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 Java 1.5 (a.k.a Java 5.0) for this class
  - We will be using 1.5-specific features, so if you’ve got a different version, you will want to upgrade
  - Some of the new features in Java 1.5 came as a response to pressure from Microsoft’s C#

Object-Orientation

- Java is a class-based, object-oriented language
- Classes extend other classes to inherit
  - The root of the inheritance hierarchy is Object
  - Why have a root of the hierarchy?
- Classes also implement interfaces
  - Interface is like a class with declarations but no code
- Classes may extend one other class, but can implement many interfaces
  - Multiple inheritance is tricky to understand/implement
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 $o_1$ 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 $T$.
– Does our definition of subtyping in terms of extends and implements obey this principle?
Polymorphism in Java

• Subtyping is a kind of polymorphism
  – Sometimes called *subtype polymorphism*
  – Allows method to accept objects of *many* types

• Another kind: *parametric polymorphism*
  – Implemented as generic methods in Java

• *Ad-hoc polymorphism* is overloading
  – Method overloading

A Stack of Integers

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;
    }
  }
}
Inner Classes

- Classes can be nested inside other classes
  - These are called inner classes

- Within a class that contains an inner class, you can use the inner class just like any other class

```java
class Stack {
    ...
    private int numEntries;
    class Entry {
        Integer elt; Entry next;
        Entry(Integer i) { elt = i; next = null; numEntries++; }
    }
}
```

- Each inner “object” has an implicit reference to the outer “object” whose method created it
  - Can refer to fields directly, or use outer class name
Other Features of Inner Classes

• Outside of the outer class, use outer.inner notation to refer to type of inner class
  – E.g., Stack.Entry

• An inner class marked static does not have a reference to outer class
  – Can’t refer to instance variables of outer class
  – Must also use outer.inner notation to refer to inner class

• Question: Can Stack.Entry be made static?

Compiling Inner Classes

• The JVM doesn’t know about inner classes
  – Compiled away, similar to generics
  – Inner class Foo of outer class A produces A$Foo.class
  – Anonymous inner class of outer class A produces A$1.class
    • We’ll see these later

• Why are inner classes useful?
IntegerStack Client

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

```java
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

```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

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

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:
  static <T> T id(T x) { return x; }
  Integer j = id(new Integer(3));

  – There’s no need to explicitly instantiate id; compiler figures out the correct type.
  • In contrast, consider
    List<Integer> list = new ArrayList<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> { ... }

• But they didn’t change the JVM to add generics
  – So how does that work?
  – Will answer this question shortly.
Subtyping for Generics

• Is Stack<Integer> a subtype of Stack<Object>?
  – The following code seems OK:

```java
int count(Collection<Object> c) {
    int j = 0;
    Iterator<Object> i = c.iterator();
    while (i.hasNext()) {
        Object e = i.next();
        j++;
    }
    return j;
}
```

• But I’m not allowed to call count(x) where x has type Stack<Integer>
• Let’s a take a step back and consider arrays …

Subtyping and Arrays

• Java has a 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

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)
}

• 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

<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
• Only works for methods, which can instantiate the type differently at each call site.
  – What should Class.forName(String) return?
Solution II: Wildcards

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

- Use `?` as the type variable
  - `Collection<?>` is "Collection of unknown"
- Why is this safe?

Legal Wildcard Usage

- Reasonable question:
  - Why is `Stack<Integer>` not a subtype of `Stack<Object>`, but `Stack<Integer>` is a subtype of `Stack<?>`? In both cases, I have to cast the Stack’s elements to type `Object`.
- Answer:
  - Loosely speaking: wildcards permit reading but not writing.
  - In general, if a generic class `C` is declared as
    ```java
class C<T> { ... }
    ```
  - When called on a `C<?>`, methods that return `T` can have these values cast to `Object`, but a method that takes `T` as an argument can only be given null.
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; }
```

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
        s.draw();
}
```
Bounded Wildcards

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

```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`

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 (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 always compatible with `Circle`

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?
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; you could not declare a variable with this bound without wildcards.

Bounding and Wildcards

• Our legal wildcard rule from earlier can be refined to include bounds:
  – In general, if a generic class C is declared as

    ```java
class C<T extends B> { ... }
```

  – When called on a C<?>, methods that return T can have these values cast to B, but a method that takes T as an argument can only be given null.
Exercise: Annotate Java Libraries

- Look at the Java 1.4 API, and figure out how you would best annotate the following classes
  - Collection
  - Comparator
  - Collections
  - Class

  - Look at others too!

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());

- Uh...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

- Thus, 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