Lecture 15-16: Intra-Class Variable Access, Method Calling, and Method Overloading

Last time:
1. Project #3 due 3/05
2. Exam #1 coming 3/07
3. Copy Constructors
4. APIs, comments and documentation

Today:
1. Intra-Class Variable Access and Method Calling
2. Call Stack
3. Method Overloading
Accessing static/non-static Variables inside Methods

- **Example**
  ```java
  public class C {
      public int a = 0;
      public static int b = 1;
      public void f () { ...}
      public static void g () {...}
  }
  ```
  - Can body of `f` refer to `a`?
    - Yes
  - Can body of `f` refer to `b`?
    - Yes
  - Can body of `g` refer to `a`?
    - No
  - Can body of `g` refer to `b`?
    - Yes

- **What are the rules?**
  - A static method may refer only to its own local variables or static variables in its class body
  - A non-static method may refer to its own local variables and also static/instance variables in its class body
  - Public / private doesn’t matter in this context
Methods Calling Methods

- Four kinds of methods so far
  - public non-static
  - public static
  - private non-static
  - private static

- Question: What kind of methods can call other kinds of methods?
- The rules are similar to those for accessing static/non-static variables!
Examples

- Assume following class specification
  ```java
  public class C {
      public void f () { ...}
      public static void g () {...}
      private void h () {...}
      private static void i () {...}
  }
  ```
- Can:
  - Body of `f` call `g`? Yes
  - Body of `g` call `f`? No
  - Body of `h` call `g`? Yes
  - Body of `f` call `h`? Yes
  - Body of `h` call `f`? Yes
  - Body of `i` call `g`? Yes
  - Body of `i` call `h`? No
Accessing static/non-static methods

- Recall:
  - Static methods are owned by class
  - Other methods are owned by objects
- So for methods in same class:
  - Non-static methods may call static or non-static methods
  - Static methods may call other static methods
  - Public / private doesn’t matter in relation to who may call whom
Stacks in Computer Science

- A stack is a data structure ("device" for holding values)
- Three operations on a stack
  - push: add a new value into the stack
  - pop: remove the most recently added value still in stack
  - top: return the most recently added value in stack
- Think: stack of plates in a restaurant
  - push = put new plate on top
  - pop = remove top plate
  - top = look at top plate
Why Is It Called “The Stack”? 

- Stack is part of main memory used to hold variables and their values.
- When a variable is referenced, its value is found by looking in stack.
- When methods are invoked, temporary additions (“stack frames”) are made to stack before body of method is executed.
Example

- S.push (3);
- S.push (4);
- S.top == ??
  4
- S.pop ();
- S.push (5);
- S.top == ??
  5
What Do “Stacks” Have To Do with “The Stack”

- Main memory: “stack” is short for “call stack”
- “call stack” = information for processing method calls
- Contents of call stack: stack frames!
- At beginning of method call, new stack frame pushed onto call stack
- After body of method has finished executing, stack is “popped”
Example

- public class C {
  int f (int i) {
    return (i+1);
  }

  int g (int i) {
    return (1+f(i+1));
  }
}

- What is printed by:
  System.out.println(C.g(1));
- 4
Local Variables and the Stack

- Local variables are treated like method arguments
  - When declared, they are added to current stack frame
  - When stack frame is popped, those local variables disappear
- Whenever a new block is introduced, so is a new stack frame
  ```java
  int x = 3;
  ...
  }
  ```
  - This introduces a new stack frame associating `x` with `3`
  - When the block is finished, the stack frame goes away
Initialization of Instance Variables

- Instance variables are assigned values even if none explicitly provided
  - Primitives are set to 0 / false
  - References are set to null (i.e. point to nothing)
  - Consider:
    ```java
    public class C {
      public int x;
    }
    ...
    C cObj = new C ();
    System.out.println (cObj.x);
    ```
  - 0 is printed

- Don’t rely on this in your programs!
  - Compilers famously get this wrong
  - Better practice:
    - Set default values when variable is declared in class
      ```java
      public int x = 0;
      ```
    - In constructors, supply alternative values as desired
Initialization of Local Variables

- Local variables are *not* initialized.
  - Java checks if variables are uninitialized
  - If use of uninitialized variable is possible, the Java compiler (and Eclipse also) complain
- Example:
  ```java
  public void f() {
    int x;
    int z = x + 3;  // won't compile
  }
  ```
- What about static variables?
  - They are treated like instance variables
  ```java
class C {
  public static int x;
}
```
  - Expression `C.x` evaluates to 0
- Good practice: always initialize variables explicitly
  - Avoids compiler errors (local variables)
  - You don’t have to remember different treatments of local, instance variables
  - You don’t run the risk of a misbehaving compiler
Method Overloading

- **Overloading**: declaration of two methods in the same class with same name
  - Java allows overloading
  - The different methods must have different argument lists
  - e.g
    ```java
    public class C {
        int f ();
        int f (int f);  // OK
    }
    ```

- **Why use overloading?**
  - To allow similar operations to have the same name
  - We have seen overloading before with *constructors*
Example

- Recall Student class toString method
  ```java
  // The method for converting dates to strings
  public String toString () {
      return "name: " + name + ", id: " + id + ", tokens: " + tokenLevel;
  }
  ```

- We could write a new print method for the Student class as follows:
  ```java
  // Method for printing dates to System.out
  public void print () {
      System.out.print (toString ());
  }
  ```

- May want another print method that allows specification of surrounding text
  ```java
  public void print (String pre, String post){
      System.out.print (pre);
      print ();
      System.out.print (post);
  }
  ```

- Java allows this

- What is output of

```java
Student s = new Student ();
s.print ("The date is ", ".");
```
When Is Overloading Allowed?

- **Terminology:**
  - **Prototype:** I/O types, behavior (public, static, etc.) for method
    ```java
    public static void f(int x, float y)
    ```
  - **Signature:** Input types for method
    ```java
    f(int x, float y)
    ```

- You can only overload a methods if the signatures are different:
  ```java
  void f (int)      // fine
  int f (float)    // fine
  ```

- Just having different return types is not enough:
  ```java
  void f (int)      // not fine
  int f (int)      // not fine
  ```

- **Why?**
- Because compiler can’t always tell which f to call just based on return type
  ```java
  f(3);  // Which f should be run?
  ```