Detailed Example

- Shows
  - polymorphism for both method receiver and arguments
  - static vs. instance methods
  - overriding instance variables

Source code for classes

```
class A {
        String f(A x) { return "A.f(A) "; }
        String f(B x) { return "A.f(B) "; }
  static String g(A x) { return "A.g(A) "; }
  static String g(B x) { return "A.g(B) "; }
        String h = "A.h";
        String getH() {return "A.getH(): " + h; }
}
class B extends A {
        String f(A x) { return "B.f(A)/ " + super.f(x); }
        String f(B x) { return "B.f(B)/ " + super.f(x); }
  static String g(A x) { return "B.g(A) "; }
  static String g(B x) { return "B.g(B) "; }
        String h = "B.h";
        String getH() {return "B.getH(): " + h + "/" + super.h; }
}
```


Invocation and results

```
A a = new A();  A ab = new B();  B b = new B();

System.out.println( a.f(a) + a.f(ab) + a.f(b) );
System.out.println( ab.f(a) + ab.f(ab) + ab.f(b) );
System.out.println( b.f(a) + b.f(ab) + b.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B)/A.f(B)
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B) A.f(B)

System.out.println( a.g(a) + a.g(ab) + a.g(b) );
System.out.println( ab.g(a) + ab.g(ab) + ab.g(b) );
System.out.println( b.g(a) + b.g(ab) + b.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)

System.out.println( a.h + "  " + a.getH() );
System.out.println( ab.h + "  " + ab.getH() );
System.out.println( b.h + "  " + b.getH() );
// A.h  A.getH():A.h
// A.h  B.getH():B.h/A.h
// B.h  B.getH():B.h/A.h
```


What to notice

- Invoking **ab.f(ab)** invokes **B.f(A)**
  - run-time type of object determines method invoked
  - compile-time type of arguments used
- **ab.h** gives the **A** version of **h**
- **ab.getH()**
  - **B.getH()** method invoked
  - in **B.getH()**, **h** gives **B** version of **h**
- Use of **super** in class **B** to reach **A** version of methods/variables
- **super** not allowed in static methods

Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can/should be **this(args)** or **super(args)**
  - if those are omitted, **super()** is called
  - must be very first statement, even before variable declarations
- *not* used for type conversions or assignments
- void constructor generated if no constructors given

Static class components

- They belong to the class
  - static variables allocated once, no matter how many objects created
  - static methods are not specific to any class instance, so can't refer to **this** or **super**
- Can reference class variables and methods through either class name or an object ref
  - better to reference via object references

Interfaces

- An interface is an object type – no associated code or instance variables
  - only describes methods supported by interface
- A class can *implement* (be a subtype of) many interfaces
- Interfaces may have final static variables
  - to define a set of constants

Interface example

```
public interface Comparable {
        public int compareTo(Object o)
}
public class Util {
        public static void sort(Comparable []) { … }
}
public class Choices implements Comparable {
        public int compareTo(Object o) {
                return … ;
        }
}
…
        Choices [] options = … ;
        Util.sort(options);
…
```


No multiple inheritance

- A class type can be a subtype of many other types (**implements**)
- But can only inherit method implementations from one superclass (**extends**)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often badly used
- And it's complicated to implement well

Garbage collection

- Objects that are no longer accessible can be garbage collected
- Method **void finalize()** called when an object is collected
  - best to avoid using it, since no way to tell when it will get called
- Garbage collection not a major performance bottleneck
  - **new/delete** in C++ can be expensive too

Class Objects

- For each class, there is an object of type **Class**
- Describes the class as a whole
  - used extensively in Reflection package
- **Class.forName("MyClass")**
  - returns class object for **MyClass**
  - will load **MyClass** if needed
- **Class.forName("MyClass").newInstance()**
  - creates a new instance of **MyClass**
- **MyClass.class** gives the **Class** object for **MyClass**

Types

- A type describes a set of values that can be:
  - held in a variable
  - returned in an expression
- Types include:
  - primitive types – boolean, char, short, int , …
  - Reference types:
    - Class types
    - Array types
    - Interface types

Class types

- Using the name of a class as a type means that a reference to an instance of that class or a subclass is a permitted value
  - a subclass has *all* the fields of its superclass
  - a subclass has *all* the methods of its superclass
- **null** is also an allowed value

Array types

- If **S** is a subtype of **T**
  - **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types
- Storing into an array generates a run-time check that the type stored is a subtype of the *declared* type of the array elements
- Performance penalty?
- Similar (and maybe worse) problems in C++

Example: Object[]

```
public class TestArrayTypes {
        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;
                }
        }
        public static void main(String [] args) {
                reverseArray(args);
                for(int i=0; i < A.length; i++)
                        System.out.println(args[i]);
        }
}
```


Interface types

- Using the name of an interface as a type means
  - a reference to any instance of a class that implements the interface is a permitted value
  - **null** is also allowed
- Object referenced is guaranteed to support *all* the methods of the interface
  - invoking a method on an interface might be a bit less efficient

Object Obligations

- many operations have default implementations
  - which may not be the ones you want

```
public boolean equals (Object that) { … } // return this == that
public String toString() { … } // returns print representation
public int hashCode() { … } // key for accessing object
// important that a.equals(b) implies a.hashCode()==b.hashCode()
public void finalize() { … } // called before object garbage is
// collected, default is {}
public Object clone() { … } // default is shallow bit-copy if class
// implements Cloneable, throw CloneNotSupportedException
// otherwise
```


Poor man's polymorphism

- Every object is an **Object**
- An **Object[]** can hold references to any objects
- E.g., for a data structure **Set** that holds a set of **Object**
  - can use it for a set of **String**
  - or a set of images
  - or a set of anything
- Java's container classes are all containers of **Object**
  - when you get a value out, have to downcast it

Interacting with External Environment

Applications and I/O

- Java external interface is a public class
- via **public static void main(String [] args)**
- **args[0]** is first argument
  - unlike C/C++
- **System.out** and **System.err** are **PrintStreams**'s
  - should be **PrintWriter**'s, but would break 1.0 code
  - **System.out.print(…)** prints a string
  - **System.out.println(…)** prints a string with a newline
- **System.in** is an **InputStream**
  - not quite so easy to use

Input (JDK 1.1 and higher)

- Wrap **System.in** in an **InputStreamReader**
  - converts from bytes to characters
- Wrap the result in a **BufferedReader**
  - makes input operations efficient
  - supports **readline()** interface
- **readline()** returns a string
  - returns **null** if at EOF

Example Echo Application

```
import java.io.*;
public class Echo {
   public static void main(String [] args) {
        String s;
        BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
        int i = 1;
        try {
                while((s = in.readLine()) != null)
                        System.out.println((i++) + ": " + s); }
        catch(IOException e) {
                System.out.println(e); }
   }
}
```