Exceptions and Generics

Lecture 13
CS211 – Spring 2006

Announcements

- Prelim 1
  - Thursday 7:30-9, Olin 155 or Olin 255
- Review Session
  - Wednesday 3/8, 7:30-9pm & 9-10:30pm (both sessions are identical),
  - Kimball B11
- See cool projects at BOOM ("Bits On Our Minds") tomorrow, Upson 4th floor atrium

Completing partial specs

```
/** Returns the (approximate) square root of x. */
float sqr(float x) {
    float y = x/2;
    while (Math.abs(y*y - x) > x*1e-6) {
        y = (y + x/y)/2;
    }
}
```

- Partial function -- what happens when x < 0?

1. Add *requires* clause `Requires x >= 0`. Client's fault if violated.
2. Add a *checks* clause `Checks x >= 0`. Client's fault if violated, but implementation promises to check it anyway, stop program
3. Make it legal (make sqr *total*), Implementer's fault if broken
   - 3a: encode result as special value (e.g., null)
     - `@return -1 if x < 0 ... if (x < 0.0) return -1.0;`
     - Error-prone -- easy to forget to check. Avoid if possible.
   - 3b: throw an exception
     - Stops program cleanly if you forget, but expensive (in Java)…

Exceptions

- Exceptions are usually thrown to indicate that something bad has happened
- `throw new E(…)` throws an exception object of class E:
  - 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

- 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)
  - Built-in exceptions produced by ordinary Java operations
  - Avoid using them -- makes failure harder to predict

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 a *stack trace*

Generic Types in Java 5.0

- Collection (e.g., LinkedList, HashSet, HashMap), usually contains elements of a single type T (e.g., Integer, String)
- Before 1.5, had to cast elements to T before using 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 *run time*
- 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: safer, more-efficient code
- A.k.a. *parameterized types*
- Use & implement generics in A4!

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

```
SortedSet grades = new TreeSet();
grades.put(new Integer(67));
grades.put(new Integer(88));
grades.put(new Integer(72));
Integer x = (Integer)grades.first();
System.out.println(x.intValue());
```

new

```
SortedSet<Integer> grades =
  new TreeSet<Integer>();
grades.put(67);
grades.put(88);
grades.put(72);
int x = grades.first();
System.out.println(x);
```

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

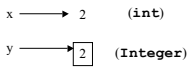
```
HashMap<String,Integer> grades =
  new HashMap<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

An Aside: Autoboxing

- Example also uses autoboxing and auto-unboxing of primitive types:

```
Integer x = new Integer(2);
int y = x.intValue();
```

```
Integer x = 2;
int y = x;
```



Using Generic Types

- <T> is read, "of T"
  - For example: Stack<Integer> is read, "Stack of Integer"
- T is a *type parameter*
- Think of Stack as a function that can be applied to a type to get a class.
- 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 *run time*

Programming Generic Types

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

public interface I
   E next();
   boolean hasNext
}
```

```
public interface List<Integer> {
   void add(Integer x);
   Iterator<Integer> iterator();
}
```

- 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)

Constrained type parameters

- What if you want to define a generic abstraction that only works on some types?

```
class List<T> {
   prettyPrint(Writer w) { … use prettyPrint on T… }
}
```

- Solution: add a *constraint/bound* on T:

Constraint interface

T is a subtype of PrettyPrintable

```
interface PrettyPrintable {
  void prettyPrint(Writer w);
}
class List<T extends PrettyPrintable> {
  prettyPrint(Writer w) {
       T elem; … elem.prettyPrint(); …
  }
}
List<Integer> -- illegal
List<Animal> -- legal if class Animal implements PrettyPrint {…}
```

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());
```

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

Type parameters in 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