Java

- Developed in 1995 by Sun Microsystems
  - Started off as Oak, a language aimed at software for consumer electronics
  - Then the web came along...
- Java incorporated into web browsers
  - Java source code compiled into Java byte code
  - Executed (interpreted) on Java Virtual Machine
    - Portability to different platforms
    - Safety and security much easier, because code is not directly executing on hardware
- These days, Java used for a lot of purposes
  - Server side programming, general platform, etc.
Java Versions

- Java has evolved over the years
  - Virtual machine quite stable, but source language has been getting new features

- Will use the latest version of Java for this class
  - If you've got an older version, you might want to upgrade

Subtyping

- Both inheritance and interfaces allow one class to be used where another is specified
  - This is really the same idea: subtyping

- We say that A is a subtype of B if
  - A extends B or a subtype of B, or
  - A implements B or a subtype of B
Liskov Substitution Principle

If for each object \( o1 \) of type \( S \) there is an object \( o2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o1 \) is substituted for \( o2 \) 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 \( T \).
– Does our definition of subtyping in terms of extends and implements obey this principle?

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
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)

• In 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 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

- 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 run time
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() (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)

- Also produces some oddities
  - LinkedList<Integer>.class == LinkedList<String>.class
    - (These are uses of reflection to get the class object)

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[]

- Lets us 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;
}
```

  - 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 Stack<?>?

- 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;
}
```

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)
        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

```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 e 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

![Diagram of Shape, Circle, Rectangle, and Square]

Bounded Wildcards (cont’d)

- 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

```
Circle  \rightarrow  \text{Shape}  \rightarrow  \text{Rectangle}
```

- `Square`

Lower Bounded Wildcards (cont’d)

- But the following is allowed:

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

A more realistic example

```java
public interface Comparable<T> {
    int compareTo(T o);
}
// e.g., Boolean implements Comparable<Boolean>
public static <T extends Comparable<? super T>>
    void sort(List<T> list) {
        Object a[] = list.toArray();
        Arrays.sort(a);
        ListIterator<T> i = list.listIterator();
        for(int j=0; j<a.length; j++) {
            i.nextIndex();
            i.set((T)a[j]);
        }
    }
```

• I'm modifying the list via the Iterator. Why is this OK?