you won't have to fully qualify their names.
  - However a static import may decrease readability, if the reader has trouble figuring out "which class defined this member."

```
package enumtest;

public class EnumTest {
    enum Day {  // note: this is a "nested inner class"
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    }
    public static void main(String[] args) {
        (new EnumTest(Day.MONDAY)).tellItLikeItIs();
        (new EnumTest(Day.WEDNESDAY)).tellItLikeItIs();
        (new EnumTest(Day.FRIDAY)).tellItLikeItIs();
        (new EnumTest(Day.SATURDAY)).tellItLikeItIs();
        (new EnumTest(Day.SUNDAY)).tellItLikeItIs();
    }
```

Importing static members of a class

- Importing the static members of an enum may significantly reduce "code clutter", because you won't have to fully-qualify their names.
  - However a static import may decrease readability, if the reader has trouble figuring out "which class defined this member."

```
package enumtest;
import static enumtest.EnumTest.Day.*;
public class EnumTest {
   enum Day {  // note: this is a "nested inner class"
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
   }
   public static void main(String[] args) {
      (new EnumTest(MONDAY)).tellItLikeItIs();
      (new EnumTest(WEDNESDAY)).tellItLikeItIs();
      (new EnumTest(FRIDAY)).tellItLikeItIs();
      (new EnumTest(SATURDAY)).tellItLikeItIs();
      (new EnumTest(SUNDAY)).tellItLikeItIs();
   }
```

Switch: semantics

```java
public void tellItLikeItIs() {
    switch (day) {
    case MONDAY: // Each case is labelled by one (or more) values
                 // in the range of the switch variable (or expression)
        System.out.println("Mondays are bad.");
        break;
    case FRIDAY: // We don't have to write Day.FRIDAY, because each case
                 // label is a value of the same type as the switch expression.
        System.out.println("Fridays are better.");
        break; // Case statements "flow-through" if there's no break!
    case SATURDAY: // Note the "flow-through" for this case
    case SUNDAY:
        System.out.println("Weekends are best.");
        break;
    default:  // You'll get a runtime error if there's no matching case
                 // but default matches any value of the switch expression
        System.out.println("Midweek days are so-so.");
        break;
} } } // https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html
```

`Object` – this is what your classes extend!

- When you define a class in Java without specifying what class you're extending, you're actually extending the `Object` class.
  - The Object as a Superclass lesson briefly discusses six of the methods which your classes inherit from `Object`.
- `protected Object clone() throws CloneNotSupportedException`
  - Creates and returns a copy of this object.
- `public boolean equals(Object obj)`
  - Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`
  - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class getClass()`
  - Returns the runtime class of an object.
- `public int hashCode()`
  - Returns a hash code value for the object.
- `public String toString()`
  - Returns a string representation of the object.

Overriding `toString()`

- "You should always consider overriding the `toString()` method in your classes.
- "The `Object`'s `toString()` method returns a `String` representation of the object, which is very useful for debugging.
  - The `String` representation for an object depends entirely on the object, which is why you need to override `toString()` in your classes."

http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

Overriding `equals()`

- The `equals()` method compares two objects for equality and returns true if they are equal.
  - The `equals()` method provided in the `Object` class uses the identity operator (==) to determine whether two objects are equal.
  - For primitive data types, this gives the correct result.
  - For objects, however, it does not.
- The `equals()` method provided by `Object` tests whether the object references are equal—that is, if the objects compared are the exact same object.
  - To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the `equals()` method.

Example: overriding equals()

```
public class Book {
   String Title;
   String Author;
   String Publisher;
   String Year;
   String ISBN;
   ...
   public boolean equals(Object obj) {
      if (obj instanceof Book)
         return ISBN.equals(((Book) obj).getISBN());
      else
         return false;
      }
   }
   ...
 }
```

Example: overriding equals()

```
public class Book {
    String Title;
    String Author;
    String Publisher;
    String Year;
    String ISBN;
    ...
    @Override  // This annotation suppresses error messages from the
    // Java compiler, and it improves readability.
    public boolean equals(Object obj) { // Note: the same signature
        // as Object.equals(), but with a different implementation
        if (obj instanceof Book)
            return ISBN.equals(((Book) obj).getISBN());
        else
            return false;
        }
    }
    ...
```

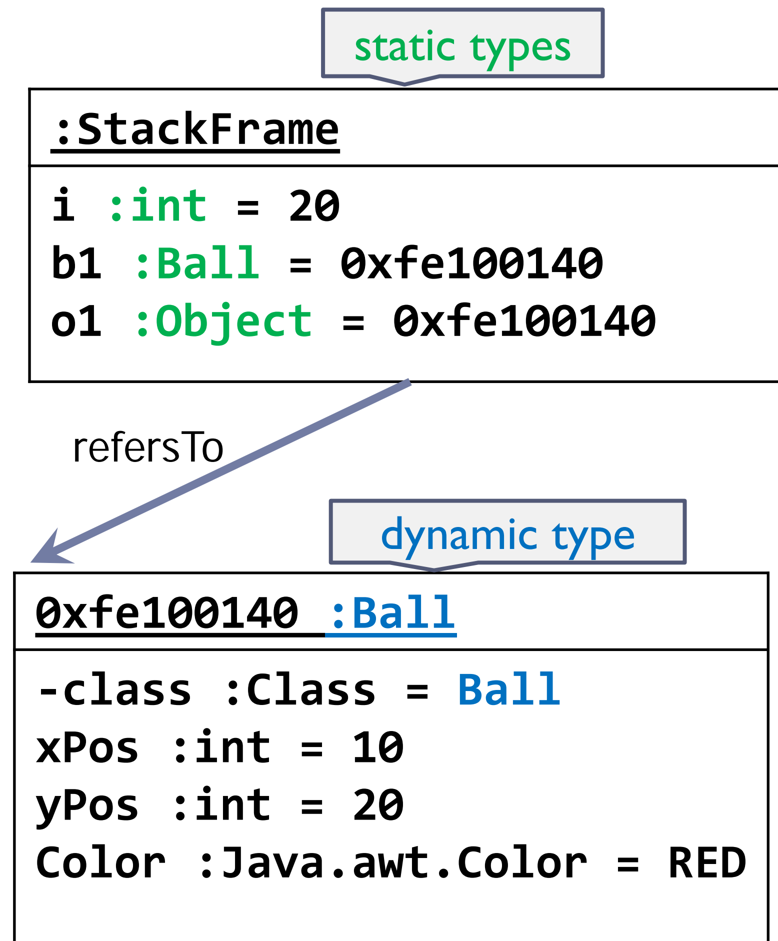
Example: testing an overridden equals()

```
    public static void main(String[] args){
        Book firstBook  = new Book("0201914670");
        Book secondBook = new Book("0201914670");
        if (firstBook.equals(secondBook)) {
            System.out.println("equivalent objects");
        } else {
            System.out.println("non-equivalent objects");
        }
        if (firstBook == secondBook){
            System.out.println("two references to the same object");
        } else {
            System.out.println("references to different objects");
        }
    }
}
```

Memory Allocation

```
int i = 20;
Ball b1
   = new Ball( 10, i, Color.RED );
Object o1 = b1;
```

- Recall: we use a reference variable to refer to instances of a class.
  - The value in a reference variable is, essentially, a pointer to an object.
    - A special value (`null`) indicates that there is no object corresponding to this reference.
    - The runtime system (the JVM) interprets a reference value as an index into a **heap**.
    - The **`new`** operator allocates sufficient memory on the heap to store all of the fields of an object of the requested type.
  - Formally: the range of allowable values for a reference variable is its reference type.
    - The reference type of **`o1`** is **`Object`**. This means it can point to any instance of **`Object`**, or to any instance of any subclass of **`Object`**.
  - Java also has primitive variables.
    - These have a primitive type, such as **`int`**.
    - They don't refer to objects.
  - In Java, a reference type is a static type, and a primitive type is also a static type.
    - Static types are determined by a static analysis of the program text.
    - A reference variable has a dynamic type, which is determined at runtime by the type of the object it is referring to.



A model of Java’s type system (for reference)

value
Variable
JavaValue
1
type
1
1
declaredType
type
JavaType
1
PrimitiveValue
ReferenceValue
pointsTo
1
JavaObject
ReferenceType
PrimitiveType
NullReference
ClassInstance
Array
InterfaceType
ClassType
ArrayType
*
conformsTo

Source: Kollman, R. and Gogolla, M., “Capturing Dynamic Program Behaviour with UML Collaboration Diagrams”, *Proc. CSMR*, 2001.

Review

- Topics in this set of slides:
  - Enum types
  - `Object.toString()`, `Object.equals()`
  - Memory allocation
  - An overview of Java's type system

- **End of Theme A: The OO Programming Paradigm**
  - We took a top-down approach: use-case analysis → class design → implementation.
  - You learned the fundamentals of OO design theory
    - You are *starting* to understand type systems
    - You have a basic understanding of Java development (JDK, Eclipse) and Java runtime (JRE).
    - You understand the difference between the static type of a reference variable (defined by its declaration) and its dynamic type (defined by its current value)
  - You have a basic proficiency in program analysis, OOD, and Java implementation
    - You are able to "learn more", if necessary, by reading, thinking, and experimenting.