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
    ```
    let hd = function (h::_) -> h
    ```

- Data types
  - Same data type can contain different data types
  - Example
    ```
    type optional_int = None | Some of int
    ```

Two Kinds of Polymorphism

- Described by Strachey in 1967

  Ad hoc parallelism
  - Range of types is finite
  - Combinations must be specified in advance
  - Behavior may differ based on type of arguments

  Parametric parallelism
  - Code written without mention of specific type
  - May be transparently used with arbitrary # of types
  - Behavior is same for different types of arguments

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 { _ } // subclass
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
```

Overloading

- Multiple copies of function
  - Same function name
  - But different number / type of parameters

  Arguments determines function actually invoked

  - Function is uniquely identified not by function name, but by name + order & number of argument type(s)
  - Examples
    ```
    static void print(Integer arg) {...}
    static void print(Float arg) {...}
    print(1); // invokes 1st print
    print(3.14); // invokes 2nd print
    ```

Polymorphism Overview

- Ad-hoc
  - Subtype (for OO languages)
  - Overloading
    - Operator overloading

- Parametric
  - Generic programming (for OO languages)
    - Bounded parametric parallelism
Operator Overloading

- Treat operators as functions
  - With special syntax for invocations
  - Behavior different depending on operand type

- Example
  - `+` in Java

```
1 + 2    // integer addition
1.0 + 3.14  // float addition
"Hello" + "world"  // string concatenation
```

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

C++
```cpp
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 class definition

  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;
    }
  }
}
```

An Object Stack Implementation

```java
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;
    }
  }
}
```

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

```java
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

```java
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

Liskov Substitution Principle

If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \) then \( S \) is a subtype of \( T \).

- I.e, if anyone expecting a \( T \) can be given an \( S \), then \( S \) is a subtype of \( T \).

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 feature of subtype polymorphism

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<ElementType> Client

```java
Stack<Integer> is = new Stack<Integer>();
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

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?

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)

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

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[] is a subtype of T[]

- 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

- 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

Solution I: Use Polymorphic Methods

- Not allowed to call count(x) where x has type Stack<Integer>

Solution II: Wildcards

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

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(); // fails: Object is not ?
        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

```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

```
Shape

Circle

Rectangle

Square
```

Bounded Wildcards (cont.)

- Should the following be allowed?

```java
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`
    - `T` is included
  - `? super Rectangle` gives a lower bound on the type accepted

![Diagram showing relationships between Shape, Circle, Rectangle, and Square]

Lower Bounded Wildcards (cont.)

- But 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