Object References

25 Jan 2011
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Outline for today

- Keyword static
- Scope
- References and constructors
- Wrapper classes for primitive types
- Date and DateFormat
- Unit testing with JUnit

static keyword

- public static void main(String args[]) {
- static keyword: class attribute
  - Shared by all instances of this class
    - vs. instance attribute: separate for each object
  - Exists before class is instantiated
    - Invoke class methods as: ClassName.method()
- Running a class vs. instantiating a class:
  - Run a class from JRE: java MyClass
    - No instances made, just MyClass.main() invoked
  - Instantiating: new MyClass()
    - Constructor is run, main() is not run

static import

- import static java.lang.Math.*;
- Import all static members of a class
- Brings static variables/methods into current namespace:
  - sqrt(36.0); instead of Math.sqrt(36.0);
  - log(E); instead of Math.log(Math.E);
- Can also bring in one particular member:
  - import static java.lang.Math.sqrt;

Scope vs. duration

- The duration (lifetime) of an identifier is the runtime period when it exists in memory
  - Automatic duration
    - Local variables disappear when block finishes
  - Static duration
    - As long as the object/module/program exists
- The scope of an identifier is the lexical extent where it can be referenced
  - Block scope
  - Class scope

Scope example

```
public class ScopeExample {
    int numApples = 0;          // class scope
    public void listApples() {
        int counter = 0;        // block scope
    }
}
```

- numApples is an instance variable with class scope: accessible to all methods of this class
- counter is a local variable with block scope: not accessible outside the listApples() method

References and copy construct.

- Straight assignment of objects merely makes an alias (reference):
  - **Student joe = new Student("Joe Smith");**
  - **Student jane = joe; // alias**
- How to implement deep copy? Copy constructor
  - Overload constructor to accept another object of the same type:
    - **public Student(String name) { … }**
    - **public Student(Student other) {// copy constr.**

      name = other.name;
  - Using the copy constructor:
    - **Student jane = new Student(joe);**

Overloaded constructors

- In summary, any well-designed class that stores data (attributes) ought to have:
- Private (or protected) attributes
- Public set/get methods as appropriate
- Several overloaded constructors:
  - Using args to initialize attributes
  - With fewer or no args (using default values)
  - With a single object of same type (copy constructor)
- Other public methods for desired functionality

Null reference

- To create an object, first declare it:
  - **Student joe;**
- Then create a new instance:
  - **joe = new Student("Joe Smith");**
  - **joe.getName();**
- Before an object is assigned, it has value null
- When accepting objects as function parameters, check to ensure they are not null references:
  - e.g., in the copy constructor:
    - **public void Student(Student other) {**
      if (other != null) { ...

Initializing object attributes

- Set default values for attributes in constructor:
  - **public class Student {**
    - **String name;**
    - **Date birthdate;**
    - **public Student() {**
      name = "Joe"; birthdate = new Date(); }
- Or initialize in declaration (only for non-objects):
  - **public class Student {**
    - **String name = "Joe";**
    - **public Student() {**
      birthdate = new Date(); }

Wrapper classes

- Java is OO: “everything is an object”
  - What about primitive types: int, char, etc.?
- Wrapper classes: Integer, Character, Double, ...
  - Auto-boxing/unboxing:
    - **Integer numApples = 15;**
    - **int numA = numApples;**
- Static methods to convert to/from Strings:
    - **int numA = Integer.parseInt(“12.58”);**
    - **Double.toString(12.58);**
- Can define .toString() for any class (Py: __str__)

Date

- Get the current date and time:
  - **import java.util.Date;**
  - **Date now = new Date();**
  - Stores number of milliseconds since midnight 1Jan1970 UTC (the "epoch")
- Format it in current timezone for display:
  - **import java.text.SimpleDateFormat;**
  - **DateFormat fmt = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");**
  - **fmt.format(now);**

DateFormat

- The date is universal, same across the globe
- How it is formatted depends on local timezone
- SimpleDateFormat creates a DateFormat formatter object that can convert between the Date (universal) and a string (localized)
  - Date → String: **fmt.format(date);**
  - String→Date: **fmt.parse("27 Jan 2010 15:00")**
- More info: see JavaSE documentation: Date, DateFormat

Class design: testbed

- Main class (Student): attribs, methods, constr.
  - **public class Student {**
    - **String name;**
    - **short ID;**
    - **public Student() {…}**
- Testbed class (StudentTest):
  - main() and other methods create instances of Student and call methods:
    - **public class StudentTest {**
      - **public static void main(String args[]) {**
        - **Student s1 = new Student();**
        - **s1.setName(“Joe Smith”);**

Unit testing with JUnit4

- Create a separate class to hold your testcases
  - **import org.junit.Test;**
  - **import static org.junit.Assert.*;**
- Each test case is a method: declare with @Test
  - Create some objects from your class
  - Call some methods on your objects
  - Make assertions: assertEquals(a, b);
- Run the test cases:
  - In Eclipse: New → JUnit Test Case, and Run
  - org.junit.runner.JUnitCore.runClasses(TestClass1.class);