Polymorphism

Recall that \( B \) is a subtype of \( A \) if, everywhere you expect an \( A \), you can accept a \( B \)
- Subtypes come from subclassing with \texttt{extends}
- Subtypes come from interfaces with \texttt{implements}

This is a kind of \textit{type polymorphism}
- Methods can accept objects of \textit{many} types, not just one
- This is usually called \textit{subtype polymorphism}

### IntegerStack Example

```java
class IntegerStack {
    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;
        }
    }
}
```

### IntegerStack Client

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

- This is OK, but what if we want other kinds of stacks?
  - Need to make one XStack for each kind of X
  - Problems: Code bloat, maintainability nightmare
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;
    }
  }
}

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

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

• This is a general feature of subtype polymorphism

Parametric Polymorphism (for Classes)

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

Stack<Client>

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

Parametric Polymorphism for Procedures

- Suppose B is a subtype of A
  1. static A id(A x) { return x; }
  2. static A id(B x) { return x; }
  3. static B id(A x) { return x; }
  4. static B id(B x) { return x; }
- Can’t pass an A to 2 or 4
- 3 doesn’t type check
- Can pass a B to 1 but you get an A out

Parametric Polymorphism, Again

- Observation: 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 it out
  ```
Summary: Kinds of Polymorphism

- Subtype polymorphism
  - Use subtype wherever supertype allowed

- Parametric polymorphism
  - When classes/methods work for any type; uses type variables

- Ad-hoc polymorphism
  - Overloading in Java

Parametric Polymorphism in Java

- Part of > Java 1.5 (called “generics”)
  - Comes with replacement for java.util.*
    - class LinkedList<A> { ... }
    - class HashMap<A, B> { ... }
    - interface Collection<A> { ... }

Implementation

- Generics translated into standard Java byte codes
  - Java VM hasn’t changed
  - Compiled programs can be run on any correct implementation of the JVM
  - Intuitively, generics “compiled out” of programs

Translation via Erasure

- (According to OOPSLA98 paper on gi)
- Replaces uses of type variables with Object
  - class A<T> { ...T x;... } ==> class A { ...Object x;... }
- Adds downcasts wherever necessary
  - Integer x = A<Integer>.get(); ==> Integer x = (Integer) (A.get());
Limitations of Translation

- Some type information not available at run-time
  - Recall type variables T are rewritten to Object

- Disallowed, assuming T is type variable
  - new T() would translate to new Object() (gjc error)
  - new T[n] would translate to new Object[n] (gjc warn)
    - Use public static <A> A[] newInstance(A[] a, int n) in java.lang.reflect.Array
  - 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> ==> class A

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

- Sometimes useful with legacy code, but...
- Dangerous feature to use, plus unsafe
  - Relies on implementation of generics, not semantics