Polymorphism

Definition
- Feature that allows values of different data types to be handled using a uniform interface

Applicable to
- Functions
  - Same function applied to different data types
  - Example:
    ```ml
    let hd = function (h::_) -> h
    ```
- Data types
  - Same data type can contain different data types
  - Example:
    ```ml
type 'a option =
| None |
| Some of 'a |
```
Subtype Polymorphism

- Found in object-oriented programming languages
  - Supported through inheritance
- Any function w/ object as parameter is polymorphic
  - If formal parameter is of class A
  - Argument may be any object from subclass of A

```
class A { ...
    class B extends A { ...

    static void f(A arg) { ...
        A a = new A();
        B b = new B();
        f(a); f(b); // f accepts arg of type A or B
    }
```
Operator Overloading (cont.)

- User-specified operator overloading
  - Supported in languages such as Ruby, C++
  - Makes user data types appear more like native types
- Examples
  - Defining function for `^` operator

```ruby
class MyS
def ^ (arg)
  ...
end
end
```

```c++
class MyS {
  MyS operator ^(MyS arg){
    ...
  }
};
```

Parametric Polymorphism

- Found in statically typed functional languages
  - OCaml, ML, Haskell
  - Example
    ```
    let hd = function (h:::) -> h 'a list -> 'a
    ```
- Also used in object oriented programming
  - Known as generic programming
  - Example: Java, C++

An Integer Stack Implementation

```java
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;
    }
  }
}
```

Integer Stack Client

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

If we also want a stack of Floats, do we need to write a Float Stack class?
An Object Stack Implementation

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();

Object stacks are polymorphic & reusable
• push() works the same
• But now pop() returns an Object
   ➔ Have to downcast back to Integer
   • Not checked until run-time

General Problem

• When we move from an X container to an Object container
  • Methods that take X’s as input parameters are OK
    ➔ If you’re allowed to pass Object in, you can pass any X in
  • Methods that return X’s as results require downcasts
    ➔ You only get Objects out, which you need to cast down to X

General characteristic of subtype polymorphism

Parametric Polymorphism (for Classes)

• Java 1.5 introduced generics
• 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

```java
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

```java
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 doesn’t type check
- 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 static 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 replacement 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 was that done?

### 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 runtime

### 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/instanceof 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, plus unsafe
  - Relies on implementation of generics, not semantics
Subtyping and Arrays

- Java has one funny subtyping feature
  - If $S$ is a subtype of $T$, then
  - $S[\cdot]$ is a subtype of $T[\cdot]$

- Let's write methods that take arbitrary arrays

  ```java
  public static void reverseArray(Object [] A) {
      for(int i=0, j=A.length-1; i<j; i++,j--) {
          Object tmp = A[i];
          A[i] = A[j];
          A[j] = tmp;
      }
  }
  ```

Problem with Subtyping Arrays

```java
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;
  }
  ```

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 c

```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;
}
```

For Loops

Java 1.5 has a more convenient syntax for this standard for loop

```java
int count(Collection<??> c) {
    int j = 0;
    for (Object e : c)
        j++;
    return j;
}
```

- This loop will get the standard iterate and set e to each element of the list, in order
More on Generic Classes

Suppose we have classes Circle, Square, and Rectangle, all subtypes of Shape

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

Bounded Wildcards

We want drawAll to take a Collection of anything that is a subtype of shape

• this includes Shape itself

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

Bounded Wildcards (cont.)

Should the following be allowed?

void foo(Collection<? extends Shape> c) {
    c.add(new Circle());
}

• No, because c might be a Collection of something that is not compatible with Circle
• This code is forbidden at compile time
Lower Bounded Wildcards

- Dual of the upper bounded wildcards
- ? super Rectangle denotes a type that is a supertype of Rectangle
  - Type Rectangle is included
- ? super Rectangle gives a lower bound on the type accepted

Lower Bounded Wildcards (cont.)

- Now the following is allowed

```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
  - Once again, this only works for methods