

CMSC 330: Organization of Programming Languages

Polymorphism

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 'a option =  
  None  
  | Some of 'a
```

Two Kinds of Polymorphism

- ▶ Described by Strachey in 1967
- ▶ Ad hoc polymorphism
 - Range of types is finite
 - Combinations must be specified in advance
 - Behavior may **differ** based on type of arguments
- ▶ Parametric polymorphism
 - Code written without mention of specific type
 - May be transparently used with arbitrary # of types
 - Behavior is **same** for different types of arguments

CMSC 330

3

Polymorphism Overview

- ▶ Ad-hoc
 - Subtype (for OO languages)
 - Sometimes not considered ad-hoc, but referred to as **subtype polymorphism**
 - Overloading
 - Operator overloading
- ▶ Parametric
 - ML types
 - A.k.a. generic programming (for OO languages)
 - **Bounded parametric polymorphism** combines subtype and parametric polymorphism

CMSC 330

4

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
```

CMSC 330

5

Overloading

- ▶ Multiple copies of function
 - Same function name
 - But different number / type of parameters
- ▶ Arguments determine function actually invoked
 - Function is uniquely identified not by function name, but by name + order & number of argument type(s)
 - print(Integer i) → print_Integer(...)
 - print(Float f) → print_Float(...)

```
static void print(Integer arg) { ... }  
static void print(Float arg) { ... }  
print(1); // invokes 1st print  
print(3.14); // invokes 2nd print
```

CMSC 330

6

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

CMSC 330

7

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

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

Ruby

```
class MyS {
  MyS operator^(MyS arg) {
    ...
  }
}
```

C++

CMSC 330

8

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++

CMSC 330

9

An Integer Stack Implementation

```
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;
        }
    }
}
```

CMSC 330

10

Integer Stack Client

```
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?

CMS 330

11

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;
        }
    }
}
```

CMS 330

12

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

CMSC 330

13

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

CMSC 330

14

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

CMS 330

15

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;
        }
    }
}
```

CMS 330

16

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

CMSC 330

17

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

CMSC 330

18

Parametric Polymorphism, Again

- ▶ But `id()` doesn't care about the type of `x`
 - It works for any type
- ▶ So **parameterize** the static method

```
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`

CMS 330

19

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?

CMS 330

20

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

CMS 330

21

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

CMS 330

22

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

CMS 330

23

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

```
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;  
    }  
}
```

CMS 330

24

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

CMSC 330

25

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:

```
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>`

CMSC 330

26

Solution I: Use Polymorphic Methods

```
<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

CMSC 330

27

Solution II: Wildcards

- ▶ Use ? as the type variable
 - Collection<?> is “Collection of unknown”

```
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

CMSC 330

28

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”

CMSC 330

29

Example: Can Read But Cannot Write c

```
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; }
```

CMSC 330

30

For Loops

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

```
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

CMS 330

31

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) // not allowed,  
        s.draw();    assumes ? is  
}                    Shape
```

CMS 330

32

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();  
}
```

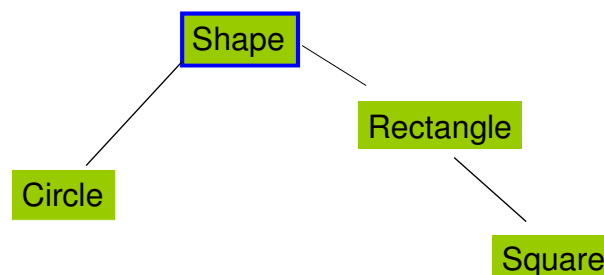
- This is a **bounded wildcard**
- We can pass Collection<Circle>
- We can safely treat s as a Shape

CMSC 330

33

Upper Bounded Wild Cards

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



CMSC 330

34

Bounded Wildcards (cont.)

- ▶ Should the following be allowed?

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

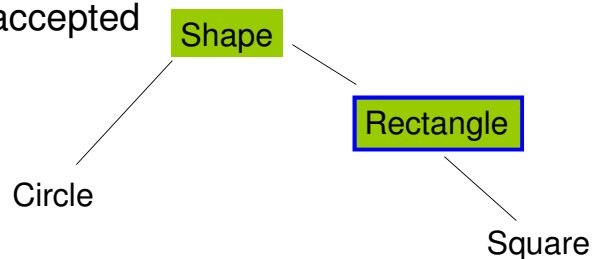
- No, because `c` might be a `Collection` of something that is not compatible with `Circle`
- This code is forbidden at compile time

CMSC 330

35

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



CMSC 330

36

Lower Bounded Wildcards (cont.)

- Now the following is allowed

```
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`

CMSC 330

37

Bounded Type Variables

- You can also add bounds to regular type vars

```
<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

CMSC 330

38