

# Preliminaries II



# Agenda

## **Objects and classes**

- Encapsulation and information hiding
- Documentation
- Packages

## **Inheritance**

- Polymorphism
- Implementation of inheritance in Java
- Abstract classes
- Interfaces
- Generics

# Desirable qualities of software systems

- Usefulness
- Timeliness
- Reliability
- Maintainability
- Reusability
- User friendliness
- Efficiency

Not all these qualities are attainable at the same time, nor are they of equal importance.

# Object-oriented programming

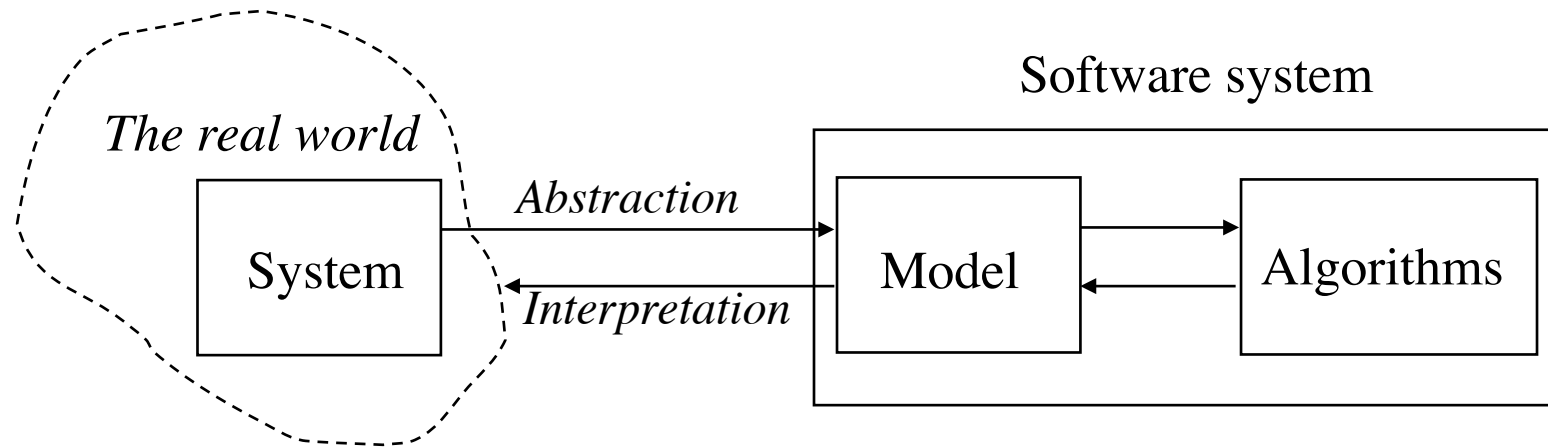
Focuses primarily on

- Reusability
- Maintainability

Maintainability is simplified by

- Flexibility (aspects are easily changeable)
- Simplicity
- Readability

# Software development



**Abstraction:** only essential and relevant parts are captured, others are ignored

# Basic concepts in object-oriented programming

## Object:

*Interpretation in the real world:*

An object represents anything in the real world that can be distinctly identified

*Representation in the model:*

An object has an *identity*, a *state*, and a *behavior*

# State and behavior

The **state** of an object is composed of a set of *fields*, or *attributes*. Each field has a name, a type, and a value.

The **behavior** of an object is defined by a set of *methods* that may operate on the object. In other words, a method may access or manipulate the state of the object.

The **features** of an object refer to the combination of the state and the behavior of the object.

# Classes

## **Class:**

*Interpretation in the real world:*

A *class* represents a set of objects with similar characteristics and behavior. These objects are called *instances* of the class.

*Representation in the model:*

A *class* characterizes the structure of states and behaviors that are shared by all its instances.



# Noun-verb analysis in object oriented design



Noun-verb analysis is a means of identifying classes and their methods.

If you look in the “problem statement” for *nouns*, they will often wind up as *classes*.

If you look for *verbs*, they will be *methods* of those classes.

# An example of a class

```
class Point {    // Class name
    int x, y;    // Fields

    void move(int dx, int dy) {    // Method
        x += dx;
        y += dy;
    }
}
```

A class is a template, blueprint, that defines what an object's fields and methods will be.

# A simple example

**figure 3.1**

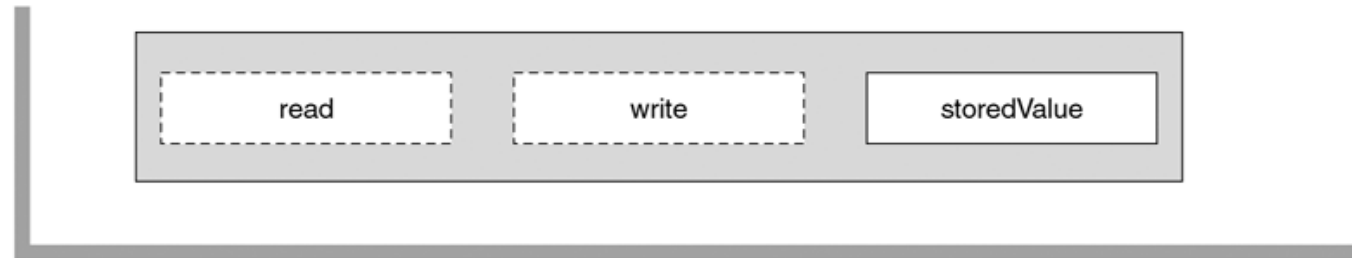
A complete  
declaration of an  
IntCell class

```
1 // IntCell class
2 // int read( )      --> Returns the stored value
3 // void write( int x ) --> x is stored
4
5 public class IntCell
6 {
7     // Public methods
8     public int read( )      { return storedValue; }
9     public void write( int x ) { storedValue = x; }
10
11     // Private internal data representation
12     private int storedValue;
13 }
```

# A view of the accessibility

**figure 3.2**

IntCell members:  
read and write are  
accessible, but  
storedValue is hidden.



# Using the class

```
1 // Exercise the IntCell class
2
3 public class TestIntCell
4 {
5     public static void main( String [ ] args )
6     {
7         IntCell m = new IntCell( );
8
9         m.write( 5 );
10        System.out.println( "Cell contents: " + m.read( ) );
11
12        // The next line would be illegal if uncommented
13        // because storedValue is a private member
14        // m.storedValue = 0;
15    }
16 }
```

**figure 3.3**

A simple test routine to show how IntCell objects are accessed

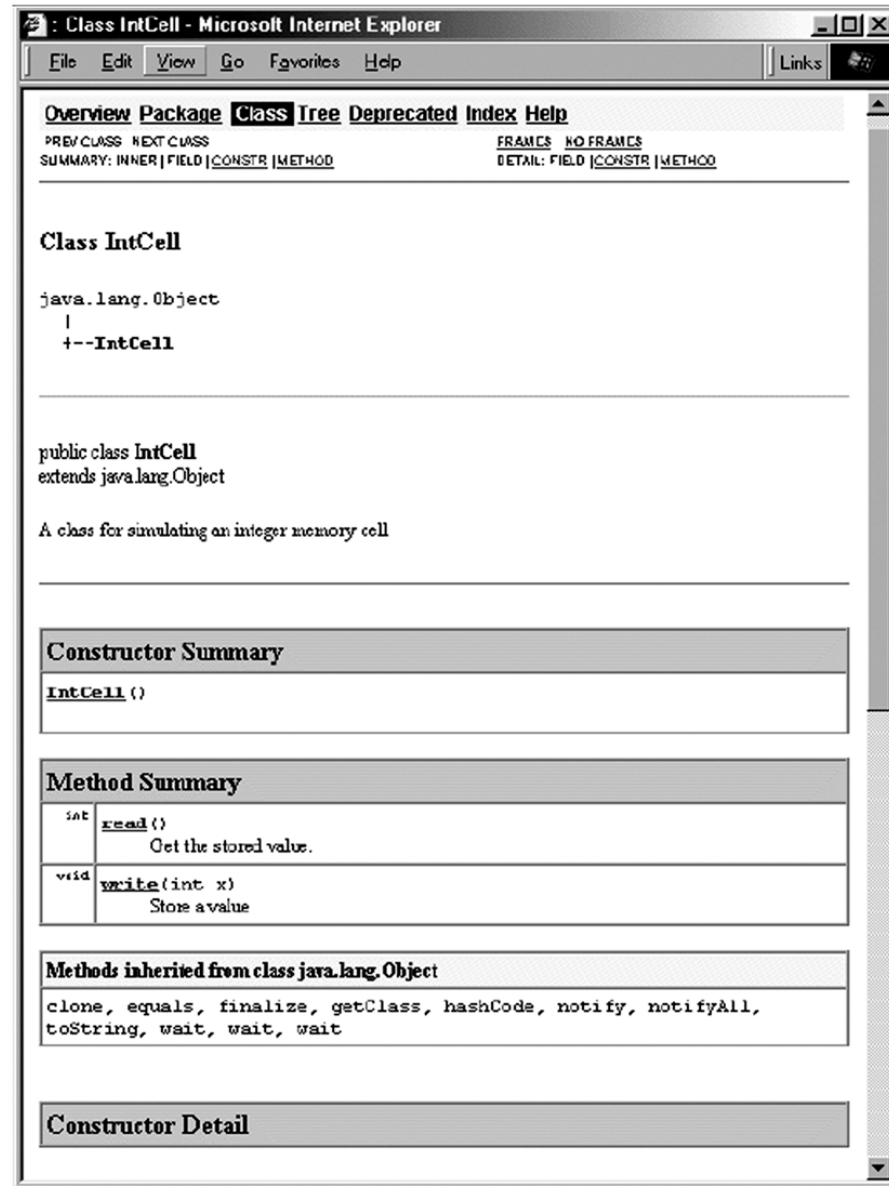
# Javadoc for the class

**figure 3.4**

The IntCell  
declaration with  
javadoc comments

```
1 /** ←----- opening tag (called begin-comment delimiter)
2  * A class for simulating an integer memory cell
3  * @author Mark A. Weiss
4  */
5
6 public class IntCell
7 {
8     /** ←----- opening tag
9     * Get the stored value.
10    * @return the stored value.
11    */
12    public int read( )
13    {
14        return storedValue;
15    }
16
17    /** ←----- opening tag
18    * Store a value.
19    * @param x the number to store.
20    */
21    public void write( int x )
22    {
23        storedValue = x;
24    }
25
26    private int storedValue;
27 }
```

Javadoc is a tool for  
generating API  
documentation in  
HTML format from  
comments in source  
code



**figure 3.5**  
*javadoc* output for Figure 3.4 (partial output)

# Javadoc tags

Tag & Parameter	Usage	Applies to
<b>@author</b> <i>John Smith</i>	Describes an author.	Class, Interface, Enum
<b>@version</b> <i>version</i>	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum
<b>@since</b> <i>since-text</i>	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method
<b>@see</b> <i>reference</i>	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method
<b>@param</b> <i>name description</i>	Describes a method parameter.	Method
<b>@return</b> <i>description</i>	Describes the return value.	Method
<b>@exception</b> <i>classname description</i> <b>@throws</b> <i>classname description</i>	Describes an exception that may be thrown from this method.	Method
<b>@deprecated</b> <i>description</i>	Describes an outdated method.	Class, Interface, Enum, Field, Method



# Constructors

```
public class Point {  
    int x, y;  
  
    public Point() { // no-arg constructor  
        x = 0; y = 0;  
    }  
  
    public Point(int x0, int y0) {  
        x = x0; y = y0;  
    }  
}
```

May be omitted, since primitive fields are initialized to zero. Reference fields are initialized to null

A constructor is a method that has the same name as the class and no return type.

Constructors may be **overloaded**:

A class may have several constructors, if only they have a different number of parameters, or their types differ.

# Creation of **Point** objects

```
Point p1 = new Point();
```

```
Point p2 = new Point(13, 17);
```

If no constructor is provided for a class, a default no-arg constructor with an empty body is provided implicitly.

Note that once a constructor is written, a default no-arg constructor is no longer provided.

```

1 // Minimal Date class that illustrates some Java features
2 // No error checks or javadoc comments
3
4 public class Date
5 {
6     // Zero-parameter constructor
7     public Date( )
8     {
9         month = 1;
10        day = 1;
11        year = 2010;
12    }
13
14    // Three-parameter constructor
15    public Date( int theMonth, int theDay, int theYear )
16    {
17        month = theMonth;
18        day = theDay;
19        year = theYear;
20    }
21
22    // Return true if two equal values
23    public boolean equals( Object rhs )
24    {
25        if( ! ( rhs instanceof Date ) )
26            return false;
27        Date rhDate = ( Date ) rhs;
28        return rhDate.month == month && rhDate.day == day &&
29            rhDate.year == year;
30    }
31
32    // Conversion to String
33    public String toString( )
34    {
35        return month + "/" + day + "/" + year;
36    }
37
38    // Fields
39    private int month;
40    private int day;
41    private int year;
42 }

```

**figure 3.6**

A minimal Date class that illustrates constructors and the equals and toString methods

← explained later



# The `this` reference

The keyword `this` may be used inside an instance method or constructor to denote the receiving instance of the call.

Example of use:

```
public class Point {  
    int x, y;  
  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

Allows access to fields “shadowed” (hidden) by parameters.

# Passing **this** as a parameter

```
class Line {
    Point p1, p2;

    public Line(Point p1, Point p2) {
        this.p1 = p1; this.p2 = p2;
    }
}

public class Point {
    int x, y;

    public Line connect(Point otherPoint) {
        return new Line(this, otherPoint);
    }
}
```



## Dealing with aliasing

```
public class Point {  
    public Line connect(Point otherPoint) {  
        if (this == otherPoint)  
            return null;  
        return new Line(this, otherPoint);  
    }  
}
```

# The `this` shorthand for constructors

Many classes have multiple constructors that behave similarly.

We can use `this` inside a constructor to call one of the other constructors.

Example:

```
public Date() {  
    this(1, 1, 2010);  
}
```

The call to `this` must be the first statement in the constructor.

# The `instanceof` operator

The `instanceof` operator performs a runtime test of class membership.

The result of

```
exp instanceof ClassName
```

is `true` if `exp` is an instance of `ClassName`, and `false` otherwise.

If `exp` is `null`, the result is always `false`.





## Static fields and methods

A **static field** is used when we have a variable that all the instances of the same class need to *share*. Typically, this is a symbolic constant, but need not be.

A **static method** is a method that does not need a controlling object, and thus is typically called by supplying the class name in stead of the controlling object.

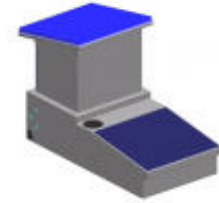
Static fields and methods are called **class fields** and **class methods**, respectively.

Non-static fields and methods are called **instance fields** and **instance methods**, respectively.

```
1 class Ticket
2 {
3     public Ticket( )
4     {
5         System.out.println( "Calling constructor" );
6         serialNumber = ++ticketCount;
7     }
8
9     public int getSerial( )
10    {
11        return serialNumber;
12    }
13
14    public String toString( )
15    {
16        return "Ticket #" + getSerial( );
17    }
18
19    public static int getTicketCount( )
20    {
21        return ticketCount;
22    }
23
24    private int serialNumber;
25    private static int ticketCount = 0;
26 }
27
```

**figure 3.9**

The Ticket class: an example of static fields and methods



# Static initializers

Static fields are initialized when the class is loaded. Occasionally, we need a complex initialization.

Such an initialization may be performed in a block preceded by the keyword `static`. The block must follow the declaration of the static field.

**figure 3.10**

An example of a static initializer

```
1 public class Squares
2 {
3     private static double [ ] squareRoots = new double[ 100 ];
4
5     static
6     {
7         for( int i = 0; i < squareRoots.length; i++ )
8             squareRoots[ i ] = Math.sqrt( ( double ) i );
9     }
10    // Rest of class
11 }
```



# Packages

**Packages** are used to organize similar classes. A package is a collection of related classes, interfaces, or other packages.

Package declaration is file based; that is, all classes in the same source file belong to the same package. The name of the package may be given in the beginning of the source file:

```
package PackageName ;
```

# An example package



The source file Point.java:

```
package geometry;

public class Point {
    int x, y;
    // ...
}
```

The source file Line.java:

```
package geometry;

public class Line {
    Point p1, p2;
}
```

# Examples of application of the package

(1) Using the fully qualified name:

```
import geometry;  
geometry.Point p = new geometry.Point(3, 4);
```

(2) Importing the class and using the simple class name:

```
import geometry.Point;  
Point p = new Point(3, 4);
```

(3) Importing all classes:

```
import geometry.*;  
Point p1 = new Point(3, 4),  
      p2 = new Point(6, 9);  
Line l = new Line(p1, p2);
```

# Packages and directory structure



Use of a package requires a directory structure that corresponds to the name of the package.

Example:

The package

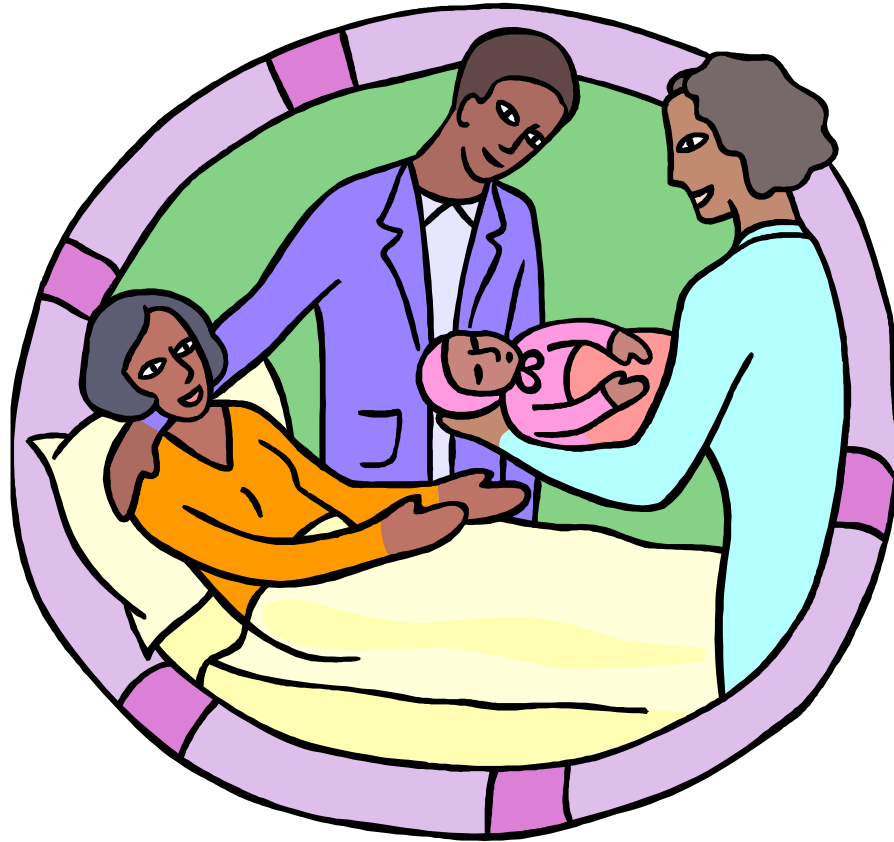
```
dk.ruc.jDisco
```

must be placed in the directory

```
dk/ruc/jDisco
```



# Inheritance





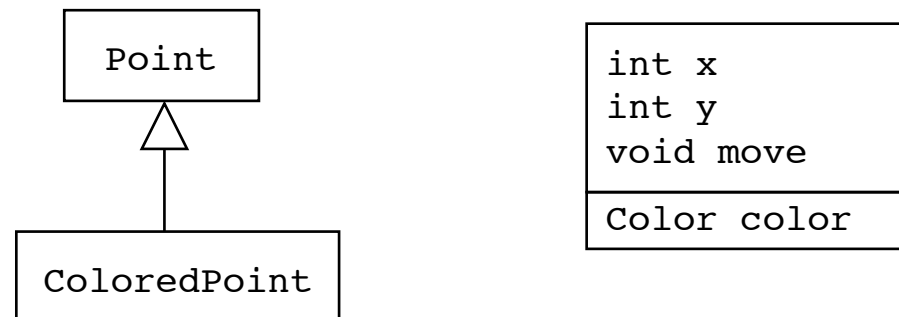
# Inheritance

*Inheritance* is the fundamental object-oriented principle that is used to reuse code among related classes.

Inheritance models the **IS-A relation**.

Example:

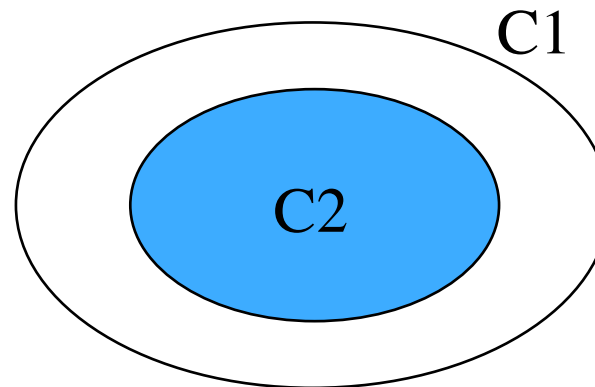
```
class ColoredPoint extends Point {  
    Color color;  
}
```





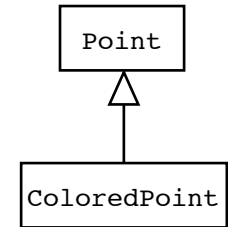
# Inheritance terminology

A class  $C2$  is said to *inherit* from another class  $C1$ , if all instances of  $C2$  are also instances of  $C1$ .



$C2$  is said to be a *subclass* of  $C1$ .  
 $C1$  is said to be a *superclass* of  $C2$ .

# Interpretations of inheritance



A subclass is an *extension* of its superclass.

A subclass is a *specialization* of its superclass.

A superclass is a *generalization* of its subclasses.

```
1 class Person
2 {
3     public Person( String n, int ag, String ad, String p )
4         { name = n; age = ag; address = ad; phone = p; }
5
6     public String toString( )
7         { return getName( ) + " " + getAge( ) + " "
8           + getPhoneNumber( ); }
9
10    public String getName( )
11        { return name; }
12
13    public int getAge( )
14        { return age; }
15
16    public String getAddress( )
17        { return address; }
18
19    public String getPhoneNumber( )
20        { return phone; }
21
22    public void setAddress( String newAddress )
23        { address = newAddress; }
24
25    public void setPhoneNumber( String newPhone )
26        { phone = newPhone; }
27
28    private String name;
29    private int    age;
30    private String address;
31    private String phone;
32 }
```

**figure 4.1**

The Person class stores name, age, address, and phone number.

**figure 4.2**

The Student class stores name, age, address, phone number, and gpa via copy-and-paste.

```
1 class Student
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5         { name = n; age = ag; address = ad; phone = p; gpa = g; }
6
7     public String toString( )
8         { return getName( ) + " " + getAge( ) + " "
9           + getPhoneNumber( ) + " " + getGPA( ); }
10
11    public String getName( )
12        { return name; }
13
14    public int getAge( )
15        { return age; }
16
17    public String getAddress( )
18        { return address; }
19
20    public String getPhoneNumber( )
21        { return phone; }
22
23    public void setAddress( String newAddress )
24        { address = newAddress; }
25
26    public void setPhoneNumber( String newPhone )
27        { phone = newPhone; }
28
29    public double getGPA( )
30        { return gpa; }
31
32    private String name;
33    private int age;
34    private String address;
35    private String phone;
36    private double gpa // grade point average
37 }
```

```
1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5     {
6         /* OOPS! Need some syntax; see Section 4.1.6 */
7         gpa = g; }
8
9     public String toString( )
10    { return getName( ) + " " + getAge( ) + " "
11        + getPhoneNumber( ) + " " + getGPA( ); }
12
13    public double getGPA( )
14    { return gpa; }
15
16    private double gpa;
17 }
```

**figure 4.3**

Inheritance used to  
create Student class

Person Class

name	age
address	phone

Student Class

name	age
address	phone
gpa	

**figure 4.4**

Memory layout with inheritance. Light shading indicates fields that are private, and accessible only by methods of the class. Dark shading in the **Student** class indicates fields that are not accessible in the **Student** class but are nonetheless present.

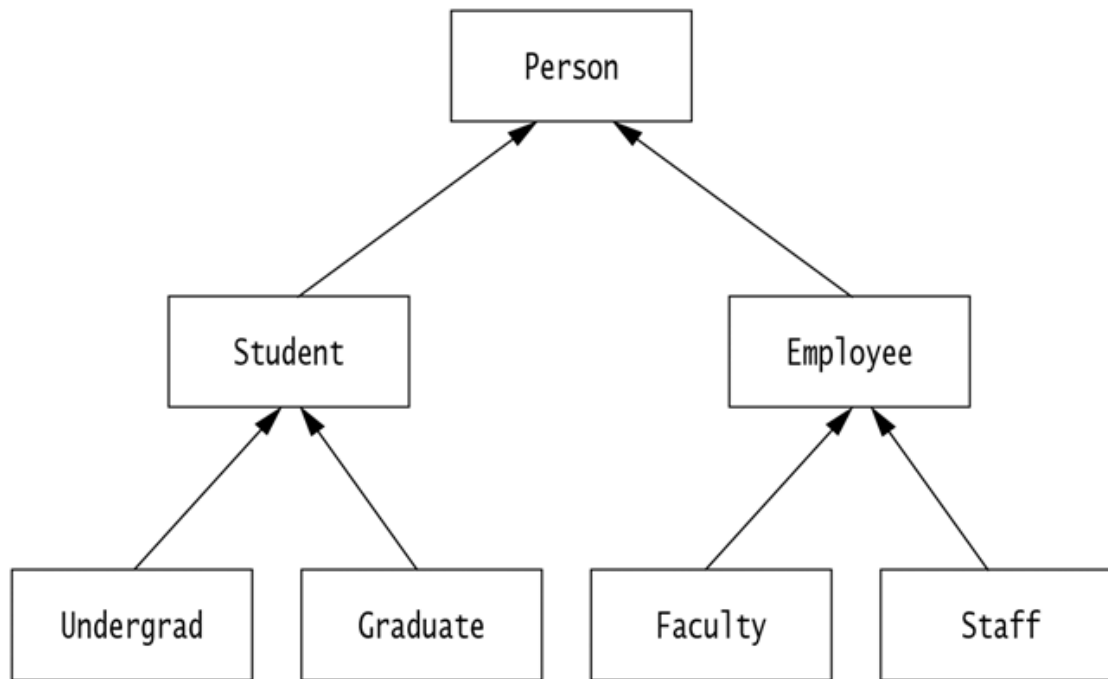


**figure 4.5**

The general layout  
of inheritance

```
1 public class Derived extends Base
2 {
3     // Any members that are not listed are inherited unchanged
4     // except for constructor.
5
6     // public members
7     // Constructor(s) if default is not acceptable
8     // Base methods whose definitions are to change in Derived
9     // Additional public methods
10
11     // private members
12     // Additional data fields (generally private)
13     // Additional private methods
14 }
```

# Inheritance hierarchy



**figure 4.6**  
The Person hierarchy

# Constructors for subclasses

```
import java.awt.Color;

public class ColoredPoint extends Point {
    public Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);           // must be the first statement
        this.color = color;
    }

    public ColoredPoint(int x, int y) {    // black point
        this(x, y, Color.black); // must be the first statement
    }

    public ColoredPoint() {
        color = Color.black;    // invokes super() implicitly
    }
}
```

```
1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5         { super( n, ag, ad, p ); gpa = g; }
6
7     // toString and getAge omitted
8
9     private double gpa;
10 }
```

**figure 4.7**

A constructor for new class Student; uses super

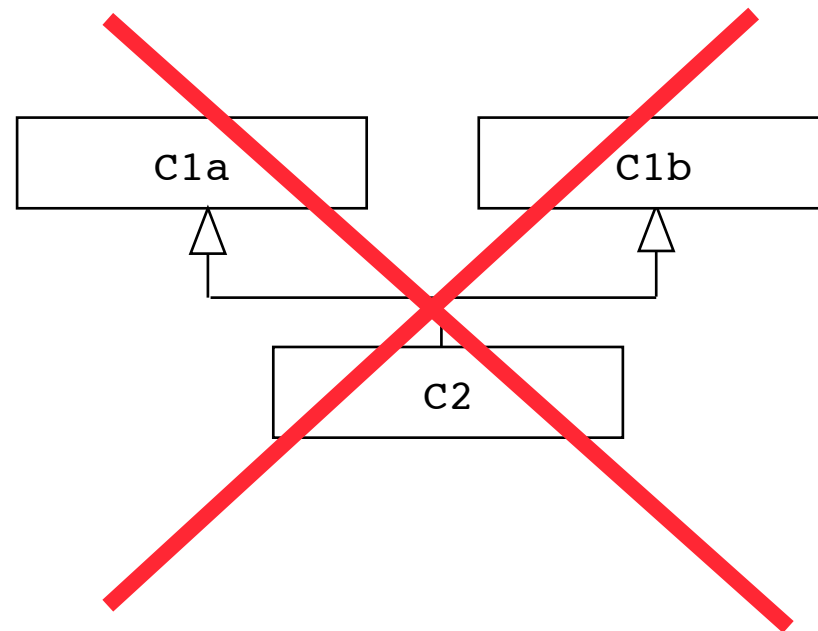
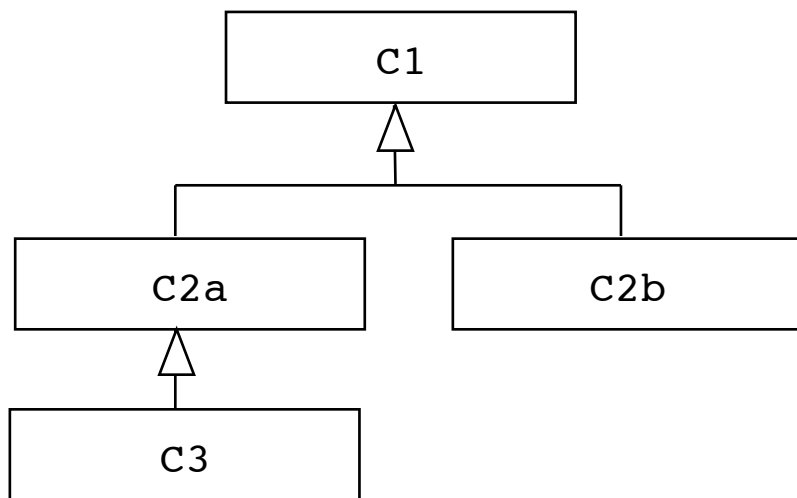
**figure 4.8**

The complete Student and Employee classes, using both forms of super

```
1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5         { super( n, ag, ad, p ); gpa = g; } ←
6
7     public String toString( )
8         { return super.toString( ) + getGPA( ); } ←
9
10    public double getGPA( )
11        { return gpa; }
12
13    private double gpa;
14 }
15
16 class Employee extends Person
17 {
18     public Employee( String n, int ag, String ad,
19                     String p, double s )
20         { super( n, ag, ad, p ); salary = s; } ←
21
22     public String toString( )
23         { return super.toString( ) + " $" + getSalary( ); } ←
24
25     public double getSalary( )
26         { return salary; }
27
28     public void raise( double percentRaise )
29         { salary *= ( 1 + percentRaise ); }
30
31     private double salary;
32 }
```

# Single inheritance

In Java, a class may inherit from at most one superclass.



# Class Object



In Java, all classes are organized in a hierarchy (tree) that has the class `Object` as root.

All classes, except `Object`, has a unique superclass.

If no superclass is specified for a class, then `Object` is its superclass.

# Class Object

```
public class Object {  
    public String toString();  
    public boolean equals(Object obj);  
    public int hashCode();  
    protected Object clone();  
  
    ...  
}
```



# Polymorphic assignment



**Rule of assignment:** The type of the expression at the right-hand side of an assignment must be a subtype of the type at the left-hand side of the assignment.

```
Person p = new Person(...);
Student s = new Student(...);
Employee e = new Employee(...);

p = s;           // ok

p = p;          // ok

s = e;          // compilation error

e = p;          // compilation error
```

*Polymorphic* (from Greek): having many forms

# Type conversion (casting)

The rule of assignment is checked at **compile time**.

```
e = p;    // compilation error
```

The rule may be satisfied by narrowing the type at the right-hand side of the assignment:

```
e = (Employee) p;    // explicit cast, ok
```

The validity of an explicit cast is always checked at **run time**.

If the cast is invalid, a `ClassCastException` is thrown.

# Javadoc for `toString` in class `Object`

## `toString`

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

### Returns:

a string representation of the object.

# Overriding methods



Methods in a superclass may be *overridden* by methods defined in a subclass. *Overriding* refers to the introduction of an instance method in a subclass that has the same name, same signature, and a type-compatible return type of a method in the superclass. Implementation of the methods in the subclass *replaces* the implementation in the superclass.

```
class Shape {
    public String toString() {
        return "Shape";
    }
}

class Line extends Shape {
    Point p1, p2;

    public String toString() {
        return "Line from " + p1 + " to " + p2;
    }
}
```

# @Override

The annotation `@Override` forces the compiler to check that a method is overridden.

```
class Line extends Shape {  
    @Override  
    public String toString() {  
        ...  
    }  
}
```

Misspelling



The compiler prints out the error message

```
method does not override or implement a method from its supertype
```

# @Override

```
class Date {  
    @Override  
    public boolean equals(Date rhs) {  
        ...  
    }  
}
```

Since the parameter type must be `Object`, the compiler prints out the error message

```
method does not override or implement a method from its supertype
```

# @Override

```
class Date {  
    @Override  
    boolean equals(Object rhs) {  
        ...  
    }  
}
```

The compiler prints out the error message

```
equals(java.lang.Object) in Date cannot override  
equals(java.lang.Object) in java.lang.Object  
attempting to assign weaker access privileges; was public
```

# Polymorphic method invocation

Which implementation of a overridden method will be invoked depends on the actual class of the object referenced by the variable at **run time**, not the declared type of the variable. This is known as *dynamic binding*.

Example:

```
Person p;  
p = Math.random() > 0.5 ? new Student(...)  
                        : new Employee(...);  
System.out.println(p.toString());
```



```

1 class PersonDemo
2 {
3     public static void printAll( Person [ ] arr )
4     {
5         for( int i = 0; i < arr.length; i++ )
6         {
7             if( arr[ i ] != null )
8             {
9                 System.out.print( "[" + i + " ] " );
10                System.out.println( arr[ i ].toString( ) );
11            }
12        }
13    }
14
15    public static void main( String [ ] args )
16    {
17        Person [ ] p = new Person[ 4 ];
18
19        p[0] = new Person( "joe", 25, "New York",
20                        "212-555-1212" );
21        p[1] = new Student( "jill", 27, "Chicago",
22                          "312-555-1212", 4.0 );
23        p[3] = new Employee( "bob", 29, "Boston",
24                            "617-555-1212", 100000.0 );
25
26        printAll( p );
27    }
28 }

```

**figure 4.9**

An illustration of polymorphism with arrays

# Field and method modifiers



## **public**

The feature is accessible to any class.

## **private**

The feature is only accessible by the class itself.

## **protected**

The feature is accessible by the class itself, all its subclasses, and all the classes within the same package.

## Neither **public**, **private**, nor **protected**

The feature is accessible by all the classes in the same package.

## **final**

A **final** field has a constant value, which may not be changed.

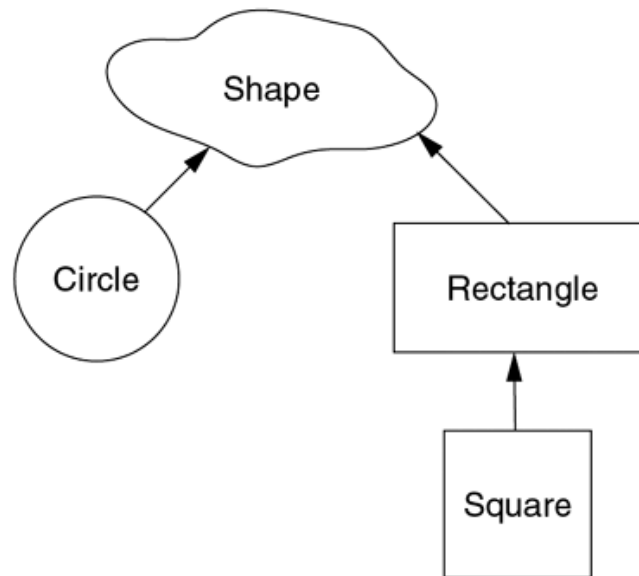
A **final** method may not be overridden in subclasses.

## **static**

A **static** field is shared by all instances of the class.

A **static** method accesses only static fields.

# Design of hierarchies



**figure 4.10**

The hierarchy of shapes used in an inheritance example

**figure 4.11**

A possible Shape class

```
1 public class Shape
2 {
3     public double area( )
4     {
5         return -1;
6     }
7 }
```

```

1 public class Circle extends Shape
2 {
3     public Circle( double rad )
4         { radius = rad; }
5
6     public double area( )
7         { return Math.PI * radius * radius; }
8
9     public double perimeter( )
10        { return 2 * Math.PI * radius; }
11
12    public String toString( )
13        { return "Circle: " + radius; }
14
15    private double radius;
16 }
17
18 public class Rectangle extends Shape
19 {
20     public Rectangle( double len, double wid )
21         { length = len; width = wid; }
22
23     public double area( )
24         { return length * width; }
25
26     public double perimeter( )
27         { return 2 * ( length + width ); }
28
29     public String toString( )
30         { return "Rectangle: " + length + " " + width; }
31
32     public double getLength( )
33         { return length; }
34
35     public double getWidth( )
36         { return width; }
37
38     private double length;
39     private double width;
40 }
41
42 public class Square extends Rectangle
43 {
44     public Square( double side )
45         { super( side, side ); }
46
47     public String toString( )
48         { return "Square: " + getLength( ); }
49 }

```

**figure 4.12**

Circle, Rectangle, and  
Square classes

**figure 4.13**

A sample program  
that uses the shape  
hierarchy

```
1 class ShapeDemo
2 {
3     public static double totalArea( Shape [ ] arr )
4     {
5         double total = 0;
6
7         for( Shape s : arr )
8             if( s != null )
9                 total += s.area( );
10
11         return total;
12     }
13
14     public static void printAll( Shape [ ] arr )
15     {
16         for( Shape s : arr )
17             System.out.println( s );
18     }
19
20     public static void main( String [ ] args )
21     {
22         Shape [ ] a = { new Circle( 2.0 ),
23                         new Rectangle( 1.0, 3.0 ),
24                         null, new Square( 2.0 ) };
25
26         System.out.println( "Total area = " + totalArea( a ) );
27         printAll( a );
28     }
29 }
```

# Abstract methods and classes



An **abstract method** is a method that declares functionality that all derived class objects must eventually implement. In other words, it says what these objects *can do*. However, it does not provide a default implementation.

A class that has a least one abstract method is an **abstract** class. Java requires that abstract classes explicitly be declared as such (using the keyword `abstract`).

Any attempt to construct an instance of an abstract class is illegal.

```
1 public abstract class Shape
2 {
3     public abstract double area( );
4     public abstract double perimeter( );
5
6     public double semiperimeter( )
7         { return perimeter( ) / 2; }
8 }
```

**figure 4.14**

An abstract Shape class. Figures 4.12 and 4.13 are unchanged.



# Interfaces



An **interface** can be thought of as a special form of class, which declares only the *features to be supported* by the class.

Java interfaces provide no implementation. Implementation is deferred to classes that implement the interfaces.

Interface features can be either abstract methods or constants (that is, static and final fields). All features are public.

Like abstract classes, an interface may not have instances.

# Implementation of interfaces

A class that *implements* an interface provides implementation for the abstract methods declared in interface by overriding those methods.

```
interface Drawable {
    void draw(Graphics g);
}

public class Line implements Drawable {
    Point p1, p2;

    public void draw(Graphics g) {
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}
```

# Extending and implementing interfaces

Classes that implement an interface must provide implementations of all methods of the interface (unless the class itself should be abstract).

An interface may inherit from one or more interfaces but not from a class.

Java allows only single inheritance for class extension but *multiple inheritance* for interface extension and implementation.

# Multiple interface implementations

```
interface Movable {
    void move(int dx, int dy);
}

public class Line implements Drawable, Movable {
    Point p1, p2;

    public void move(int dx, int dy) {
        p1.move(dx, dy);
        p2.move(dx, dy);
    }

    public void draw(Graphics g) {
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}
```

**figure 4.15**

The Comparable  
interface  
(pre-Java 5)

```
1 package java.lang;
2
3 public interface Comparable
4 {
5     int compareTo( Object other );
6 }
```

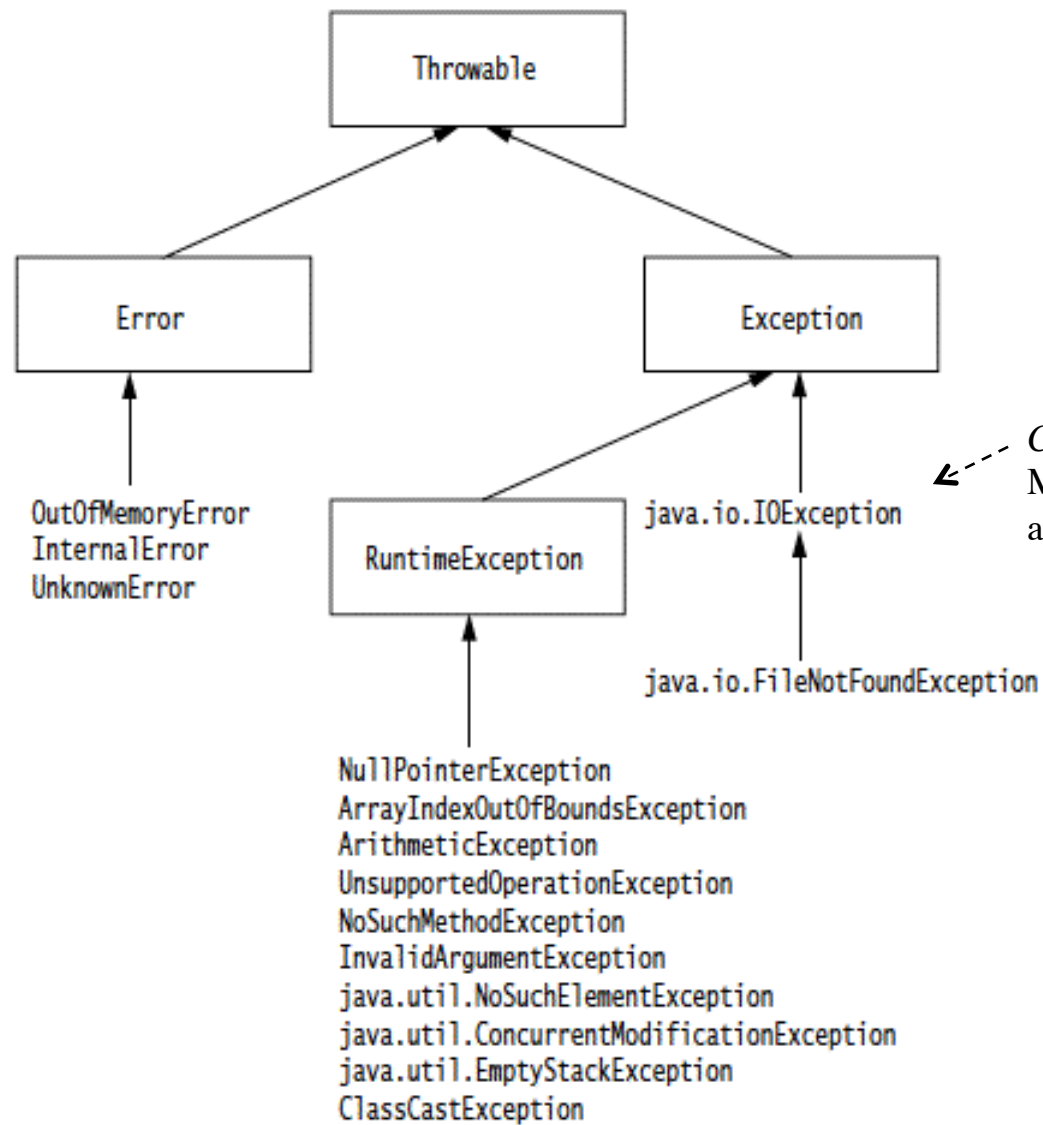
Contract of the method `compareTo`:

Result < 0,	if this precedes other;
Result = 0,	if neither this precedes other, nor other precedes this;
Result > 0,	if other precedes this.

```
1 public abstract class Shape implements Comparable
2 {
3     public abstract double area( );
4     public abstract double perimeter( );
5
6     public int compareTo( Object rhs )
7     {
8         Shape other = (Shape) rhs;
9         double diff = area( ) - other.area( );
10
11         if( diff == 0 )
12             return 0;
13         else if( diff < 0 )
14             return -1;
15         else
16             return 1;
17     }
18
19     public double semiperimeter( )
20     { return perimeter( ) / 2; }
21 }
```

**figure 4.16**

The Shape class (final version), which implements the Comparable interface



**figure 4.18**  
The hierarchy of exceptions (partial list)

← *Checked exception:*  
Must be caught or declared in a throws clause

**figure 4.19**

NoSuchElementException,  
implemented in  
weiss.util

```
1 package weiss.util;
2
3 public class NoSuchElementException extends RuntimeException
4 {
5     /**
6      * Constructs a NoSuchElementException with
7      * no detail message.
8      */
9     public NoSuchElementException( )
10    {
11    }
12
13    /**
14     * Constructs a NoSuchElementException with
15     * a detail message.
16     * @param msg the detail message.
17     */
18    public NoSuchElementException( String msg )
19    {
20        super( msg );
21    }
22 }
```



# Generics



# Using Object for genericity

```
1 // MemoryCell class
2 // Object read( )      --> Returns the stored value
3 // void write( Object x ) --> x is stored
4
5 public class MemoryCell
6 {
7     // Public methods
8     public Object read( )      { return storedValue; }
9     public void write( Object x ) { storedValue = x; }
10
11     // Private internal data representation
12     private Object storedValue;
13 }
```

**figure 4.22**

A generic MemoryCell class (pre-Java 5)

Generic: broadly covering

```
1 public class TestMemoryCell
2 {
3     public static void main( String [ ] args )
4     {
5         MemoryCell m = new MemoryCell( );
6
7         m.write( "37" );
8         String val = (String) m.read( );
9         System.out.println( "Contents are: " + val );
10    }
11 }
```

**figure 4.23**

Using the generic  
MemoryCell class  
(pre-Java 5)

**figure 4.24**

A simplified `ArrayList`,  
with `add`, `get`, and `size`  
(pre-Java 5)

```
1  /**
2  * The SimpleArrayList implements a growable array of Object.
3  * Insertions are always done at the end.
4  */
5  public class SimpleArrayList
6  {
7      /**
8       * Returns the number of items in this collection.
9       * @return the number of items in this collection.
10     */
11     public int size( )
12     {
13         return theSize;
14     }
15
16     /**
17     * Returns the item at position idx.
18     * @param idx the index to search in.
19     * @throws ArrayIndexOutOfBoundsException if index is bad.
20     */
21     public Object get( int idx )
22     {
23         if( idx < 0 || idx >= size( ) )
24             throw new ArrayIndexOutOfBoundsException( );
25         return theItems[ idx ];
26     }
27
28     /**
29     * Adds an item at the end of this collection.
30     * @param x any object.
31     * @return true (as per java.util.ArrayList).
32     */
33     public boolean add( Object x )
34     {
35         if( theItems.length == size( ) )
36         {
37             Object [ ] old = theItems;
38             theItems = new Object[ theItems.length * 2 + 1 ];
39             for( int i = 0; i < size( ); i++ )
40                 theItems[ i ] = old[ i ];
41         }
42
43         theItems[ theSize++ ] = x;
44         return true;
45     }
46
47     private static final int INIT_CAPACITY = 10;
48
49     private int         theSize = 0;
50     private Object [ ] theItems = new Object[ INIT_CAPACITY ];
51 }
```



# Wrapper classes

Because in Java primitive types are not objects, *wrapper* classes are provided to “wrap” the values of primitive types into objects when needed. Each primitive type has a corresponding wrapper class.

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	<b>Character</b>
double	Double
float	Float
int	<b>Integer</b>
long	Long
short	Short

Each wrapper object is *immutable* (meaning its state can never change), and stores one primitive value that is set when the object is constructed.



# Boxing conversions

Automatic conversion of a value of a primitive type to an instance of a wrapper class is called **auto-boxing**.

```
Integer val = 7;  
    is equivalent to  
Integer val = new Integer(7);
```

Automatic conversion of an instance of a wrapper class to a value of a primitive type is called **auto-unboxing**.

```
int x = val;  
    is equivalent to  
int x = val.intValue();
```

```
1 import java.util.ArrayList;
2
3 public class BoxingDemo
4 {
5     public static void main( String [ ] args )
6     {
7         ArrayList<Integer> arr = new ArrayList<Integer>( );
8
9         arr.add( new Integer( 46 ) );
10        Integer wrapperVal = arr.get( 0 );
11        int val = wrapperVal.intValue( );
12        System.out.println( "Position 0: " + val );
13    }
14 }
```

**figure 4.25**

An illustration of the Integer wrapper class using Java 5 generic ArrayList

Without automatic conversion

**figure 4.26**

Autoboxing and  
unboxing

```
1 import java.util.ArrayList;
2
3 public class BoxingDemo
4 {
5     public static void main( String [ ] args )
6     {
7         ArrayList<Integer> arr = new ArrayList<Integer>( );
8
9         arr.add( 46 );
10        int val = arr.get( 0 );
11        System.out.println( "Position 0: " + val );
12    }
13 }
```



```

1 import java.math.BigInteger;
2
3 class FindMaxDemo
4 {
5     /**
6      * Return max item in a.
7      * Precondition: a.length > 0
8      */
9     public static Comparable findMax( Comparable [ ] a )
10    {
11        int maxIndex = 0;
12
13        for( int i = 1; i < a.length; i++ )
14            if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
15                maxIndex = i;
16
17        return a[ maxIndex ];
18    }
19
20    /**
21     * Test findMax on BigInteger and String objects.
22     */
23    public static void main( String [ ] args )
24    {
25        BigInteger [ ] bi1 = { new BigInteger( "8764" ),
26                               new BigInteger( "29345" ),
27                               new BigInteger( "1818" ) };
28
29        String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };
30
31        System.out.println( findMax( bi1 ) );
32        System.out.println( findMax( st1 ) );
33    }
34 }

```

**figure 4.29**

A generic findMax routine, with demo using shapes and strings (pre-Java 5)

# Generics in Java 5



Generics allow a type or method to operate on objects of various types while providing compile-time type safety.

In Java 5, classes, interfaces and methods may be declared with one or more *type parameters*.

Benefits:

- Compile-time type safety
- Greater readability (unnecessary casting is avoided)

# A generic ArrayList

```
public class ArrayList<E> implements List {  
    public boolean add(E o);  
    public E get(int i);  
    public boolean addAll(Collection<? extends E> c);  
    ...  
}
```

The name of the type parameter, here `E`, may be chosen freely.

Alternative: `ElementType`.

The notation `<? extends E>` denotes an arbitrary subtype of `E` (or `E` itself). The wildcard character `?` stands for an unknown type.



# Example of application

```
ArrayList<String> names = new ArrayList<String>();

names.add("Barack Obama");    // OK
names.add(new Integer(21));  // Compile-time error

String name = names.get(0);   // OK without cast

for (String name : names)
    System.out.println(name.toUpperCase());
```

**figure 4.30**

Generic  
implementation of the  
MemoryCell class

```
1 public class GenericMemoryCell<AnyType>
2 {
3     public AnyType read( )
4         { return storedValue; }
5     public void write( AnyType x )
6         { storedValue = x; }
7
8     private AnyType storedValue;
9 }
```

# Generic Comparable interface

```
1 package java.lang;
2
3 public interface Comparable<AnyType>
4 {
5     public int compareTo( AnyType other );
6 }
```

**figure 4.31**

Comparable interface,  
Java 5 version which  
is generic

# Generic collections are not covariant

```
1 public static double totalArea( ArrayList<Shape> arr )
2 {
3     double total = 0;
4
5     for( Shape s : arr )
6         if( s != null )
7             total += s.area( );
8
9     return total;
10 }
```

**figure 4.32**

totalArea method that  
does not work if  
passed an  
ArrayList<Square>

*Covariance:* Type compatibility

ArrayList<Square> is not type compatible with ArrayList<Shape>.  
(In contrast, Square[ ] is type compatible with Shape[ ]. Arrays are covariant.)

# Use of wildcards with bound

**figure 4.33**

totalArea method  
revised with wildcards  
that works if passed  
an ArrayList<Square>

```
1 public static double totalArea( ArrayList<? extends Shape> arr )
2 {
3     double total = 0;
4
5     for( Shape s : arr )
6         if( s != null )
7             total += s.area( );
8
9     return total;
10 }
```



# Generic static method

```
1 public static <AnyType>
2 boolean contains( AnyType [ ] arr, AnyType x )
3 {
4     for( AnyType val : arr )
5         if( x.equals( val ) )
6             return true;
7
8     return false;
9 }
```

**figure 4.34**

Generic static method  
to search an array

The type parameters in a generic method precede the return type.

```
1 public static <AnyType> AnyType findMax( AnyType [ ] a )
2 {
3     int maxIndex = 0;
4
5     for( int i = 1; i < a.length; i++ )
6         if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
7             maxIndex = i;
8
9     return a[ maxIndex ];
10 }
```

**figure 4.35**

Generic static method  
to find largest element  
in an array that does  
not work

The compiler cannot prove that the call to `compareTo` at line 6 is valid.

# Using type bounds

**figure 4.36**

Generic static method to find largest element in an array. Illustrates a bounds on the type parameter

```
1 public static <AnyType extends Comparable<? super AnyType>>
2 AnyType findMax( AnyType [ ] a )
3 {
4     int maxIndex = 0;
5
6     for( int i = 1; i < a.length; i++ )
7         if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
8             maxIndex = i;
9
10    return a[ maxIndex ];
11 }
```

The notation `<? extends AnyType>` denotes an arbitrary supertype of `AnyType` (or `AnyType` itself).

**figure 4.37**  
SimpleArrayList class  
using generics

```
1 /**
2  * The GenericSimpleArrayList implements a growable array.
3  * Insertions are always done at the end.
4  */
5 public class GenericSimpleArrayList<AnyType>
6 {
7     /**
8     * Returns the number of items in this collection.
9     * @return the number of items in this collection.
10    */
11    public int size( )
12    {
13        return theSize;
14    }
15
16    /**
17    * Returns the item at position idx.
18    * @param idx the index to search in.
19    * @throws ArrayIndexOutOfBoundsException if index is bad.
20    */
21    public AnyType get( int idx )
22    {
23        if( idx < 0 || idx >= size( ) )
24            throw new ArrayIndexOutOfBoundsException( );
25        return theItems[ idx ];
26    }
27
28    /**
29    * Adds an item at the end of this collection.
30    * @param x any object.
31    * @return true.
32    */
33    public boolean add( AnyType x )
34    {
35        if( theItems.length == size( ) )
36        {
37            AnyType [ ] old = theItems;
38            theItems = (AnyType [ ])new Object[ size( ) * 2 + 1 ];
39            for( int i = 0; i < size( ); i++ )
40                theItems[ i ] = old[ i ];
41        }
42
43        theItems[ theSize++ ] = x;
44        return true;
45    }
46
47    private static final int INIT_CAPACITY = 10;
48
49    private int theSize;
50    private AnyType [ ] theItems;
51 }
```

Because we cannot generate arrays of generic objects, we must create an array of `Object` and use a typecast.

**figure 4.38**

The `SimpleRectangle` class, which does not implement the `Comparable` interface

```
1 // A simple rectangle class.
2 public class SimpleRectangle
3 {
4     public SimpleRectangle( int len, int wid )
5         { length = len; width = wid; }
6
7     public int getLength( )
8         { return length; }
9
10    public int getWidth( )
11        { return width; }
12
13    public String toString( )
14        { return "Rectangle " + getLength( ) + " by "
15          + getWidth( ); }
16
17    private int length;
18    private int width;
19 }
```

# The Comparator interface

```
1 package weiss.util;
2
3 /**
4  * Comparator function object interface.
5  */
6 public interface Comparator<AnyType>
7 {
8     /**
9     * Return the result of comparing lhs and rhs.
10    * @param lhs first object.
11    * @param rhs second object.
12    * @return < 0 if lhs is less than rhs,
13    *         0 if lhs is equal to rhs,
14    *         > 0 if lhs is greater than rhs.
15    */
16    int compare( AnyType lhs, AnyType rhs );
17 }
```

**figure 4.39**

The Comparator interface, originally defined in java.util and rewritten for the weiss.util package

**figure 4.40**

The generic findMax algorithm, using a function object

```
1 public class Utils
2 {
3     // Generic findMax with a function object.
4     // Precondition: a.length > 0.
5     public static <AnyType> AnyType
6     findMax( AnyType [ ] a, Comparator<? super AnyType> cmp )
7     {
8         int maxIndex = 0;
9
10        for( int i = 1; i < a.length; i++ )
11            if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
12                maxIndex = i;
13
14        return a[ maxIndex ];
15    }
16 }
```

**figure 4.41**

An example of a  
function object

```
1 class OrderRectByWidth implements Comparator<SimpleRectangle>
2 {
3     public int compare( SimpleRectangle r1, SimpleRectangle r2 )
4         { return( r1.getWidth() - r2.getWidth() ); }
5 }
6
7 public class CompareTest
8 {
9     public static void main( String [ ] args )
10    {
11        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
12        rects[ 0 ] = new SimpleRectangle( 1, 10 );
13        rects[ 1 ] = new SimpleRectangle( 20, 1 );
14        rects[ 2 ] = new SimpleRectangle( 4, 6 );
15        rects[ 3 ] = new SimpleRectangle( 5, 5 );
16
17        System.out.println( "MAX WIDTH: " +
18            Utils.findMax( rects, new OrderRectByWidth( ) ) );
19    }
20 }
```



# Static nested class

```
1 import java.util.Comparator;
2
3 class CompareTestInner1
4 {
5     private static class OrderRectByWidth implements Comparator<SimpleRectangle>
6     {
7         public int compare( SimpleRectangle r1, SimpleRectangle r2 )
8         { return r1.getWidth( ) - r2.getWidth( ); }
9     }
10
11     public static void main( String [ ] args )
12     {
13         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
14         rects[ 0 ] = new SimpleRectangle( 1, 10 );
15         rects[ 1 ] = new SimpleRectangle( 20, 1 );
16         rects[ 2 ] = new SimpleRectangle( 4, 6 );
17         rects[ 3 ] = new SimpleRectangle( 5, 5 );
18
19         System.out.println( "MAX WIDTH: " +
20             Utils.findMax( rects, new OrderRectByWidth( ) ) );
21     }
22 }
```

**figure 4.42**

Using a nested class to hide the OrderRectByWidth class declaration

# Local inner class

```
1 class CompareTestInner2
2 {
3     public static void main( String [ ] args )
4     {
5         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11         class OrderRectByWidth implements Comparator<SimpleRectangle>
12         {
13             public int compare( SimpleRectangle r1, SimpleRectangle r2 )
14             { return r1.getWidth( ) - r2.getWidth( ); }
15         }
16
17         System.out.println( "MAX WIDTH: " +
18             Utils.findMax( rects, new OrderRectByWidth( ) ) );
19     }
20 }
```

**figure 4.43**

Using a local class to hide the OrderRectByWidth class declaration further

Non-static nested classes are called *inner classes*.

# Anonymous inner class

```
1 class CompareTestInner3
2 {
3     public static void main( String [ ] args )
4     {
5         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11         System.out.println( "MAX WIDTH: " +
12             Utils.findMax( rects, new Comparator<SimpleRectangle>( )
13                 {
14                     public int compare( SimpleRectangle r1, SimpleRectangle r2 )
15                     { return r1.getWidth( ) - r2.getWidth( ); }
16                 }
17             ) );
18     }
19 }
```

**figure 4.44**

Using an anonymous class to implement the function object

# Lambda expression

```
1 class CompareTestInner3
2 {
3     public static void main( String [ ] args )
4     {
5         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11         System.out.println( "MAX WIDTH: " +
12             Utils.findMax( rects, (r1, r2) -> r1.getWidth() - r2.getWidth() ) );
13
14
15
16
17
18     }
19 }
```