CS 61B Data Structures and Programming Methodology

July 8, 2008
David Sun
Announcement

- Midterm 1 is tomorrow 11:00am – 1:00pm.
- Everything up to and including today’s material can appear on the test.
The finally clause is executed regardless whether an exception is thrown or not. So in the above, the code always returns 3.
Exceptions Inside `catch` and `finally`

- An exception thrown in a `catch` clause will proceed as usual, but the `finally` clause will still get executed before the exception goes on.
- An exception thrown in a `finally` clause replaces the old exception, and the method ends immediately.
- You can also put a `try` clause inside a `catch` or `finally` clause.
Inheritance with Exceptions

class parent {
    void f() { . . . }
}

//Compiler Error
class child extends parent {
    void f() throws IOException { . . . }
}

• Why?
    – Substitution rule.

• Rule: In the overriding method in the child class cannot expand the set of checked exception thrown by the original method in the parent class.
Nested Classes

• A nested class is a class that is defined inside another class.

```java
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

• Two types:
  – Static
  – Non-static
Static Nested Class

• A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class.

• Why bother?
  – The nested class might be used only in the implementation of the outer class -- avoid name clashes or “pollution of the name space” with names that will never be used anywhere else.
  – The nested class is conceptually “subservient” to the outer class -- group classes and provide better encapsulation.
class OuterClass {
    private SomeClass outerField;
    private void outerMethod() { . . . }
    static class StaticNestedClass { ... }
}

• **Like static fields and methods, StaticNestedClass is associated with OuterClass**, i.e. to create an object of StaticNestedClass:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

• **Like static class methods, StaticNestedClass cannot refer to the private instance field outerField or the method outerMethod() defined in OuterClass.**
Example

//List.java
public class List {
    private ListNode head;
    private int size;

    public List() {
        head = null;
        size = 0;
    }
    ...
}

//ListNode.java, global //visibility
public class ListNode {
    int item;
    ListNode next;
}

//now only visible inside List
private static class ListNode {
    int item;
    ListNode next;
}
Inner classes

• An inner class is associated with an **instance** of its enclosing class so it has direct access to that object's methods and fields.

```java
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();

compare with

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```
Example

class Bank {
    private int count;
    private void connectTo( . . ) { . . . }

    public class Account {
        public Account () { count ++; }
        public void call (int number) {
            Bank.this.connectTo( . . . );
            . . .
        }
    }
}

Bank e = new Bank( . . );
Bank.Account a1 = e.new Account( . . . );
Bank.Account a2 = e.new Account( . . . );

Note: the outer reference is invisible in the definition of the inner class. You can’t explicitly refer to outer.
Cloning

• Copying a variable only copies the reference, the original variable and the copy refer to the same object, so changing one affects the other.

```java
Point original = new Point(1, 3);
Point copy = original;
copy.x  = 3;
what’s in original.x ?
```

```
original = 

x = 3
y = 3
```

```java

copy = 
```
Cloning

- If you want to make a copy of the object that begins its life being identical to original but whose state can diverge over time:

```java
Employee copy = original.clone();
copy.x = 3; // original unchanged
```

```
original = {
  x = 1
  y = 3
}
```

```
copy = {
  x = 3
  y = 3
}
```
clone()

• The clone method is a protected method of Object.
  – So clone() can only be called in side the class Point.
  – The user of Point can’t make the call original.clone();

• Default implementation in Object is a shallow copy:
  – It will make a field-by-field copy.
  – If all the fields are basic types, then field copying is fine.
  – If some of the fields contain references to subobjects, then field copying gives your another reference to the subobject.
Does Shallow Copy Matter?

• It depends...
• Sharing is “safe” if:
  – If the shared subobject object is immutable, e.g. a String
  – If the subobject remain constant throughout the lifetime of the object, with no mutators touching it.
• But frequently you want a deep copy.
Redefining clone

• How:
  1. Have the class implement the Clonable interface.
  2. Redefine the clone method with the public access modifier.

```java
class Employee implements Clonable {
    . . .
    //raise visibility to public, change return type
    public Employee clone() throws CloneNotSupportedException {
        return (Employee) super.clone;
    }
}
```
class Employee implements Clonable
{
    ...

    public Employee clone() throws CloneNotSupportedException {
        // call object.clone()
        Employee theClone = (Employee) super.clone();

        // clone mutable fields
        theClone.birthDay = (Date)birthDay.clone();

        // clone other mutable field
        ...

        return theClone;
    }
}
Another Example

public class List implements Cloneable {

    public List clone() throws CloneNotSupportedException {
        //same as return super.clone();
        List l = new List();
        l.head = head;
        l.size = size;
        return l;
    }
}

Another Example

public class List implements Cloneable {
    public List clone() throws CloneNotSupportedException {
        List l = (List) super.clone();
        l.head = head.clone();
    }
}

public class ListNode implements Cloneable {
    public ListNode clone() throws CloneNotSupportedException {
        ListNode theClone = (ListNode) super.clone();
        theClone.next = theClone.next.clone(); // recursive
    }
}
Cloning in the Subclass

• The `clone` method in the superclass may be sufficient if the additional state in the subclass are all primitive types.

• If data fields in the subclass contain object reference, then you may want to redefine `clone()`.
cloneable Interface

• Cloneable doesn’t specify the clone method, clone is a method defined in Object.

• In this case the interface acts as a tag, indicating that the class designer understands the cloning process.

• Cloning is dangerous if care is not taken,
  – The clone method of the object class threatens to throw a CloneNotSupportedException if clone is called on a class that doesn’t implement the cloneable interface.
for each Loop

- Introduced in Java 5.0 so you can loop over the elements in an array without having to fuss with index values:
  
  for (variable : array) statement

- Examples:

  ```java
  int[] array = {7, 12, 3, 8, 4, 9};
  for (int i : array) { //for each integer in array
    System.out.print(i + " ");
  }

  String concat = "";
  for (String s : stringArray) { //for each string in StringArray
    concat = concat + s;
  }
  ```
Enum

• Before Java 5, the standard way of representing enumerating was using int:

    public static final int SEASON_WINTER = 0;
    public static final int SEASON_SPRING = 1;
    public static final int SEASON_SUMMER = 2;
    public static final int SEASON_FALL = 3;

• Problems:
  – Not type safe: We can pass in any integer or add any two seasons
  – No namespace: we need to prefix the constant with a string (SEASON_) to avoid collision.
  – Print values are uninformative: you get a number
enum

• An `enum` type is a class whose `fields` consist of a fixed set of constants.

• Examples:
  – compass directions (values of NORTH, SOUTH, EAST, and WEST)
  – days of the week.

• Define an enum type by using the `enum` keyword:

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

- The enum class body can include methods and other fields.
- The compiler automatically adds some special methods when it creates an enum.
  - `values` returns an array containing all of the values of the enum in the order they are declared.

```java
for (Day d : Day.values()) {
    . . .
}
```
public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;

    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public String toString() {
        return rank + " of " + suit;
    }

    ...
}
Reading

• Head First Java:
  – pp 376 – 386, 671 – 672