CS61B Lecture #12: Delegation, Exceptions, Assorted Features

- Delegation
- Exceptions
- Importing
- Nested classes.
- Type testing.

Trick: Delegation and Wrappers

- Not always appropriate to use inheritance to extend something.
- Homework gives example of a `TrReader`, which *contains* another `Reader`, to which it *delegates* the task of actually going out and reading characters.
- Another example: a class that instruments objects:

```
interface Storage {
  void put(Object x);
  Object get();
}
```

```
class Monitor implements Storage {
  int gets, puts;
  private Storage store;
  Monitor(Storage x) { store = x; gets = puts = 0; }
  public void put(Object x) { puts += 1; store.put(x); }
  public Object get() { gets += 1; return store.get(); }
}
```

```
// ORIGINAL
Storage S = something;
f(S);
```

```
// INSTRUMENTED
Monitor S = new Monitor(something);
f(S);
System.out.println(S.gets + " gets");
```

`Monitor` is called a *wrapper class*.

What to do About Errors?

- Large amount of any production program devoted to detecting and responding to errors.
- Some errors are external (bad input, network failures); others are internal errors in programs.
- When method has stated precondition, it's the client's job to comply.
- Still, it's nice to detect and report client's errors.
- In Java, we *throw exception objects,* typically:

```
throw new SomeException(optional description);
```

- Exceptions are objects. By convention, they are given two constructors: one with no arguments, and one with a descriptive string argument (which the exception stores).
- Java system throws some exceptions implicitly, as when you dereference a null pointer, or exceed an array bound.

Catching Exceptions

- A **throw** causes each active method call to *terminate abruptly,* until (and unless) we come to a **try** block.
- Catch exceptions and do something corrective with **try**:

```
try {
    Stuff that might throw exception;
} catch (SomeException e) {
    Do something reasonable;
} catch (SomeOtherException e) {
    Do something else reasonable;
}
Go on with life;
```

- When *SomeException* exception occurs during "Stuff..." and is not handled there, we immediately "do something reasonable" and then "go on with life."
- Descriptive string (if any) available as `e.getMessage()` for error messages and the like.

Catching Exceptions, II

- Using a supertype as the parameter type in a **catch** clause will catch any subtype of that exception as well:

```
try {
    Code that might throw a FileNotFoundException or a
        MalformedURLException ;
} catch (IOException ex) {
    Handle any kind of IOException;
}
```

- Since `FileNotFoundException` and `MalformedURLException` both inherit from *IOException*, the **catch** handles both cases.
- Subtyping means that multiple **catch** clauses can apply; Java takes the first.
- Stylistically, it's nice to be more specific (concrete) about exception types where possible.
- In particular, our style checker will therefore balk at the use of `Exception`, `RuntimeException`, `Error`, and `Throwable` as exception supertypes.

Catching Exceptions, III

- There's a relatively new shorthand for handling multiple exceptions the same way:

```
try {
    Code that might throw IllegalArgumentException
        or IllegalStateException;
} catch (IllegalArgumentException|IllegalStateException ex) {
    Handle exception;
}
```

Exceptions: Checked vs. Unchecked

- The object thrown by **throw** command must be a subtype of `Throwable` (in `java.lang`).
- Java pre-declares several such subtypes, among them
  - `Error`, used for serious, unrecoverable errors;
  - `Exception`, intended for all other exceptions;
  - `RuntimeException`, a subtype of `Exception` intended mostly for programming errors too common to be worth declaring.
- Pre-declared exceptions are all subtypes of one of these.
- Any subtype of `Error` or `RuntimeException` is said to be *unchecked.*
- All other exception types are *checked.*

Unchecked Exceptions

- Intended for
  - Programmer errors: many library functions throw `IllegalArgumentException` when one fails to meet a precondition.
  - Errors detected by the basic Java system: *e.g.*,
    - Executing `x.y` when `x` is null,
    - Executing `A[i]` when `i` is out of bounds,
    - Executing `(String) x` when `x` turns out not to point to a `String`.
  - Certain catastrophic failures, such as running out of memory.
- May be thrown anywhere at any time with no special preparation.

Checked Exceptions

- Intended to indicate exceptional circumstances that are *expected* to happen from time to time. Examples:
  - Attempting to open a file that does not exist.
  - Input or output errors on a file.
  - Receiving an interrupt.
- Every checked exception that can occur inside a method must either be handled by a `try` statement, or reported in the method's declaration.
- For example,

```
void myRead() throws IOException, InterruptedException { ... }
```

means that myRead (or something it calls) *might* throw `IOException` or `InterruptedException`.

A Language Design Issue

Java makes the following illegal for checked exceptions like `IOException`. Why?

```
class Parent {
   void f() { ... }
}
```

```
class Child extends Parent {
    void f() throws IOException { ... }
}
```

A Language Design Issue

Java makes the following illegal for checked exceptions like `IOException`. Why?

```
class Parent {
   void f() { ... }
}
```

```
class Child extends Parent {
    void f() throws IOException { ... }
}
```

Consider, for example,

```
static void process(Parent p) {
    p.f();
}
```

According to the specification for class `Parent`, this is supposed to be OK, but the call `p.f()` actually calls `Child.f`, which might throw `IOException`. So contrary to the intent of checked exceptions, the `process` method might throw a checked exception that it does not list.

Good Practice

- Throw exceptions rather than using print statements and System.exit everywhere,
- ...because response to a problem may depend on the *caller*, not just method where problem arises.
- Nice to throw an exception when programmer violates preconditions.
- Particularly good idea to throw an exception rather than let bad input corrupt a data structure.
- Good idea to document when methods throw exceptions.
- To convey information about the cause of exceptional condition, put it into the exception rather than into some global variable:

```
class MyBad extends Exception {
   public IntList errs;
   MyBad(IntList nums) { errs=nums; }
}
```

```
try {...
} catch (MyBad e) {
 ... e.errs ...
}
```

Terminology

- Many students speak of "throwing" an error when they mean "throwing an exception in order to *report* an error."
- This is a confusion of implementation (the code a program uses to *internally* signal an exceptional condition) with visible behavior (printing an error message).
- Users are not supposed to see them, but rather the messages that interpret these exceptions and report the exceptional conditions:

  Good: Program prints

  ```
  File myData.txt not found.
  ```

  Bad: Program prints

  ```
  Exception in thread "main" java.io.IOException: File not found
       at foo.main(foo.java:4)
  ```

Importing

- Writing `java.util.List` every time you mean `List` or `java.lang.regex.Pattern` every time you mean `Pattern` is annoying.
- The purpose of the **import** clause at the beginning of a source file is to define abbreviations:
  - `import java.util.List;` means "within this file, you can use `List` as an abbreviation for `java.util.List`.
  - `import java.util.*;` means "within this file, you can use *any* class name in the package `java.util` without mentioning the package."
- Importing does *not* grant any special access; it *only* allows abbreviation.
- In effect, your program always contains `import java.lang.*;`

Static importing

- One can easily get tired of writing `System.out` and `Math.sqrt`. Do you really need to be reminded with each use that `out` is in the `java.lang.System` package and that `sqrt` is in the Math package (duh)?
- Both examples are of *static* members. A feature of Java allows you to abbreviate such references:
  - `import static java.lang.System.out;` means "within this file, you can use `out` as an abbreviation for `System.out`.
  - `import static java.lang.System.*;` means "within this file, you can use *any* static member name in `System` without mentioning the package.
- Again, this is *only* an abbreviation. No special access.
- Alas, you can't do this for classes in the anonymous package.

Nesting Classes

- Sometimes, it makes sense to *nest* one class in another. The nested class might
  - be used only in the implementation of the other, or
  - be conceptually "subservient" to the other
- Nesting such classes can help avoid name clashes or "pollution of the name space" with names that will never be used anywhere else.
- Example: Polynomials can be thought of as sequences of terms. Assuming that terms aren't used outside of Polynomials, you might define a class to represent a term *inside* the Polynomial class:

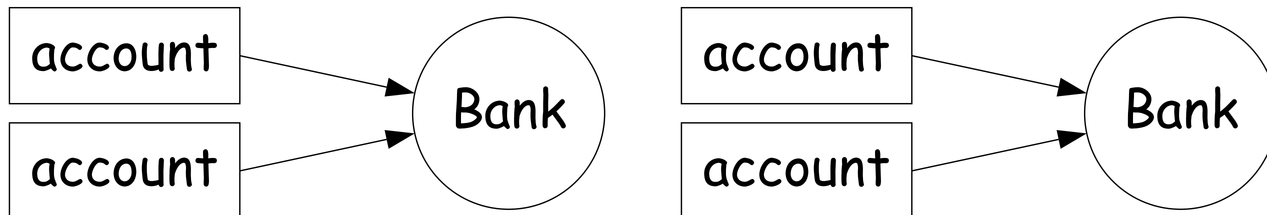
```
class Polynomial {

  methods on polynomials

  private Term[] terms;
  private static class Term {
    ...
  }
}
```

Inner Classes

- Last slide showed a *static* nested class. Static nested classes are just like any other, except that they can be private or protected, and they can see private variables of the enclosing class.
- Non-static nested classes are called *inner classes.*
- Used when each instance of the nested class is created by and naturally associated with an instance of the containing class, like Banks and Accounts:



Example: Banks and Accounts

```
class Bank {
    private void addFunds(int amount) { totalFunds += amount; }

    public class Account {
        int _balance;
        public void deposit(int amount) {
            _balance += amount;
            Bank.this.addFunds(_balance);
        } // Bank.this means "the bank that created me"
    }
}

---------------------------------------------------

Bank bank = new Bank(...);
Bank.Account a0 = bank.new Account(...);
Bank.Account a1 = bank.new Account(...);
```

Example: Iterators

```
public class ArrayList<T> {
    ...
    public T get(int k) { ... }
    public int size() { ... }

    public Iterator<T> iterator() {
        return new ArrayIterator<T>();
        // or return this.new ArrayIterator<T>();
    }

    private class ArrayIterator implements Iterator<T> {
        int _k;
        ArrayIterator() { _k = 0; }
        public boolean hasNext() { return _k < size(); }
        public T next() { _k += 1; return get(_k - 1); }
    }
}
```

Type testing: instanceof

- It is possible to ask about the dynamic type of something:

```
void typeChecker(Reader r) {
  if (r instanceof TrReader)
    System.out.print("Translated characters: ");
  else
    System.out.print("Characters: ");
  ...
}
```

- However, this is seldom what you want to do. Why do this:

```
if (x instanceof StringReader)
  read from (StringReader) x;
else if (x instanceof FileReader)
  read from (FileReader) x;
...
```

when you can just call `x.read()`?!

- In general, use instance methods rather than **instanceof**.