CMSC 330: Organization of Programming Languages

Polymorphism
class Stack {
    class Entry {
        Integer elt; Entry next;
        Entry(Integer i, Entry n) { elt = i; next = n; }
    }
    Entry theStack;
    void push(Integer i) {
        theStack = new Entry(i, theStack);
    }
    Integer pop() throws EmptyStackException {
        if (theStack == null) {
            throw new EmptyStackException();
        } else {
            Integer i = theStack.elt;
            theStack = theStack.next;
            return i;
        }
    }
}
Stack is = new Stack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();

Now we want a stack of Floats
... or some objects ...
Polymorphism Using Object

class Stack {
    class Entry {
        Object elt; Entry next;
        Entry(Object i, Entry n) { elt = i; next = n; }
    }
    Entry theStack;
    void push(Object i) {
        theStack = new Entry(i, theStack);
    }
    Object pop() throws EmptyStackException {
        if (theStack == null)
            throw new EmptyStackException();
        else {
            Object i = theStack.elt;
            theStack = theStack.next;
            return i;
        }
    }
}
New Stack Client

Stack is = new Stack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = (Integer) is.pop();

• Now Stacks are reusable
  – push() works the same
  – But now pop() returns an Object
    • Have to downcast back to Integer
    • Not checked until run-time
Polymorphism

• From the Greek...
  – πολοί: “poloi” = “many”
  – μορφή: “morphi” = “form”

• Polymorphism refers to the concept of taking on different forms
  – Data representing multiple concepts
  – Code operating on multiple input types
Polymorphism

• Subtyping is a kind of polymorphism
  – Sometimes called *subtype polymorphism*
  – Allows method to accept objects of *many* types

• We saw *parametric polymorphism* in OCaml
  – It’s polymorphism because polymorphic functions can be applied to many different types

• *Ad-hoc polymorphism* is overloading
  – Operator overloading in C++
  – Method overloading in Java
Parametric Polymorphism (for Classes)

• After Java 1.5, we can parameterize the Stack class by its element type

• Syntax:
  – Class declaration: `class A<T> { ... }`
    • `A` is the class name, as before
    • `T` is a type variable, can be used in body of class (...)
  – Client usage declaration: `A<Integer> x;`
    • We instantiate `A` with the `Integer` type
Parametric Polymorphism for Stack

class Stack<ElementType> {
    class Entry {
        ElementType elt; Entry next;
        Entry(ElementType i, Entry n) { elt = i; next = n; }
    }
    Entry theStack;
    void push(ElementType i) {
        theStack = new Entry(i, theStack);
    }
    ElementType pop() throws EmptyStackException {
        if (theStack == null)
            throw new EmptyStackException();
        else {
            ElementType i = theStack.elt;
            theStack = theStack.next;
            return i;
        }
    }
}
Stack<Element> Client

Stack<Integer> is = new Stack<Integer>();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();

• No downcasts
• Type-checked at compile time
• No need to duplicate Stack code for every usage
  – line i = is.pop(); can stay the same even if the type of is isn’t an integer in every path through the program
Parametric Polymorphism for Methods

• **String** is a subtype of **Object**
  1. static **Object** id(**Object** x) { return x; }
  2. static **Object** id(**String** x) { return x; }
  3. static **String** id(**Object** x) { return x; }
  4. static **String** id(**String** x) { return x; }

• Can’t pass an **Object** to 2 or 4
• 3 isn’t type-safe (doesn’t pass type checking)
• Can pass a **String** to 1 but you get an **Object** back
Parametric Polymorphism, Again

• But id() doesn’t care about the type of x
  – It works for any type

• So parameterize the method:
  
  ```java
  static <T> T id(T x) { return x; }
  Integer i = id(new Integer(3));
  ```
  
  – Notice no need to instantiate id; compiler figures out the correct type at usage
  – The formal parameter has type T, the actual parameter has type Integer
Standard Library, and Java 1.5 (and later)

• Part of Java 1.5 (called “generics”)
  – Comes with replacements for java.util.*
    • class LinkedList<A> { ... }
    • class HashMap<A, B> { ... }
    • interface Collection<A> { ... }
  – Excellent tutorial listed on references page

• But they didn’t change the JVM to add generics
  – How did they do that?
Translation via Erasure

• Replace uses of type variables with Object
  – class A<T> { ...T x;... } becomes
  – class A { ...Object x;... }

• Add downcasts wherever necessary
  – Integer x = A<Integer>.get();  becomes
  – Integer x = (Integer) (A.get());

• So why did we bother with generics if they’re just going to be removed?
  – Because the compiler still did type checking for us
  – We know that those casts will not fail at run time
Limitations of Translation

• Some type information not available at compile-time
  – Recall type variables $T$ are rewritten to Object

• Disallowed, assuming $T$ is type variable:
  – new $T()$ would translate to new Object() (error)
  – new $T[n]$ would translate to new Object[n] (warning)
  – Some casts/instanceofs that use $T$
    • (Only ones the compiler can figure out are allowed)
Using with Legacy Code

- Translation via type erasure
  - class A <T> becomes class A

- Thus class A is available as a “raw type”
  - class A<T> { ... }
  - class B { A x; } // use A as raw type

- Sometimes useful with legacy code, but...
  - Dangerous feature to use
  - Relies on implementation of generics, not semantics
Polymorphism Quiz #1

• What kind of polymorphism?
  – Possible answers: subtype, parametric (classes or methods), or ad-hoc

• foo(x) takes floats and ints because the programmer provided two different versions of the function, one for each type

```c
int foo(int x) { return x+1; }
float foo(float x) { return x+1.0; }
```
Polymorphism Quiz #2

• What kind of polymorphism?
  – Possible answers: subtype, parametric (classes or methods), or ad-hoc

• foo(x) takes floats and ints because the function body does not perform any operation that constrains the type of x

  let foo x = x
Polymorphism Quiz #3

• What kind of polymorphism?
  – Possible answers: subtype, parametric (classes or methods), or ad-hoc

• foo(x) takes Floats and Integers because both are subclasses of Numbers

```java
class Number { }
class Float extends Number { }
class Integer extends Number { }
String foo(Number x)
    { return x.toString(); }
```
Polymorphism Quiz #4

• What kind of polymorphism?
  – Possible answers: subtype, parametric (classes or methods), or ad-hoc

• foo(x) takes Floats and Integers because the programmer used generics

  \[ <T> \ T \ foo(T \ x) \ \{ \ \text{return} \ \ x; \ \} \]
Polymorphism Quiz #5

• What kind of polymorphism?
  – Possible answers: subtype, parametric (classes or methods), or ad-hoc

• the constructor for class Foo takes Floats and Integers because the programmer used generics

```java
class Foo<T> {
    Foo(T x) {
    }
}
```
Subtyping and Arrays

• Java has one interesting subtyping feature:
  – If $S$ is a subtype of $T$, then
  – $S[]$ is a subtype of $T[]$
Problem with Subtyping Arrays

public class A { ... }
public class B extends A { void newMethod(); }
...

void foo(void) {
    B[] bs = new B[3];
    A[] as;

    as = bs;    // Since B[] subtype of A[]
    as[0] = new A();    // (1)
    bs[0].newMethod();    // (2) Fails since not type B
}

• Program compiles without warning
• Java must generate run-time check at (1) to prevent (2)
  – Type written to array must be subtype of array contents
Subtyping for Generics

• Is `Stack<Integer>` a subtype of `Stack<Object>`?
  – We could have the same problem as with arrays
  – Thus Java forbids this subtyping

• Now consider the following method:

```java
int count(Collection<Object> c) {  
    int j = 0;
    for (Iterator<Object> i = c.iterator(); i.hasNext(); ) {  
        Object e = i.next(); j++;
    }
    return j;
}
```

  – Not allowed to call `count(x)` where `x` has type `Stack<Integer>`
Solution I: Use Polymorphic Methods

```java
<T> int count(Collection<T> c) {
    int j = 0;
    for (Iterator<T> i = c.iterator(); i.hasNext(); ) {
        T e = i.next(); j++;
    }
    return j;
}
```

- But requires a “dummy” type variable that isn’t really used for anything
Solution II: Wildcards

• Use ? as the type variable
  – Collection<?> is “Collection of unknown”

```java
int count(Collection<?> c) {
    int j = 0;
    for (Iterator<?> i = c.iterator(); i.hasNext(); ) {
        Object e = i.next(); j++;
    }
    return j; }
```

• Why is this safe?
  – Using ? is a contract that you’ll never rely on having a particular parameter type
  – All objects subtype of Object, so assignment to e ok
Legal Wildcard Usage

• Reasonable question:
  – Stack<Integer> is not a subtype of Stack<Object>
  – Why is Stack<Integer> a subtype of Collection<?>?

• Answer:
  – Wildcards permit “reading” but not “writing”
Example: Can read but cannot write

```java
int count(Collection<?> c) {
    int j = 0;
    for (Iterator<?> i = c.iterator(); i.hasNext(); ) {
        Object e = i.next();
        c.add(e); // fails: Object is not ?
        j++;
    }
    return j;
}
```
Shapes Example

• Suppose we have classes Circle, Square, and Rectangle, all subtypes of Shape
More on Generic Classes

• Suppose we have the following method:

```java
void drawAll(Collection<Shape> c) {
    for (Shape s : c)
        s.draw();
}
```

  - Can we pass this method a `Collection<Square>`?
    • No, not a subtype of `Collection<Shape>`
  - How about the following?

```java
void drawAll(Collection<?> c) {
    for (Shape s : c) // not allowed, assumes ? is Shape
        s.draw();
}
```
Bounded Wildcards

• We want drawAll to take a Collection of anything that is a subtype of shape
  – this includes Shape itself

```java
void drawAll(Collection<? extends Shape> c) {
    for (Shape s : c)
        s.draw();
}
```

– This is a bounded wildcard
– We can pass Collection<Circle>
– We can safely treat s as a Shape
Upper Bounded Wild Cards

- ? extends Shape actually gives an *upper bound* on the type accepted
- Shape is the upper bound of the wildcard
Lower Bounded Wildcards

- Dual of the upper bounded wildcards
- \(? \text{ super} \) Rectangle denotes a type that is a supertype of Rectangle
  - Rectangle is included, and is a lower bound on the type accepted

```
Shape
  \-- Circle
  \-- Rectangle
    \-- Square
```
Lower Bounded Wildcards (cont’d)

• Example:

```java
void foo(Collection<? super Circle> c) {
    c.add(new Circle());
    c.add(new Rectangle()); // fails
}
```

- Because `c` is a `Collection` of something that is always compatible with `Circle`
Bounded Type Variables

• You can also add bounds to regular type vars

```java
<T extends Shape> T getAndDrawShape(List<T> c) {
    c.get(1).draw();
    return c.get(2);
}
```

– This method can take a List of any subclass of Shape
  • This addresses some of the reason that we decided to introduce wild cards
Ruby “Generics”

• Does Ruby support parametric polymorphism?

• If so, how?
CMSC 330: Organization of Programming Languages

Exceptions
Preconditions

• Functions often have requirements on their inputs

// Return maximum element in A[i..j]
int findMax(int[] A, int i, int j) { ... }

– A is nonempty
– A isn't null
– i and j must be nonnegative
– i and j must be less than A.length
– i < j (maybe)

• These are called *preconditions*
Dealing with Errors

• What do you do if a precondition isn’t met?

• What do you do if something unexpected happens?
  – Try to open a file that doesn’t exist
  – Try to write to a full disk
Signaling Errors

• Style 1: Return invalid value

    // Returns value key maps to, or null if no
    // such key in map
    Object get(Object key);

– Disadvantages?
Signaling Errors (cont’d)

• Style 2: Return an invalid value and status

```c
static int lock_rdev(mdk_rdev_t *rdev) {
    ...
    if (bdev == NULL)
        return -ENOMEM;
    ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```
Problems with These Approaches

- What if all possible return values are valid?
  - E.g., `findMax` from earlier slide
- What about errors in a constructor?
- What if client forgets to check for error?
- What if client can’t handle error?
  - Needs to be dealt with at a higher level
- Poor modularity- exception handling code becomes scattered throughout program
- 1996 Ariane 5 failure classic example of this …
Ariane 5 failure

Design issues: In order to save funds and ensure reliability, and since the French Ariane 4 was a successful rocket, the Inertial Reference System (SRI) from Ariane 4 was reused for the Ariane 5.

What happened?: On June 4, 1996 the Ariane 5 launch vehicle failed 39 seconds after liftoff causing the destruction of over $100 million in satellites.

Cause of failure: The SRI, which controls the attitude (direction) of the vehicle by sending aiming commands to the rocket nozzle, sent a bad command to the rocket causing the nozzle to move the rocket toward the horizontal. The backup SRI had failed for the same reason 72 millisec earlier.

The vehicle had to then be destroyed.
Why Ariane 5 failed

• SRI tried to convert a floating point number out of range to integer. Therefore it issued an error message (as a 16 bit number). This 16 bit number was interpreted as an integer by the guidance system and caused the nozzle to move accordingly.

• Ada range checking was disabled since the SRI was supposedly processing at 80% load and the extra time needed for range checking was deemed unnecessary since the Ariane 4 software worked well.

• The ultimate cause of the problem was that the Ariane 5 has a more pronounced angle of attack and can move horizontally sooner after launch. The “bad value” was actually the appropriate horizontal speed of the vehicle.
Better approaches: Exceptions in Java

• On an error condition, we *throw* an exception

• At some point up the call chain, the exception is *caught* and the error is handled

• Separates normal from error-handling code

• A form of non-local control-flow
  – Like `goto`, but structured
Throwing an Exception

• Create a new object of the class `Exception`, and throw it

```java
if (i >= 0 && i < a.length )
    return a[i];
throw new ArrayIndexOutOfBoundsException();
```

• Exceptions thrown are part of the return type in Java
  – When overriding method in superclass, cannot throw any more exceptions than parent’s version
Method throws declarations

• A method declares the exceptions it might throw

```java
public void openNext() throws UnknownHostException, EmptyStackException {
  ...
}
```

• Must declare any exception the method might throw
  – Unless it is caught in (masked by) the method
  – Includes exceptions thrown by called methods
  – Certain kinds of exceptions excluded
Java Exception Hierarchy

Thrownable

Error

Exception

RuntimeException

checked

unchecked
Checked Exceptions

• Represent invalid conditions in areas outside the immediate control of the program
  – Invalid user input
  – Database problems
  – Network outages
  – Absent files
  – etc.

• A method is obliged to establish a policy for all checked exceptions thrown by its implementation
  – Either pass the checked exception further up the stack, or handle it somehow
Unchecked Exceptions

- Represent defects in the program (bugs)
- Are subclasses of RuntimeException
- Common types:
  - IllegalArgumentException
  - NullPointerException
  - IllegalStateException
- A method is not obliged to establish a policy for the unchecked exceptions thrown by its implementation (and they almost always do not do so)
- These should not be encountered in production code if proper debugging was done
Errors

- Represent serious problems that a “reasonable” program shouldn’t attempt to handle
  - Assertions
  - Character set decoding issue
  - Class file linkage errors
  - JVM configuration errors
  - Out of virtual memory
- Are subclasses of Error
- A method is not obliged to establish a policy for errors
Java Exception Hierarchy

```
 Throwable
  ↓               ↓
 Error           Exception
                ↓
 RuntimeException
```

- **Throwable**
- **Error**
- **Exception**
- **RuntimeException**

- Checked
- Unchecked
Implementing Exceptions in Java

- JVM knows about exceptions, and has built-in mechanism to handle them

```java
class A {
    void foo() {
        try {
            Object f = null;
            f.hashCode();
        }
        catch (NullPointerException e) {
            System.out.println("caught");
        }
    }
}
```
Exception Handling

• First **catch** with supertype of the exception catches it

• **finally** is always executed

```java
try { if (i == 0) return; myMethod(a[i]); } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("a[] out of bounds"); } catch (MyOwnException e) {
    System.out.println("Caught my error"); } catch (Exception e) {
    System.out.println("Caught" + e.toString()); throw e; } finally { /* stuff to do regardless of whether an exception */ /* was thrown or a return taken */ }
```
Implementing Exceptions in Java

void foo();

Code:
0:    aconst_null
1:    astore_1
2:    aload_1
3:    invokevirtual #2;  //hashCode
6:    pop
7:    goto    19

10:   astore_1
11:   getstatic #4;  //System.out
14:   ldc #5;   //String caught
16:   invokevirtual #6;  //println
19:   return

Exception table:
from   to  target type
0     7    10   NullPointerException

• Exception table tells JVM what handlers there are for which region of code
Unhandled Exceptions

• Design battle: resumption vs. termination
  – Resumption: an exception handler can resume computation at the place where the exception was thrown
  – Termination: throwing an exception terminates execution at the point of the exception

• Resumption is risky, but so is termination
  – What do you think?
Issues with Exceptions

• Exceptions are essentially a return to GOTO statements
  – Very hard to analyze statically
  – See Djikstra’s famous 1968 letter “Go To Statement Considered Harmful” for other reasons why GOTO statements are a bad idea

• We can try to more formally and completely describe our preconditions, but sometimes this is too difficult

• In practice, exceptions are usually the best solution