Java
Odds & Ends

Lecture 25
CS211 – Fall 2005

Announcements

- Final Exam
  - Wednesday, 12/14
  - 9:00-11:30am
  - Uris Aud
- Review Session
  - Sunday, 12/11
  - 1:00-2:30pm
  - Kimball B11
- Check your final exam schedule!
- For exam conflicts:
  - Notify Kelly Patwell (patwell@cs.cornell.edu)
  - You must provide
    - your entire exam schedule
    - include the course numbers
- Definition of exam conflict:
  - Two exams at the same time or
  - *Three* or more exams within 24 hours

GUI Drawing & Painting

GUI Drawing & Painting

- For a drawing area:
  - Extend JPanel and override the method **public void paintComponent(Graphics g)**
- **paintComponent** must contain the code to completely draw *everything* in your drawing panel
- Note that **paintComponent** is *never* called directly
  - It is *requested* via a call to repaint
    Example: **myDrawingPanel.repaint();**

Java Graphics

- The Graphics class has methods for colors, fonts, and various shapes and lines
  - **setColor (Color c)**
  - **drawOval (int x, int y, int width, int height)**
  - **fillOval (int x, int y, int width, int height)**
  - **drawLine (int x1, int y1, int x2, int y2)**
  - **drawString(String str, int x, int y)**
- Take a look at
  - java.awt.Graphics (for basic graphics)
  - java.awt.Graphics2D (for “more sophisticated” control)
  - the 2D Graphics Trail (http://java.sun.com/docs/books/tutorial/2d/index.html)

Exceptions

Exceptions

- Exceptions are usually thrown to indicate that something bad has happened
  - IOException on failure to open or read a file
  - ClassCastException if attempted to cast an object to a type that is not a supertype of the dynamic type of the object
  - NullPointerException if tried to dereference null
  - ArrayIndexOutOfBoundsException if tried to access an array element at index i < 0 or ≥ the length of the array

Handling Exceptions

- Exceptions can be caught by the program using a `try/catch` block
- `catch` clauses are called *exception handlers*

```
Integer x = null;
try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
}
```

Defining Your Own Exceptions

- An exception is an object (like everything else in Java)
  - You can define your own exceptions and throw them

```
class MyOwnException extends Exception {}

...

if (input == null) {
   throw new MyOwnException();
}
```

The **throws** Clause

- In general, any exception you throw must be either *declared* in the method header or *caught*

```
void foo(int input) throws MyOwnException {
  if (input == null) {
    throw new MyOwnException();
  }
  ...
}
```

- Note: throws means "can throw", not "does throw"
- Subtypes of RuntimeException do *not* have to be declared (e.g., NullPointerException, ClassCastException)
  - These represent exceptions that can occur during "normal operation of the Java Virtual Machine"

How Exceptions are Handled

- If the exception is thrown from *inside* a try/catch block with a handler for that exception (or a superclass of the exception), then that handler is executed
  - Otherwise, the method terminates abruptly and control is passed back to the calling method
- If the calling method can handle the exception (i.e., if the call occurred within a try/catch block with a handler for that exception) then that handler is executed
  - Otherwise, the calling method terminates abruptly, etc.
- If *none* of the calling methods handle the exception, the entire program terminates with an error message

Generic Types in Java 5.0

Generic Types in Java 5.0

- When using a collection (e.g., LinkedList, HashSet, HashMap), we generally have a single type T of elements that we store in it (e.g., Integer, String)
- Before 1.5, when extracting an element, had to cast it to T before we could invoke T's methods
- Compiler could not check that the cast was correct at *compile-time*, since it didn't know what T was
- Inconvenient and unsafe, could fail at *runtime*

- Generics in Java 1.5 provide a way to communicate T, the type of elements in a collection, to the compiler
- Compiler can check that you have used the collection consistently
- Result is safer and more-efficient code

Example

old

```
//removes 4-letter words from c
//elements must be Strings
static void purge(Collection c) {
  Iterator i = c.iterator();
  while (i.hasNext()) {
    if (((String)i.next()).length() == 4)
      i.remove();
}}
```

new

```
//removes 4-letter words from c
static void purge(Collection<String> c) {
  Iterator<String> i = c.iterator();
  while (i.hasNext()) {
    if (i.next().length() == 4)
      i.remove();
}}
```

Another Example

old

```
HashMap grades = new HashMap();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = (Integer)grades.get("John");
System.out.println(x.intValue());
```

new

```
HashMap<String,Integer> grades =
  new HashMap<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

Type Casting

- In effect, Java inserts the correct cast automatically, based on the declared type
- In this example, grades.get("John") is automatically cast to Integer

```
HashMap<String,Integer> grades =
  new HashMap<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

An Aside: Autoboxing

- Java 5.0 also has autoboxing and auto-unboxing of primitive types, so the example can be further simplified

```
HashMap<String,Integer> grades =
  new HashMap<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

```
HashMap<String,Integer> grades =
  new HashMap<String,Integer>();
grades.put("John", 67);
grades.put("Jane", 88);
grades.put("Fred", 72);
System.out.println(grades.get("John"));
```

Using Generic Types

- <T> is read, “of T”
  - For example: Stack<Integer> is read, “Stack of Integer”
- The type annotation <T> informs the compiler that all extractions from this collection should be automatically cast to T
- Specify type in declaration, can be checked at compile time – can eliminate explicit casts

Advantage of Generics

- Declaring **Collection<String> c** tells us something about the variable c (i.e., c holds only Strings)
  - This is true wherever c is used
  - The compiler checks this and won't compile code that violates this
- Without use of generic types, explicit casting must be used
  - A cast tells us something the programmer *thinks* is true at a single point in the code
  - The Java virtual machine checks whether the programmer is right only at *runtime*

Subtypes

Stack<Integer> is *not* a subtype of **Stack<Object>**

```
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
Stack<Object> t = s; //gives compiler error
t.push("bad idea");
System.out.println(s.pop().intValue());
```

However, **Stack<Integer>** *is* a subtype of **Stack** (for backward compatibility with 1.4.2)

```
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
Stack t = s; //compiler allows this
t.push("bad idea");
System.out.println(s.pop().intValue());
```

Programming Generic Types

```
public interface List<E> { //E is a type variable
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

- To use the generic type declaration **List<E>**, supply an *actual type* argument, e.g., **List<Integer>**
- All occurrences of the formal type parameter (**E** in this case) are replaced by the actual type argument (**Integer** in this case)

Wildcards

old

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  while (i.hasNext()) {
    System.out.println(i.next());
}}
```

bad

```
void printCollection(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

good

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

Bounded Wildcards

```
static void sort(List<? extends Comparable> c) {
  ...
}
```

- Note that if we declared the parameter **c** to be of type **List<Comparable>** then we could not sort an object of type **List<String>** (even though String is a subtype of Comparable)
  - Suppose Java treated **List<String>** as a subtype of **List<Comparable>**
  - Then, for instance, a method passed an object of type **List<Comparable>** would be able to store Integers in our **List<String>**
- Wildcards let us specify exactly what types are allowed

Generic Methods

Adding all elements of an array to a **Collection**

bad

```
static void a2c(Object[] a, Collection<?> c) {
  for (Object o : a) {
    c.add(o); //compile time error
}}
```

good

```
static <T> void a2c(T[] a, Collection<T> c) {
  for (T o : a) {
    c.add(o); //ok
}}
```

Some Generic Type Examples

```
class Simplex<V> extends AbstractSet<V> implements Set<V>
  …
  public Simplex (Collection<? extends V> collection)
  …
  public static <V> Set<Set<V>> boundary
         (Set<? extends Simplex<V>> simplexSet)

public class Triangulation<V> implements Iterable<Simplex<V>>
  …
  public Triangulation (Simplex<V> simplex)
  …
  public Iterator<Simplex<V>> iterator ()
```

For More Info on Generic Types

- See the online Java Tutorial for more information on generic types and generic methods
- The text also has a section (4.7) on this topic