COMP 401 – Fall 2017

Recitation 11: Generics, Abstract Classes, and Exceptions

Agenda

- Deep(er) dive into Generics
- Abstract Classes/Methods in Java
- Exceptions

- Code is in 20171105_recitation11_code_project.zip

Generics (very brief intro)

- Java 1.5 introduced Generics, which provide a degree of polymorphism
- Idea: I want to write a class or a method that operates on objects, but the implementation doesn't really care what specific objects they are.
- You could do this by simply operating on instances of `Object`; that's how we did things for a long time.
- However, I would like to take advantage of strong type-checking at compile time and avoid the proliferation of type-casting

Generics (brief intro, continued)

- Semantically, I want to declare that my class or method operates on some type T, but I don't care what the actual type of T is.
- Semi-trivial example:

```
public <T> T getArrayElement(T[] array, int index) {
  if(index >= 0 && index < array.length) {
    return array[index];
  } else {
    return null;
}
```

Syntax

- Type Parameter

```
class Foo<T> { ... }
class Foo2<T,U> { ... }
<T> void bar(T p1) { ... }
<T,U> List<T> baz(U[] us) { ... }
<V extends MyType> void blah(V p1) {...}
```

- Where the type parameter(s) (e.g., **`T`** and **`U`**, above) are Java identifier (typically single-character capital letters). These identify *types* in the code that follows.
- Type parameters may be bounded (as in the case of **`V`**, above), or may be unbounded.

Wildcard Parameters

- The type parameter may also be a “wildcard” expression; this may be used where the code:
  - Doesn’t care about the type of its argument(s) and
  - Doesn’t need to refer to the *name* of the type of its argument(s)
- Simple example:

```
boolean evenSize(List<?> list) {
      return (list.size() % 2) == 0;
}
```

- Wildcard type can also be bounded, e.g.:

```
boolean oddSize(List<? extends Foo> list) {
      return (list.size() % 2) == 1;
}
```

- For more details, see, e.g.,:

https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Type Erasure

- Java Generics is “syntactic sugar” in that:
  - It is a nicety provided by the *compiler* affording strong type safety and code clarity
  - It *does not* provide additional functional or expressive power (any program using Generics could be correctly implemented without Generics)
- Generics are implemented with *type erasure,* wherein:
  - If the type parameter is unbounded, it is replaced with Object, or if bounded, the appropriate bound
  - Casts are added, as appropriate, to guarantee type safety
  - (For completeness): “Bridge methods” are added to aid polymorphism
- Unlike, e.g., templates in C++, no new code is generated for each concrete parameterized type.

Generic Classes – Simple Examples

- Polymorphic Pair (Naïve implementation):

```
public class SimplePair<T> {
   protected T first;
   protected T second;

   public SimplePair(T first, T second) {
      this.first=first;
      this.second=second;
   }


   public T getFirst() {
       return this.first;
   }
   public void setFirst(T first) {
       this.first = first;
   }
}
```

Generic Classes – Simple Examples

- Polymorphic Pair (Better Implementation):

```
public class GeneralPair<T, U> {
   protected T first;
   protected U second;

   public GeneralPair(T first, U second) {
      this.first=first;
      this.second=second;
   }


   public T getFirst() {
       return this.first;
   }

   public void setFirst(T first) {
       this.first = first;
   }
}
```

Generic Methods

- Same idea as generic classes, but enclosing class need not be generic or use the same type(s)
- Example:

```
public static <T> boolean allNonNull(T[] array) {
    if(array==null) {
        return true;
    }

    for(T a: array) {
        if(null == a) {
                return false;
        }
    }

    return true;
}
```

Fancier Generic Methods

```
public static <T,U> List<GeneralPair<T,U>> toPairs(T[] ts, U[] us) {
    //Ignore sanity-check of inputs
    List<GeneralPair<T,U>> res = new ArrayList<GeneralPair<T,U>>(ts.length);
    for(int i=0; i<ts.length; i++) {
        res.add(new GeneralPair<T, U>(ts[i],us[i]));
    }

    return res;
}

public static <T extends Comparable<T>, U extends Collection<T>>
        void addIfLargest(T element, U coll) {

    for(T elt : coll) {
        if(element.compareTo(elt)<0) {
                return;
        }
    }

    coll.add(element);
}
```

Abstract classes and methods

- An abstract class is declared with the modifier `abstract`.
- An abstract method is a method declared with the modifier `abstract` and without an implementation.
- Abstract classes may or may not contain abstract methods.
- A class containing abstract methods must itself be declared as `abstract`.
- Abstract classes may not be instantiated
- Abstract classes may contain non-public, non-final, and non-static methods and data as well as non-abstract methods (having implementations).

What's this good for?

- An abstract class cannot be used/instantiated directly – a concrete implementation must be built which `extends` the abstract base class.
- Why would we want abstract classes/methods when we have interfaces?

- Typical use case: implementing a set of classes which all use a **common subset of data and/or logic**.
  - E.g., `AbstractMap` (JCF) – implementations include `HashMap`, `TreeMap`, and `ConcurrentHashMap`
  - E.g., `AbstractList`, `AbstractSequentialList`

Example

```
public abstract class AbstractQueue<T>
        implements SimpleQueue<T> {

    public AbstractQueue() {};
    public abstract void enqueue(T e);
    public abstract T dequeue();
    public abstract int size();
    ...
    public boolean isEmpty() {
        return this.size() == 0;
    }

    public void clear() {
        while (size() > 0) {
                this.dequeue();
        }
    }
}
```

Abstract classes

- Abstract classes can (and often should) implement one or more interfaces
- Often a good choice when implementing a family of classes that share common (partial) implementations and/or non-static member variables
- Interfaces should be used when a set of *unrelated* classes implement a common set of semantics
- Because abstract classes are implemented via inheritance, a given class can only extend a single abstract base class

Error Handling: Exceptions

- High-Level Semantics:
  - Within some code, something “bad” has happened
  - The code cannot continue and cannot return while fulfilling its return semantics
  - By design, the problem should not or otherwise cannot be handled within the local scope; handling is delegated to the caller(s)
  - A caller *may* wish to detect and possibly handle such errors in code it calls or may wish to delegate such handling to its caller(s)
  - Additionally, some problems may be relatively common and result from, say, invalid data or user input while other problems are “unexpected” and may be more serious
  - We don’t want to burden callers with having to deal with every possible error condition that could occur within our code, but we may wish to force them to handle or explicitly not handle certain conditions.

Exceptions

- Semantics:
  - When an error occurs within some method and the code does not wish to or cannot handle it, an *exception* is *thrown*
  - Execution of the method halts
  - The exception propagates up the stack until:
    - It reaches the top, in which case the program terminates
    - It reaches a stack frame where the calling code declares that it will *catch* an exception matching the type of that which was thrown.
  - Exceptions should NOT be used to implement typical, “happy path” logic, i.e., in place of returning a value.

Exceptions in Java

- All exceptions extend parent class `Throwable`
- `Throwable` is extended by three primary exception types/families:
  - `Exception` – Checked exceptions
  - `RuntimeException` – Unchecked exceptions
  - `Error` – Unchecked, generally fatal runtime errors
- Many subclasses within each family
- Inheritance hierarchy is often used to encapsulate the set of errors occurring within a given package/module

Checked and Unchecked Exceptions

- Checked exceptions:
  - A method that can generate a *checked exception* must declare that it throws an exception of the respective type.
  - A method that calls a method that declares that it throws a checked exception must either:
    - Explicitly catch that exception and/or
    - Declare that it throws an exception of the respective type
- Unchecked exceptions
  - May be thrown without a throws declaration
  - May also be thrown by the JVM (e.g., `NullPointerException`)
  - Callers choose which, if any, unchecked exceptions they wish to catch

Exception Generation

- Java keyword `throw` – think “throw up your hands”
- Syntax example:

```
void foo() throws TooBigException {
       if( size > MAX_SIZE ) {
              throw new TooBigException();
       }
       //Code…
}
```

- Example with message

```
       //somewhere else…
       if( !checkUsername()) {
              throw new LoginException(“Invalid Username”);
       } else if( !checkPassword()) {
              throw new LoginException(“Invalid password”);
       }
```

Declaring thrown checked exceptions

- Methods that throw checked exceptions must be declared with the “throws” modifier following the arguments list.
- The throws list may contain one or more identifiers, each of which must extend Throwable (and generally extend Exception)
- Example:

```
void foo() throws MyCheckedException { … }
void bar() throws IOException, AuthException { … }
```

- The throws list *is part of the method’s signature*

Exception Handling

- When calling a method that may throw one or more exceptions, we can choose to handle such exceptions.
- In Java, the try…catch (..finally) construct provides such functionality.
- Semantics:
  - Try to do something that may throw an exception
  - If an exception of type A is thrown, execute some code X
  - Else if an exception of type B is thrown, execute some code Y
  - (Optionally) Finally, execute some code Z whether or not an exception was thrown

Syntax

```
//Within some method body:

try {
        //Statements
} catch ( ExceptionType1 ex ) {
        //Code to deal with ExceptionType1
} [ catch ( ExceptionType2 ex) {
        //Code to deal with ExceptionType2
} ] [ finally {
        //Code to be executed whether or not an exception
        //was thrown in the try and guaranteed to be executed before
        //control leaves the try… construct.
}

//Example:

try {
        y = Integer.parseInt(s);
        this.setValue(y);
        return true;
} catch (NumberFormatException ex) {
        return false;
}
```

Chained Exceptions

- Idea: I catch some exception thrown by some method I called. Unfortunately, I can’t handle the error either. I’d like to throw an exception that tells my caller about both the error I’m generating as well as the *cause* of that error
- Java provides a lovely mechanism called exception chaining, wherein an exception can contain an exception of another type. E.g.:

```
… } catch ( ExType1 ex) {
      throw new MyExType(ex, “Damn.”);
}
```

Code Examples

- package recitations.recitation11.generics
  - Simple examples of the use of generics
- Package recitations.recitation11
  - [Generics and abstract classes] AbstractQueue, SimpleQueue
    - LinkedQueue, ListNode
  - [Idem, continued, and exceptions]
    - CheckedQueue, BoundedQueue
      - BoundedQueueException, QueueEmptyException, QueueFullException
  - [Putting things together] DropTailBoundedQueue