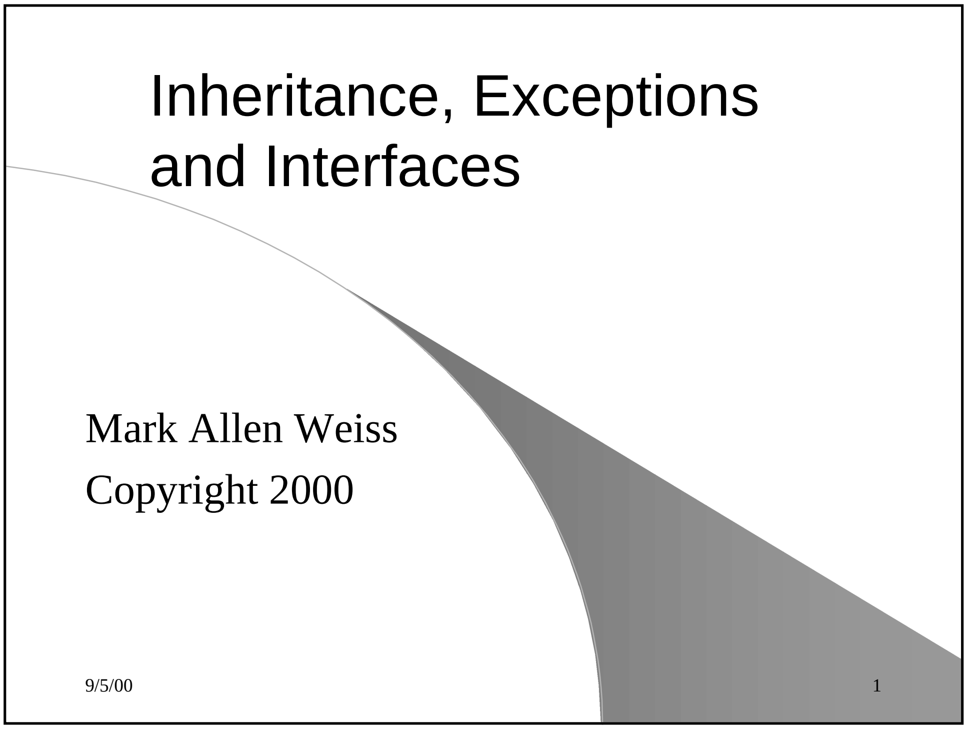
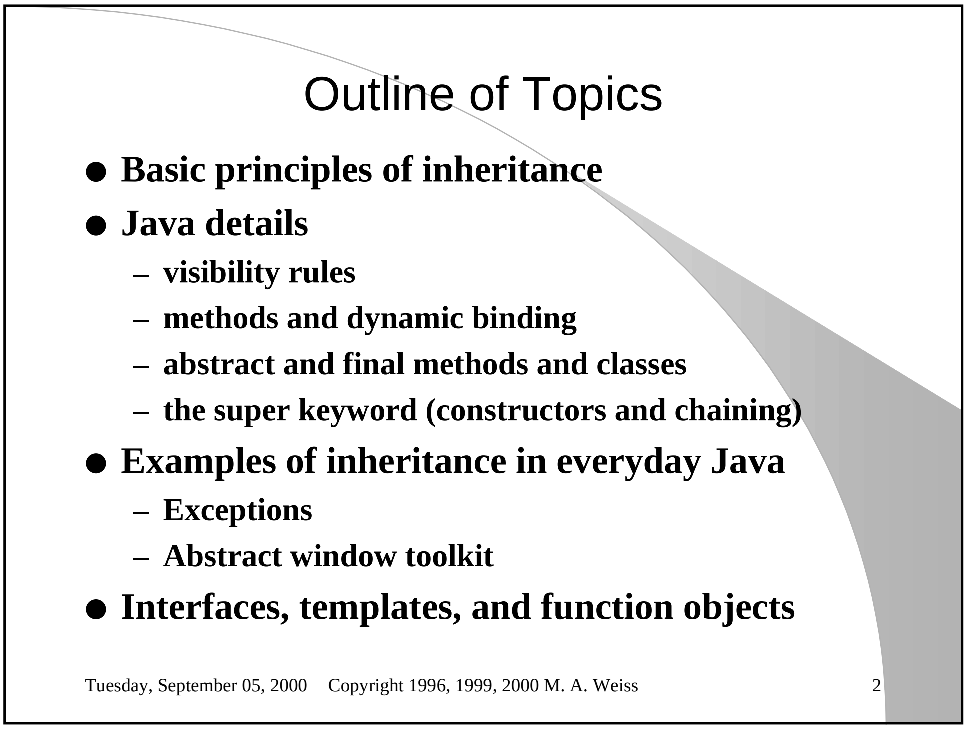


Inheritance, Exceptions and Interfaces

Mark Allen Weiss

Copyright 2000



Outline of Topics

- **Basic principles of inheritance**
- **Java details**
  - **visibility rules**
  - **methods and dynamic binding**
  - **abstract and final methods and classes**
  - **the super keyword (constructors and chaining)**
- **Examples of inheritance in everyday Java**
  - **Exceptions**
  - **Abstract window toolkit**
- **Interfaces, templates, and function objects**

Inheritance

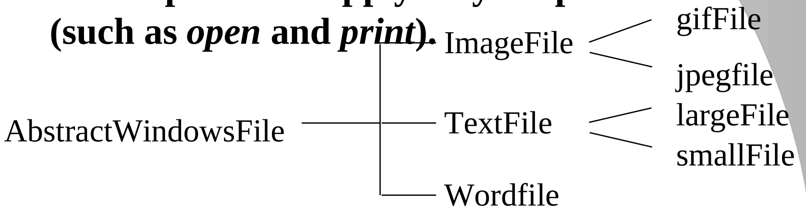
- **Allows the creation of new types with additional properties of the original type.**
- **When writing the code to define the new type, we should not alter any of the code for the original type (don't break what already works).**
- **Inheritance typifies an IS-A relationship.**
- **Basic mechanism for code reuse.**
  - **Direct reuse: get new classes from old without cut-and-paste**
  - **Indirect reuse: existing routines work with new classes automatically**

Polymorphism

- **A *polymorphic* reference type can reference objects of several different types.**
- **When operations are applied to the polymorphic type, the operation appropriate to the actual referenced object is automatically selected.**
- **Windows example: double clicking on an icon calls an appropriate open function, depending on the type of file (word document, html document, etc.). *WindowsFile* is the polymorphic object, and it can encompass various different types of files.**

Windows File Example

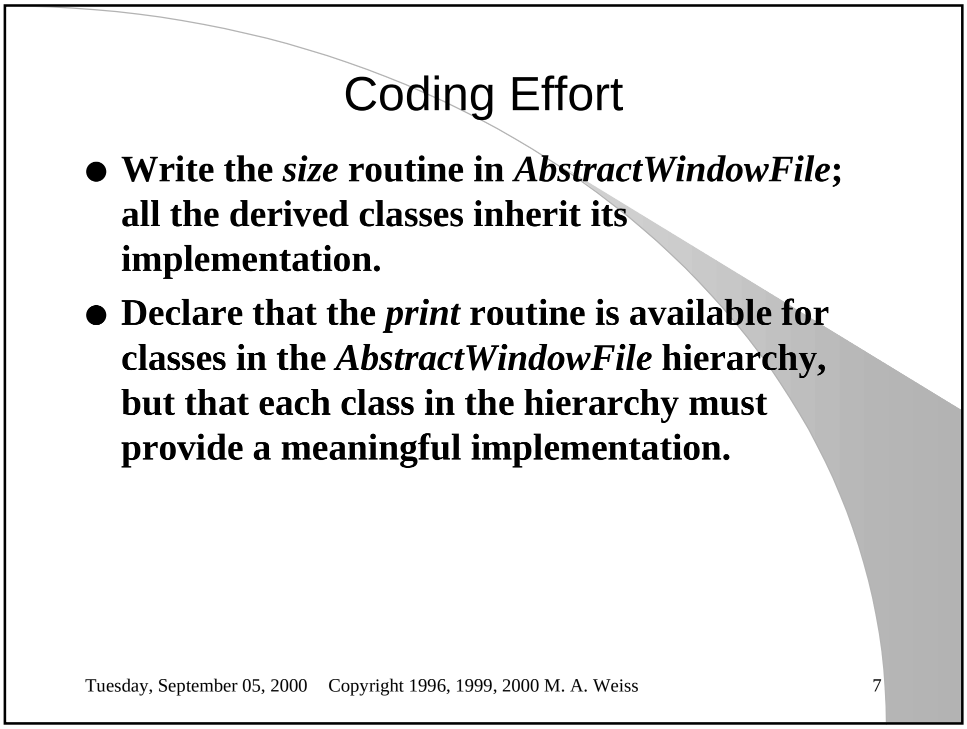
- ***AbstractWindowsFile* could be considered a class.**
- **We could have various extensions (also classes).**
- **Some operations in *AbstractWindowsFile* apply throughout (e.g. *sizeOfFile*).**
- **Some operations apply only to specific files (such as *open* and *print*).**



Polymorphic Behavior

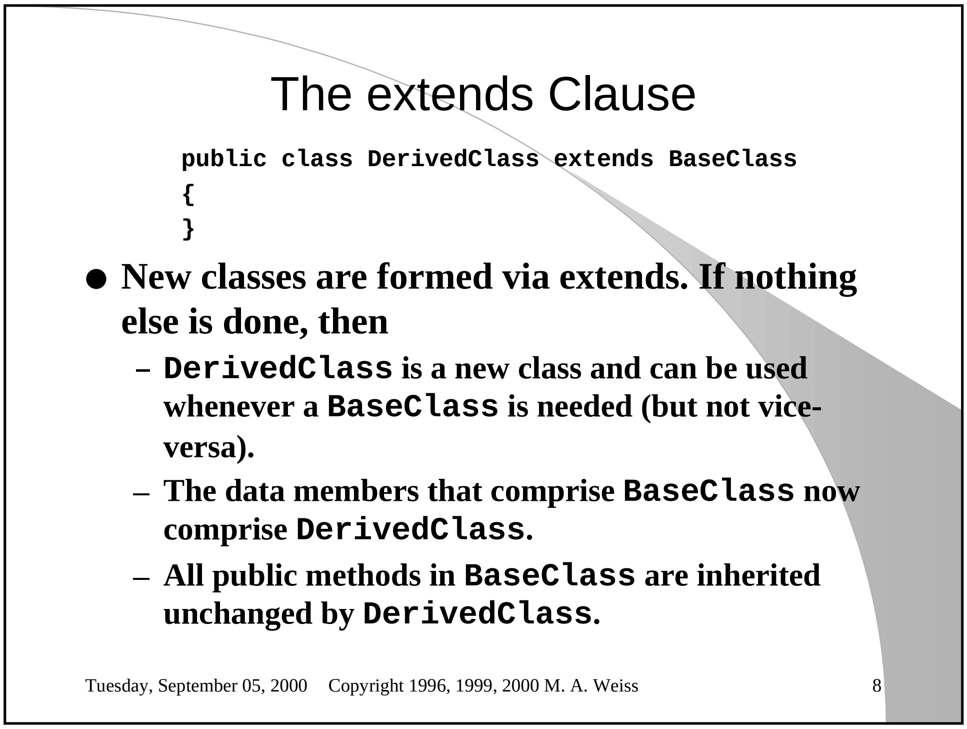
```
AbstractWindowFile f;

if( blah )
    f = new MSWord( "image.doc" );
else
    f = new NotePad( "image.txt" );
f.print( );   // Should call correct print
System.out.println( f.size( ) ); // only 1 size
```

- **Polymorphic behavior such as `print` will involve a run-time decision.**
- **However, `size` is the same for any file, and does not require a run-time decision.**



Coding Effort

- **Write the *size* routine in *AbstractWindowFile*; all the derived classes inherit its implementation.**
- **Declare that the *print* routine is available for classes in the *AbstractWindowFile* hierarchy, but that each class in the hierarchy must provide a meaningful implementation.**



The extends Clause

```
public class DerivedClass extends BaseClass
{
}
```

- **New classes are formed via extends. If nothing else is done, then**
  - **`DerivedClass` is a new class and can be used whenever a `BaseClass` is needed (but not vice-versa).**
  - **The data members that comprise `BaseClass` now comprise `DerivedClass`.**
  - **All public methods in `BaseClass` are inherited unchanged by `DerivedClass`.**

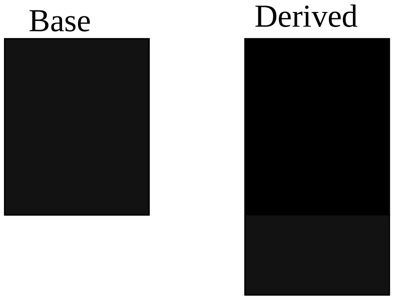
Derived Class Data

- **Derived class can add additional data members.**
- **It cannot remove data members.**

Data Layout for Inheritance

- **If we have**

```
class Derived extends Base
{
  private int newData;
}
```



Only Base class can access private Base data. Only Derived class can access private Derived data

Visibility

- **private methods and data in the base class are not accessible in derived class. The following does not work:**

```
class Base
{
  private int x;
    // Other stuff omitted.
}
class Derived extends Base
{
    // Derived has a data member x, inherited, but
  public int getX( ) { return x; }      // this fails
}
```

Name-Clashed Data Is Kept Separate Even If Public

```
class Person {
  int age = 37;
  int getAge( )
    { return age; } // Always uses Person::age
}

class OldPerson extends Person {
  int age = 99;
  int setAge( )
    { age = 50; }  // Always uses OldPerson::age

  public static void main( String [] args ) {
    OldPerson p = new OldPerson( );
    p.setAge( );
    System.out.println( p.getAge( ) );
  }
}
```

Derived Class Methods

```
public class Derived extends Base
{
    public  void method1( ) { yadaYada( ); }
    private void method2( ) { }
 }
```

- *public methods*: `method1` is now defined for class Derived. If an identical method (same signature) was defined for `Base`, it is overridden for `Derived` objects. Behavior is polymorphic.
- *private methods*: In C++ if `method2` was defined for `Base`, it is now disabled for `Derived`. In Java this is illegal.

Inheritance and Visibility

- **Inheritances typifies IS-A relationship. Everything base can do, derived can do, plus possibly more.**
- **CAN NEVER REDUCE VISIBILTY WHEN OVERRIDING.**
- **Cannot override instance method with static method and vice versa**

protected

- **protected members can be accessed in derived class**
- **They can also be accessed by other methods in any class that is in the same package**
- **In previous example, if `x` was protected, the `x` member of `Derived` would be accessible by derived.**
- **Generally, it's best to avoid protected; use base class accessors if needed.**

Final Methods and Classes

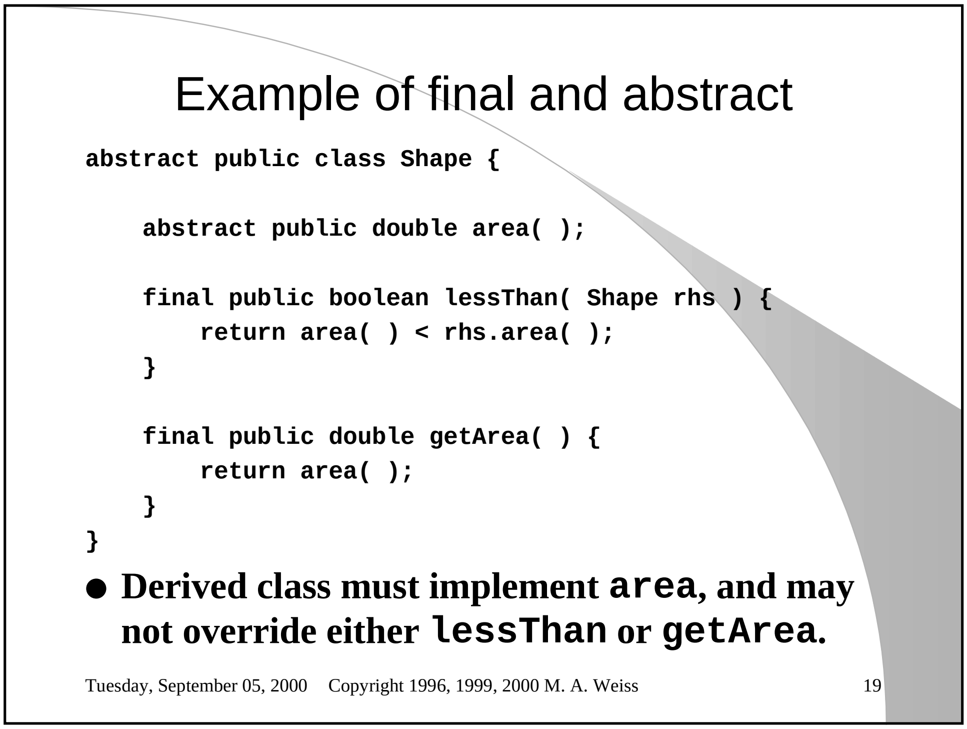
- **A *final method* cannot be overridden.**
- **A final method indicates to readers of the code that the method is invariant over the inheritance hierarchy. Example: the SizeOf routine for the *AbstractWindowsFile*.**
- **Declaring a method final prevents the derived class from erroneously redefining a class method.**
- **Declaring a method final allows the compiler to perform inline optimization.**
- ***Final classes* cannot be extended.**

Dynamic Binding

- Not applied for static methods, private methods, or final methods
- Two step algorithm:
  - Compiler deduces signature of appropriate method based on static types of parameters
  - VM walks path up from dynamic type until it reaches Object; first class that has the method being searched for is last overriding implementation and is used
  - If no class is found an exception is thrown
- Implements single dispatch

Abstract Methods and Classes

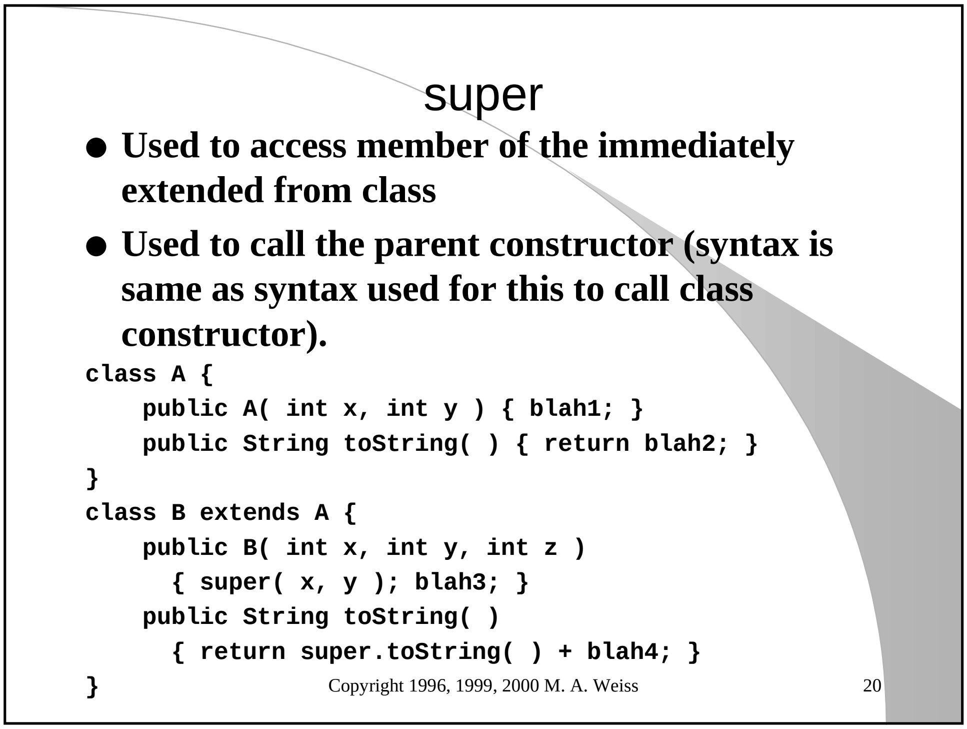
- An *abstract method* is a method that cannot be reasonably defined for a class, but makes sense for the class' extensions. Example: the displayFile routine for *AbstractWindowsFile*.
- Abstract method is a placeholder.
- Any class with an abstract method is an *abstract class*.
- An abstract class cannot be instantiated.
- A subclass of an abstract class is abstract unless it overrides all abstract methods.



Example of final and abstract

```
abstract public class Shape {

    abstract public double area( );

    final public boolean lessThan( Shape rhs ) {
        return area( ) < rhs.area( );
    }

    final public double getArea( ) {
        return area( );
    }
}
```

- **Derived class must implement `area`, and may not override either `lessThan` or `getArea`.**



super

- **Used to access member of the immediately extended from class**
- **Used to call the parent constructor (syntax is same as syntax used for this to call class constructor).**

```
class A {
    public A( int x, int y ) { blah1; }
    public String toString( ) { return blah2; }
}
class B extends A {
    public B( int x, int y, int z )
      { super( x, y ); blah3; }
    public String toString( )
      { return super.toString( ) + blah4; }
}
```

Polymorphism Example

- **A `Shape`, `Circle`, and `Square` class:**

```
public abstract class Shape
{
  public abstract double area( );
}
public class Circle extends Shape
{
  public Circle( double rad )  { radius = rad; }
  public String toString( )    ( return “Circle: ” + radius; );
  public double area( )        { return Math.PI * radius * radius; }
  private double radius;
}
public class Square extends Shape
{
  public Square( double s )    { side = s; }
  public String toString( )    ( return “Side: ” + side; );
  public double area( )        { return side * side; }
  private double side;
}
```

Can Print Area for a Collection

```
public class UseShapes
{
  public static void main( String[] args )
  {
    Shape[] s = new Shape[ 3 ];
    s[ 0 ] = new Circle( 4.0 );
    s[ 1 ] = new Square( 5.0 );
    s[ 2 ] = new Circle( 2.5 );
    printAreas( s );
  }

  public static void printAreas( Shape[] arr )
  {
    for( int i = 0; i < arr.length; i++ )
      if( arr[ i ] != null )
        System.out.println( arr[ i ] + “ ” + arr[ i ].area( ) );
  }
}
```

Analysis of Example

- **`area` for `Shape` class used simply as a placeholder so we can call `area` for both `Circle` and `Square` using dynamic binding**
- **Can add a new class (e.g. `Rectangle`)**
  - **without any change to `Shape`**
  - **without any change to `printAreas` (indirect code reuse)**
  - **only have to write `Rectangle`, and have it extend `Shape`**
- **`instanceof` operator not used**

Type Compatibility

- **A reference to a base-class type can be used to access an object of a derived class.**
- **Can only select members that make sense for the static type of a reference.**
  - **May need to down cast**
  - **Cast will be checked at run time for validity**

Type Compatibility Examples

```
class Base
{
  public void foo( ) { ... }
}
class Derived extends Base
{
  public void bar( ) { ... }
}

Derived d1 = new Base( );   // Illegal
Base b1 = new Derived( );   // Legal
Derived d2 = (Derived) b1;  // Legal; cast required!
d2.bar( );                  // Legal
((Base) b1).bar( );         // Legal; cast required
Base b3 = new Base( );      // Legal of course
((Derived) b3).bar( );      // Legal; throws a runtime exception
                            // because dynamic type is not Derived
```

Immutable Object Pattern

- **Can use inheritance to control mutability:**

```
class ImmutablePerson
{
  // only accessors
}
class Person extends ImmutablePerson
{
  // adds mutators
}

  void printPersons( ImmutablePerson[] p )
  {
    // Of course, can downcast here, but that’s
    // same as casting away constness in C++.
  }
```

`Object` class

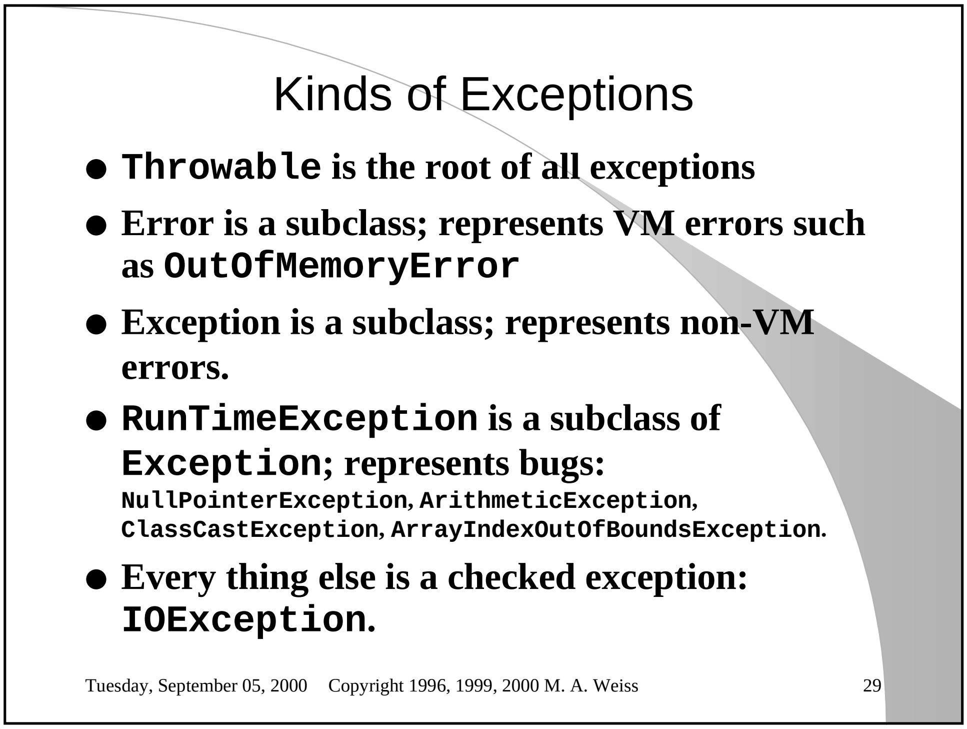
- **Root of all inheritance**
- **Defines several useful methods:**
  - `toString`
  - `equals`
  - `getClass`
- **Defines some tricky stuff**
  - `clone`
  - `finalize`
- **Thread stuff**
  - `wait`, `notify`, `notifyAll`

Exceptions

- **Exceptions are an example of Java's use of inheritance.**
- **An exception is thrown for an exceptional circumstance (bad file, array out of bounds, etc.).**
- **Syntax:**

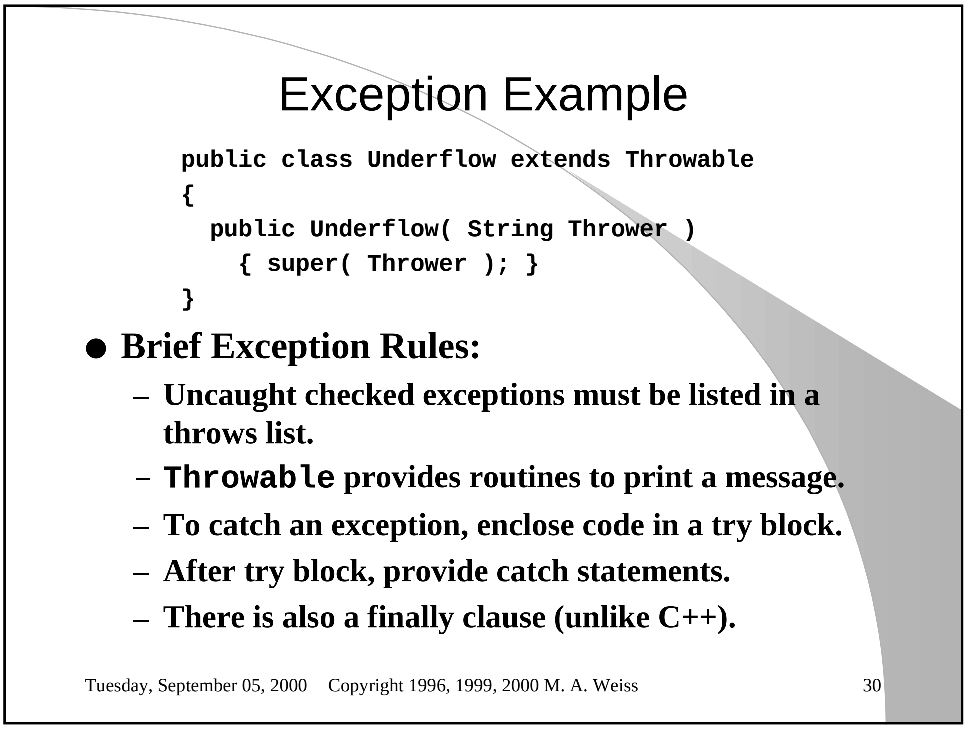
```
throw AnyExceptionObject;
```

- **Only objects that are subclasses of `java.lang.Throwable` may be thrown.**
- **Semantics are similar to C++; exceptions propagate back until caught.**



Kinds of Exceptions

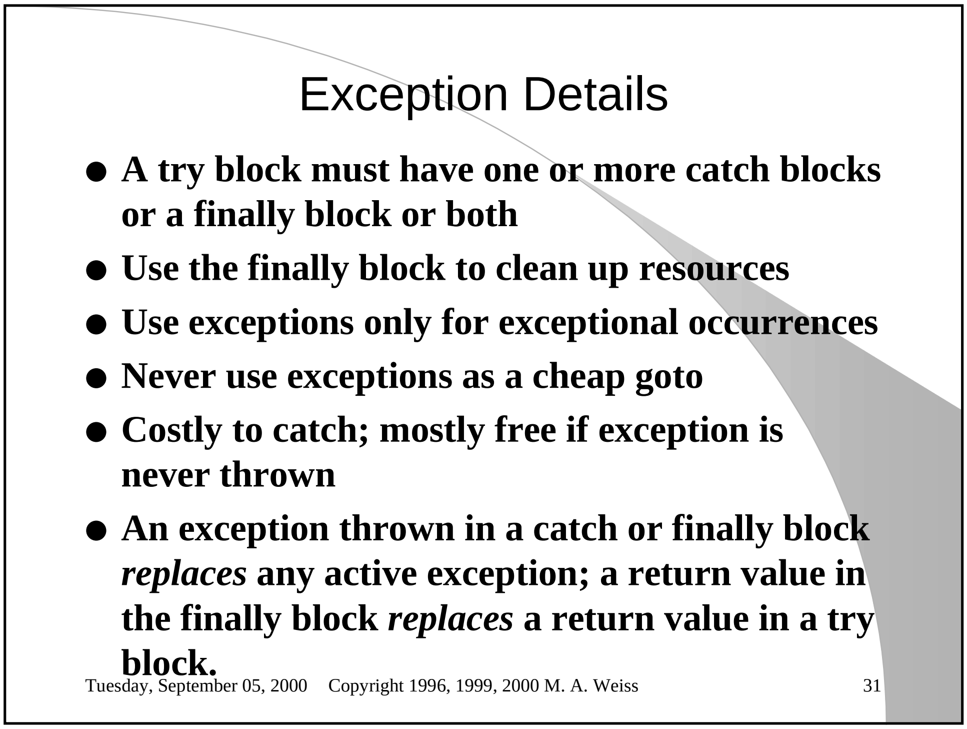
- `Throwable` **is the root of all exceptions**
- **Error is a subclass; represents VM errors such as** `OutOfMemoryError`
- **Exception is a subclass; represents non-VM errors.**
- `RunTimeException` **is a subclass of** `Exception`**; represents bugs:** `NullPointerException`, `ArithmeticException`, `ClassCastException`, `ArrayIndexOutOfBoundsException`.
- **Every thing else is a checked exception:** `IOException`**.**



Exception Example

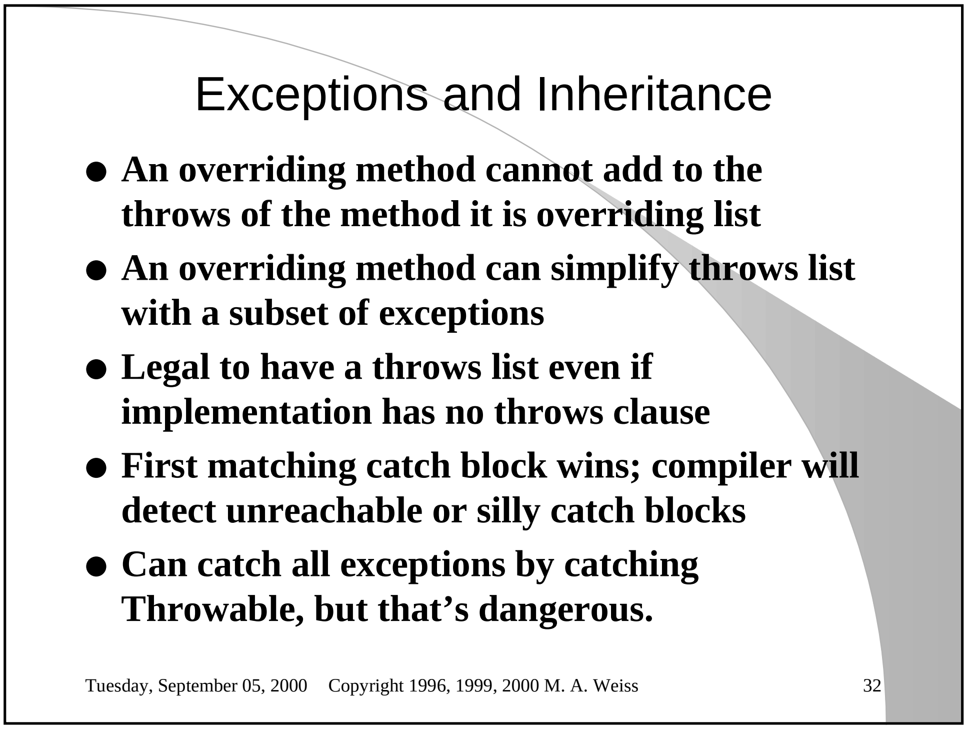
```
public class Underflow extends Throwable
{
  public Underflow( String Thrower )
    { super( Thrower ); }
}
```

- **Brief Exception Rules:**
  - **Uncaught checked exceptions must be listed in a throws list.**
  - `Throwable` **provides routines to print a message.**
  - **To catch an exception, enclose code in a try block.**
  - **After try block, provide catch statements.**
  - **There is also a finally clause (unlike C++).**



Exception Details

- **A try block must have one or more catch blocks or a finally block or both**
- **Use the finally block to clean up resources**
- **Use exceptions only for exceptional occurrences**
- **Never use exceptions as a cheap goto**
- **Costly to catch; mostly free if exception is never thrown**
- **An exception thrown in a catch or finally block *replaces* any active exception; a return value in the finally block *replaces* a return value in a try block.**

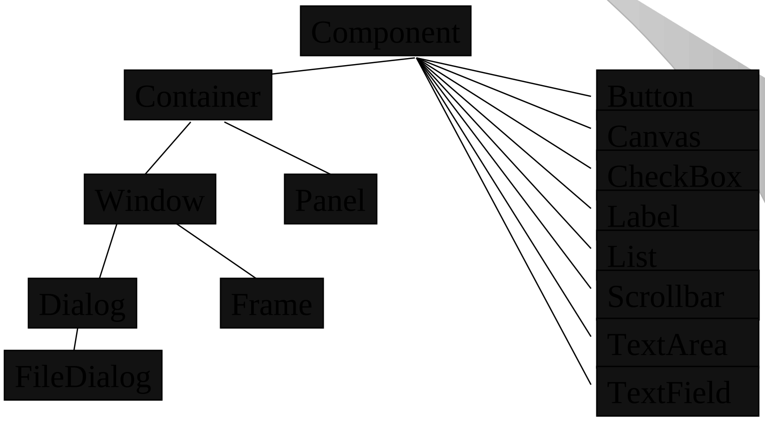


Exceptions and Inheritance

- **An overriding method cannot add to the throws of the method it is overriding list**
- **An overriding method can simplify throws list with a subset of exceptions**
- **Legal to have a throws list even if implementation has no throws clause**
- **First matching catch block wins; compiler will detect unreachable or silly catch blocks**
- **Can catch all exceptions by catching Throwable, but that’s dangerous.**

Abstract Window Toolkit

- **Makes heavy use of inheritance.**
- **Details eventually**



Component

Container

Window

Panel

Dialog

Frame

FileDialog

Button

Canvas

CheckBox

Label

List

Scrollbar

TextArea

TextField

Interfaces

- **No multiple inheritance in Java; the alternative is the *interface*.**
- **An interface is an abstract class that defines no non-abstract methods. The word interface replaces abstract class.**
- **Interface can also contain constants.**

```
public interface Drawable
{
     // automatically public and abstract
  void setColor( Color c );
}
```

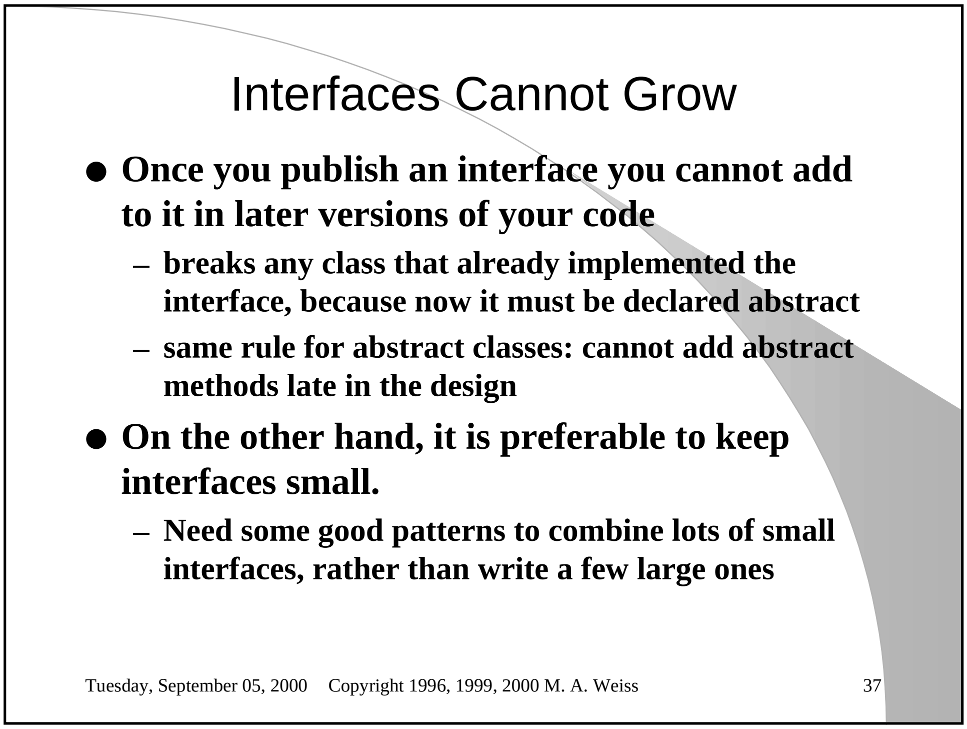
Using Interfaces

- Class implements an interface
- No limit on the number of interfaces implemented
- A class that implements an interface `X` may be used wherever it could be used if it extended class `X`.

```
public class Rectangle implements Drawable
{
    // Normal Rectangle stuff
    // Must then provide an implementation of setColor
    public void setColor( )
      { blah; }
}
```

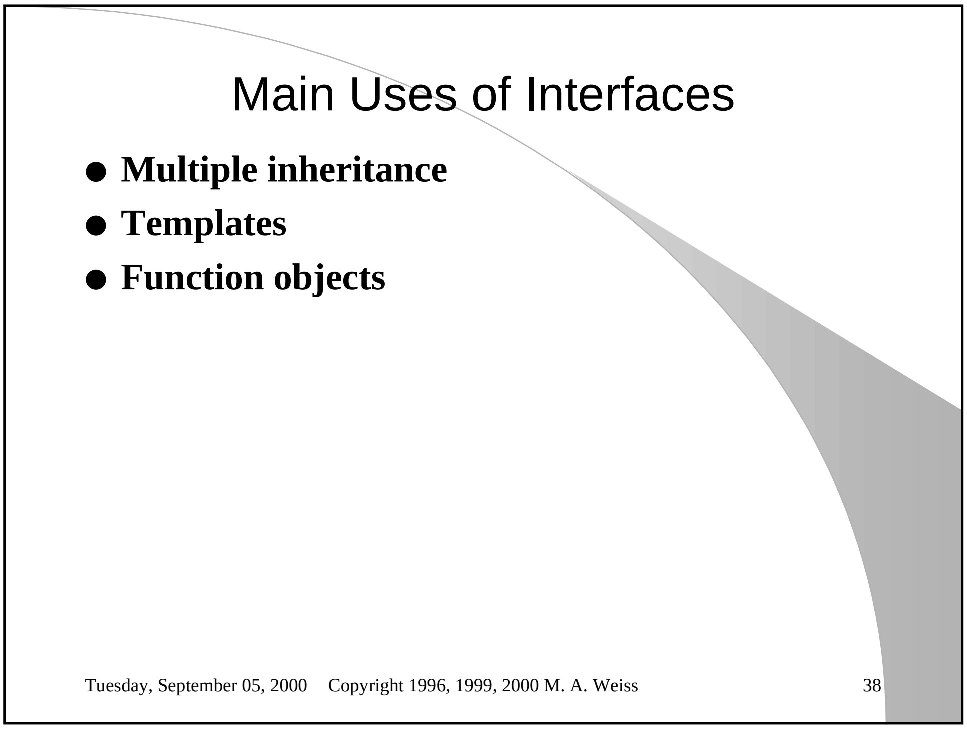
Interfaces are Abstract Classes

- Compiler will generate a .class file
- Only public interfaces visible outside of package, and must be in file of the same name
- Class that implements an interface satisfies the IS-A property, and objects of that type satisfy `instanceof`
- Class that implements interface must declare all interface methods public (why?)
- Class that implements interface but not all methods must be declare abstract
- Can extend interfaces (even multiple interfaces)



Interfaces Cannot Grow

- **Once you publish an interface you cannot add to it in later versions of your code**
  - **breaks any class that already implemented the interface, because now it must be declared abstract**
  - **same rule for abstract classes: cannot add abstract methods late in the design**
- **On the other hand, it is preferable to keep interfaces small.**
  - **Need some good patterns to combine lots of small interfaces, rather than write a few large ones**



Main Uses of Interfaces

- **Multiple inheritance**
- **Templates**
- **Function objects**

Multiple Implementation Inheritance

- **Difficult to do correctly**
  - **Often inherit conflicting implementations**
  - **Need more syntax ; what does super mean?**
  - **Supported in C++, and is very confusing**
  - **C++ experts recommend only multiple interface inheritance (inherit functionality but not implementations)**
- **Multiple implementation inheritance illegal in Java; can only extend one class.**
- **Multiple interface inheritance is legal; formalizes the advice of C++ experts**

Templates

- **Java does not support templates.**
- **They can be faked using inheritance: use `Object` as the class.**

```
class ObjCell
{
    public Object read( ) {
        return storedValue;
    }
    public void write( Object x ) {
        storedValue = x;
    }
    private Object storedValue;
}
```

Two Problems

- **Built-in-type is not an `Object`, so `ObjCell` cannot store an int, for instance. Solution: use *wrapper classes* such as `Integer`, `Double`, etc., which are predefined.**
- **If an `Integer` is stored in the `ObjCell` `m`, then the statement below does not work, because the method returns an `Object`, and an `Object` is not an `Integer`. Solution involves a type conversion.**

```
    Integer x = m.read( );   // Wrong!!
    Integer x = (Integer) (m.read( )); // OK
```

Using the `ObjCell` for ints

```
class TestObjCell {
    static public void main( String args[ ] ) {
        ObjCell m = new ObjCell( );
        m.write( new Integer( 5 ) );
        System.out.println( "Cell contents are "
                            + (Integer)m.read( ) );
    }
}
```

- **All wrappers define a `toString` method.**
- **`Integer` defines an `intValue` method that returns an `int`.**
- **Wrappers are final classes (so methods are inlined, with little overhead for using them).**

Generic Algs That Require Functions

- Routines like sorting cannot simply take **`Object`**: how to apply sort?
- Define an interface, and algorithms can work on objects of the interface type.

```
interface Comparable {
  boolean compareTo( Object other );
}
public class Utils {
    // Can sort Objects that implement Comparable interface
  public void static sort( Comparable[] arr ) {
    // sorting algorithm that orders by tests such as
    if( arr[i].compareTo( arr[j] ) < 0 )
  }
}
```

Function Object

- Cannot pass a function as a parameter to a procedure; can only pass primitives and references
- Object = state plus methods
- Create a stateless object with the method you want to pass, and send the reference
- Three steps:
  - Agreed upon function is placed in an interface
  - Class implements the interface with function def
  - Object of that type created; ref to it is passed to the procedure, which can call function through the ref

Example: The Library Side

- **Define a `Comparator` interface (this is actually now in Java 1.2)**

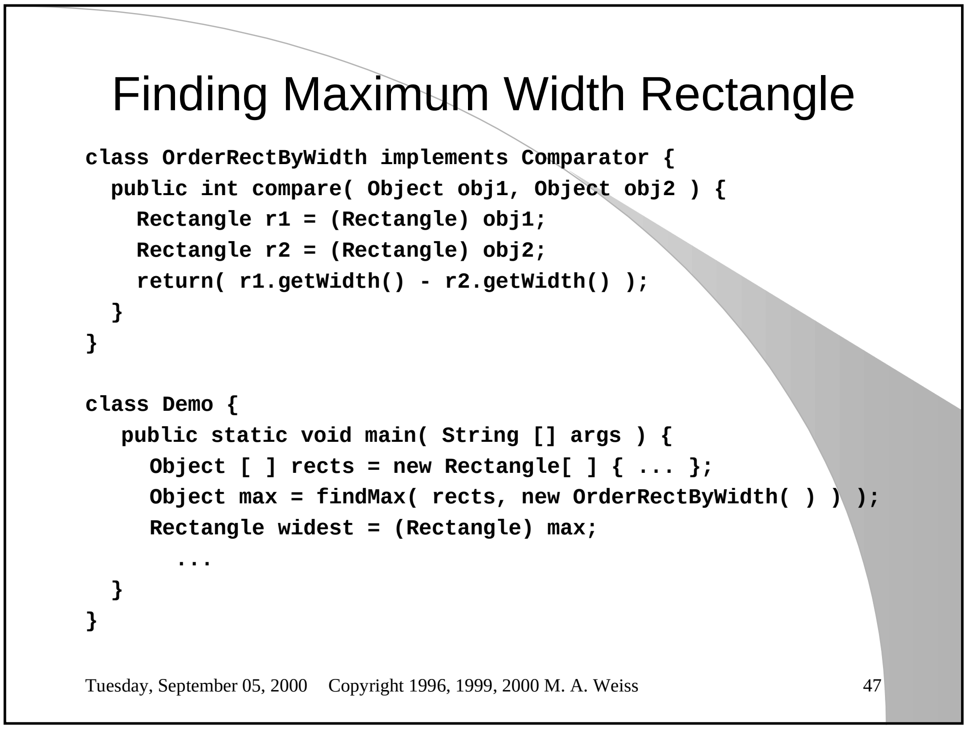
```
public interface Comparator {
   int compare( Object obj1, Object obj2 );
}
```

- **Implement generic routine using interface:**

```
class FindMaxDemo {
  public static Object findMax( Object[] a, Comparator cmp ) {
    int maxIndex = 0;
    for( int i = 1; i < a.length; i++ )
      if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
        maxIndex = i;
    return a[ maxIndex ];
   }
}
```

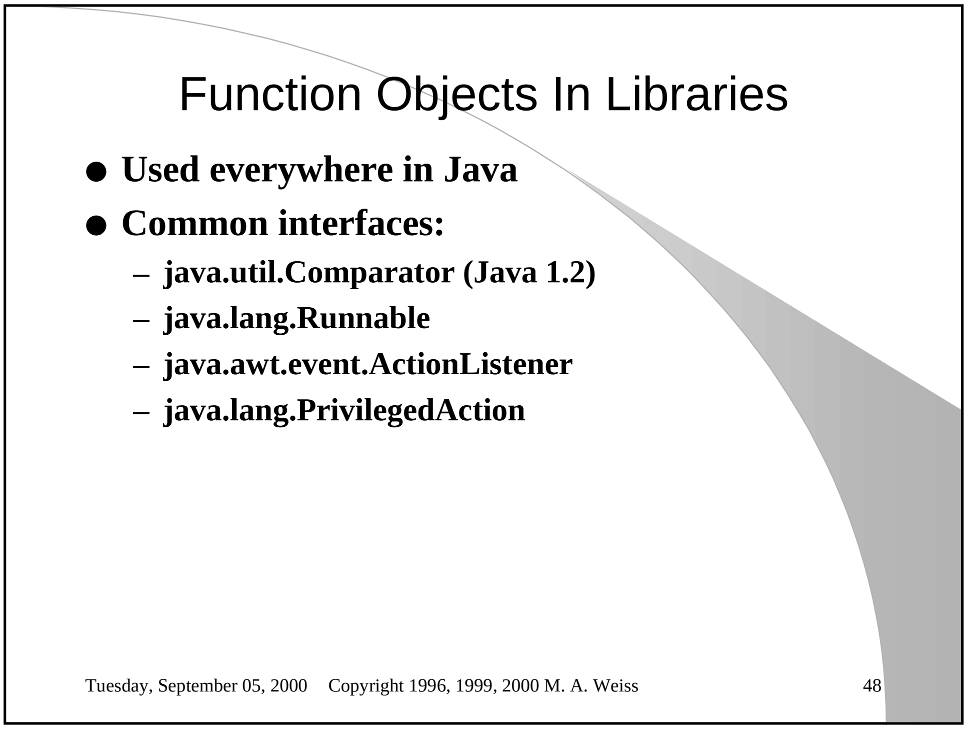
Using the Generic Routine

```
// Rectangle class; knows nothing about ordering
public class Rectangle
{
  public Rectangle( int l, int w ) {
    length = l; width = w;
  }
  public int getLength( ) {
    return length;
  }
  public int getWidth( ) {
    return width;
  }
   private int length;
   private int width;
}
```



Finding Maximum Width Rectangle

```
class OrderRectByWidth implements Comparator {
  public int compare( Object obj1, Object obj2 ) {
    Rectangle r1 = (Rectangle) obj1;
    Rectangle r2 = (Rectangle) obj2;
    return( r1.getWidth() - r2.getWidth() );
  }
}

class Demo {
   public static void main( String [] args ) {
     Object [ ] rects = new Rectangle[ ] { ... };
     Object max = findMax( rects, new OrderRectByWidth( ) ) );
     Rectangle widest = (Rectangle) max;
       ...
  }
}
```



Function Objects In Libraries

- **Used everywhere in Java**
- **Common interfaces:**
  - **java.util.Comparator (Java 1.2)**
  - **java.lang.Runnable**
  - **java.awt.event.ActionListener**
  - **java.lang.PrivilegedAction**

Summary

- **Inheritance used:**
  - **good designs**
  - **exceptions**
  - **templates**
  - **function objects**
- **Can’t do any Java programming without inheritance**