# Course of Programming in Java

by Łukasz Stafiniak

*Email:* lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
*Web:* www.ii.uni.wroc.pl/~lukstafi

# The Java Programming Language

## Chapter 3: Extending Classes

## Chapter 4: Interfaces

## Chapter 5: Nested Classes and Interfaces

by Ken Arnold, James Gosling, David Holmes

# Extending Classes

The ability to extend classes interacts with the access control mechanisms to expand the notion of **contract** that a class presents. Each class can present two different contracts: one for users of the class and one for extenders of the class. Both of these contracts must be carefully designed.

With class extension, **inheritance of contract** and **inheritance of implementation** always occur together. However, you can define new types independent of implementation by using interfaces.

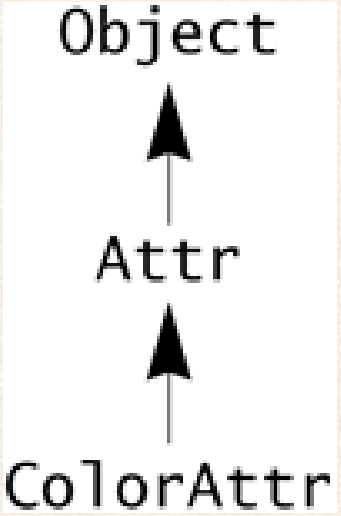
```
public class Attr {

    private final String name;
    private Object value = null;

    public Attr(String name) {
        this.name = name;
    }

    public Attr(String name, Object value) {
        this.name = name;
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public Object getValue() {
        return value;
    }
```

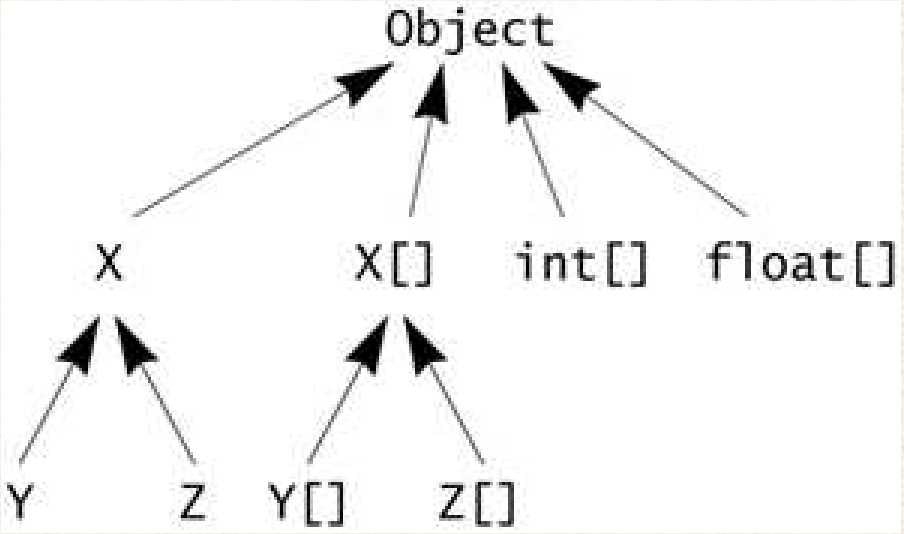
```
    public Object setValue(Object newValue) {
        Object oldVal = value;
        value = newValue;
        return oldVal;
    }

    public String toString() {
        return name + "='" + value + "'";
    }
}
```

```
class ColorAttr extends Attr {
    private ScreenColor myColor; // the decoded color

    public ColorAttr(String name, Object value) {
        super(name, value); // let the superclass ‘‘honor its contract’’
        decodeColor();
    }
    public ColorAttr(String name) {
        this(name, "transparent");
    }
    public ColorAttr(String name, ScreenColor value) {
        super(name, value.toString());
        myColor = value;
    }
    public Object setValue(Object newValue) {
        // do the superclass's setValue work first
        Object retval = super.setValue(newValue);
        decodeColor();
        return retval;
    }
```

```
    /** Set value to ScreenColor, not description */
    public ScreenColor setValue(ScreenColor newValue) {
        // do the superclass's setValue work first
        super.setValue(newValue.toString());
        ScreenColor oldValue = myColor;
        myColor = newValue;
        return oldValue;
    }

    /** Return decoded ScreenColor object */
    public ScreenColor getColor() {
        return myColor;
    }

    /** set ScreenColor from description in getValue */
    protected void decodeColor() {
        if (getValue() == null)
            myColor = null;
        else
            myColor = new ScreenColor(getValue());
    }
}
```

Single-rooted hierarchy:



**Initialization**

When an object is created, memory is allocated and cleared for all its fields, including those inherited from superclasses.

After this, construction has three phases:

1. Invoke a superclass's constructor.

   If not given, the implicit constructor `super()` is used.

2. Initialize the fields using their initializers and any initialization blocks.

3. Execute the body of the constructor.

Any methods can be called.

## Overriding

- **overloading**: add a new method, same name, different signatures
- **overriding**: "replace" a method, same name and signatures

  old method can still be accessed, e.g. `super.setValue(newValue)`
- **hiding** a field (superclass methods keep using the old field)

A method can be overridden only if it is accessible (e.g. not `private`).

Static methods can only be hidden like fields, but anyway they should be accessed via the name of its declaring class (unaffected).

```
class SuperShow {
    public String str = "SuperStr";
    public void show() {
         System.out.println("Super.show: " + str);
    }
}
class ExtendShow extends SuperShow {
    public String str = "ExtendStr";
    public void show() {
         System.out.println("Extend.show: " + str);
    }
    public static void main(String[] args) {
        ExtendShow ext = new ExtendShow();
        SuperShow sup = ext;
        sup.show();
        ext.show();
        System.out.println("sup.str = " + sup.str);
        System.out.println("ext.str = " + ext.str);
    }
}
```

There is only one object, but we have two variables containing references to it: One variable has type `SuperShow` (the superclass) and the other variable has type `ExtendedShow` (the actual class). Here is the output of the example when run:

```
Extend.show: ExtendStr

Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

```
class Base {
    /** return the class name */
    protected String name() {
        return "Base";
    }
}

class More extends Base {
    protected String name() {
        return "More";
    }
    protected void printName() {
       Base sref = (Base) this;

       System.out.println("this.name()  = "+this.name());
       System.out.println("sref.name()  = "+sref.name());
       System.out.println("super.name() = "+super.name());
    }
}
```

Although `sref` and `super` both refer to the same object using the type `Base`, only `super` will ignore the real class of the object. Here is the output of `printName`:

```
this.name()  = More

sref.name()  = More
super.name() = Base
```

## Type Compatibility and Conversion

A subclass instance can be used in place of a superclass parameter or assigned to a superclass variable.

A superclass instance can be explicitly cast to any of its subclasses.

```
if (sref instanceof More)
    mref = (More) sref;
```

If the compiler can tell that a cast is incorrect then a compile time error can occur. If the compiler cannot ascertain that the cast is correct at compile time, then a run time check will be performed. If the run time check fails because the cast is incorrect, then a `ClassCastException` is thrown.

# protected

```
class Cell {
    private Cell next;
    private Object element;
    public Cell(Object element) {
        this.element = element;
    }
    public Cell(Object e, Cell n) {
        element = e;
        this.next = n;
    }
    public Object getElement() {
        return element;
    }
    public void setElement(Object e) {
        this.element = e;
    }
    public Cell getNext() {
        return next;
    }
    public void setNext(Cell next) {
        this.next = next;
    }
}
```

```
public class SingleLinkQueue {

    protected Cell head;
    protected Cell tail;

    public void add(Object i)
    {/* ... */}
    public Object remove() {/* ... */}
}

public class PriorityQueue extends
SingleLinkQueue {
    public void add(Object i) {
        Cell current = head; // OK
        // maintain ordered list...
    }
    public void merge(PriorityQueue q)
{
        Cell first = q.head; // OK
        // put the result in q...
    }
    // ...
}
```

Some other subclass of `SingleLinkQueue` could require a different order of elements than what `PriorityQueue` provides.

```
class Cell {
    private Cell next;
    private Object element;
    public Cell(Object element) {
        this.element = element;
    }
    public Cell(Object e, Cell n) {
        element = e;
        this.next = n;
    }
    public Object getElement() {
        return element;
    }
    public void setElement(Object e) {
        this.element = e;
    }
    public Cell getNext() {
        return next;
    }
    public void setNext(Cell next) {
        this.next = next;
    }
}
```

```
public class SingleLinkQueue {

    protected Cell head;
    protected Cell tail;

    public void add(Object i)
    {/* ... */}
    public Object remove() {/* ... */}
}

public class PriorityQueue extends
SingleLinkQueue {
    public void add(Object i) {
        Cell current = head; // OK
        // maintain ordered list...
    }
    public void
    merge(SingleLinkQueue q) {
        Cell first = q.head; // error!
        // put the result in q...
    }
    // ...
}
```

# Access levels

1. Private: `private Object getElement() { ... }`
2. Package (default): `Object getElement() { ... }`
3. Protected: `protected Object getElement() { ... }`
4. Public: `public Object getElement() { ... }`

Each level has access to the next level:

for example protected fields and methods are visible across the package.

A subclass cannot make a method less visible than it was in superclass.

# `final` methods and classes

- No overriding:

  ```
  final public boolean validatePassword() { ... }
  ```

- No subclasses:

  ```
  final class NoExtending {
      // ...
  }
  ```

- helps security and efficiency – static dispatch (possible inlining)
- making all methods `final`:

  can add new functionality but the meaning of old functionality fixed

# `abstract` **methods and classes**

- `abstract` method: just a contract (a specification), no method body
- if a class has any abstract method, it must itself be abstract

```
abstract class Benchmark {

    abstract void benchmark();

    public final long repeat(int count) {
        long start = System.nanoTime();
        for (int i = 0; i < count; i++)
            benchmark();
        return (System.nanoTime() - start);
    }
}
```

```
class MethodBenchmark extends Benchmark {

    /** Do nothing, just return. */
    void benchmark() {
    }

    public static void main(String[] args) {
        int count = Integer.parseInt(args[0]);
        long time = new MethodBenchmark().repeat(count);
        System.out.println(count + " methods in " +
                           time + " nanoseconds");
    }
}
```

# The `Object` Class

`public boolean equals(Object obj)`

Compares the receiving object and the object referenced by `obj` for value equality, but by default testing physical equality `this == obj`.

`public int hashCode()`

Returns a hash code for `this` object, used when storing objects in hashed collections; by default, usually different for physically different objects.

`protected Object clone() throws CloneNotSupportedException`

Returns a clone (a new copy) of `this` object.

`public final Class<?> getClass()`

Returns the type token that represents the class of this object; when invoked on an instance of `Attr` it will return an instance of `Class<Attr>`, and so forth.

`protected void finalize() throws Throwable`

Finalizes the object during garbage collection.

`public String toString()`

Returns a string representation of the object (implicitly invoked whenever an object reference is used within a string concatenation expression as an operand of the + operator). The `Object` version of `toString` constructs a string containing the class name of the object, an `@` character, and a hexadecimal representation of the instance's hash code.

# Cloning objects

A given class can have one of four different attitudes toward `clone`:

- **Support `clone`.** Implements `Cloneable` and declares its `clone` method to throw no exceptions.
- **Conditionally support `clone`.** Implements `Cloneable`, but lets its `clone` method pass through any `CloneNotSupportedException` it may receive from other objects it tries to clone.
- **Allow subclasses to support `clone` but don't publicly support it.** Doesn't implement `Cloneable`, but (if the default implementation of `clone` isn't correct) provides a protected `clone` implementation that clones its fields correctly.
- **Forbid `clone`.** Does not implement `Cloneable` and provides a `clone` method that always throws `CloneNotSupportedException`.

`Object.clone` checks whether the object on which it was invoked implements the `Cloneable` interface and throws `CloneNotSupportedException` if it does not.

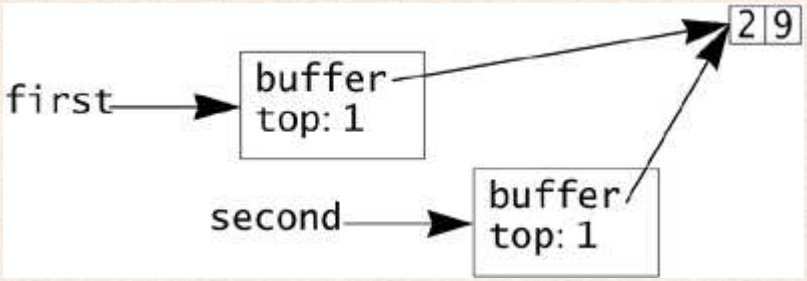
Just make a shallow copy (do not clone the fields):

```
public class MyClass extends HerClass implements Cloneable
{

    public MyClass clone()
        throws CloneNotSupportedException {
        return (MyClass) super.clone();
    }
    // ...
}
```

Correct clone (deep copy):

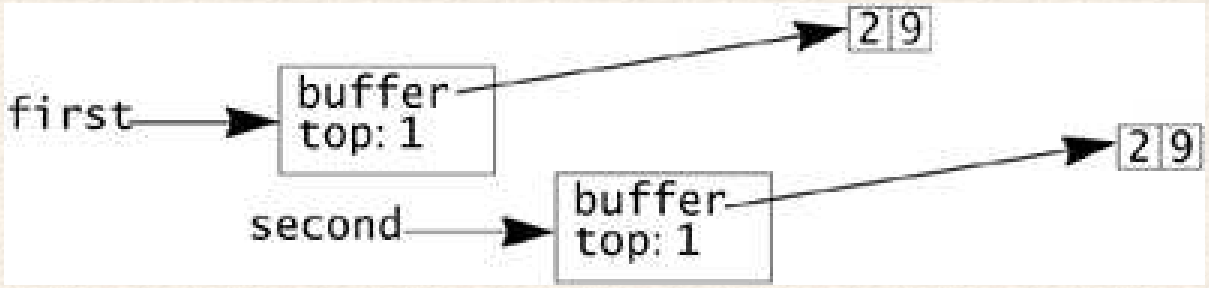
```
public class IntegerStack implements Cloneable {
    private int[] buffer;
    private int top;
    public IntegerStack(int maxContents) {
        buffer = new int[maxContents];
        top = -1;
    }
    public void push(int val) {
        buffer[++top] = val;
    }
    public int pop() {
        return buffer[top--];
    }
    public IntegerStack clone() {
        try {
            IntegerStack nObj = (IntegerStack) super.clone();
            nObj.buffer = buffer.clone();
            return nObj; // when called through super, will give
                         // the actual subclass, not IntegerStack
        } catch (CloneNotSupportedException e) {
            // Cannot happen -- we support clone
            throw new InternalError(e.toString());
        }
    }
}
```

Usually incorrect:




```
return (IntegerStack) super.clone();
```

Correct:




```
IntegerStack nObj = (IntegerStack) super.clone();
nObj.buffer = buffer.clone();
return nObj;
```

Cannot use `final int[] buffer;` because `clone` is not a constructor.

# Extending classes

**IsA** vs. **HasA**: a `Manager` *IsA* `Employee`, or a `Manager` *IsA* `Role`, and an `Employee` *HasA* `Role[]`?

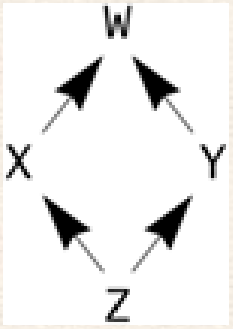
The public API is for programmers using your class. The protected API is for programmers extending your class. Do not casually make fields of your classes protected: for high-level classes, getter and setter methods are better.

If the extending classes are not trusted:

- use `protected final` methods and `private` fields
- return cloned objects from getter methods

```
abstract class SortDouble {
  private double[] values;
  private final SortMetrics curMetrics
= new SortMetrics();
  /** Invoked to do the full sort */
  public final SortMetrics
sort(double[] data) {
      values = data;
      curMetrics.init();
      doSort();
      return getMetrics();
  }
  public final SortMetrics
getMetrics() {
      return curMetrics.clone();
  }
  /** For extended classes to know the
number of elements*/
  protected final int getDataLength()
{
      return values.length;
  }
  /** For extended classes to probe
elements */
  protected final double probe(int i)
{
      curMetrics.probeCnt++;
      return values[i];
  }
  /** For extended classes to compare
elements */
  protected final int compare(int i,
int j) {
      curMetrics.compareCnt++;
      double d1 = values[i];
      double d2 = values[j];
      if (d1 == d2)
          return 0;
      else
          return (d1 < d2 ? -1 : 1);
  }
  /** For extended classes to swap
elements */
  protected final void
  swap(int i, int j) {
      curMetrics.swapCnt++;
      double tmp = values[i];
      values[i] = values[j];
      values[j] = tmp;
  }
  /** Extended classes implement this
-- used by sort */
  protected abstract void doSort();
}
```

**Why not multiple inheritance?**




In the diamond inheritance situation if both `X` and `Y` override the same method of `W`, it is unclear which method should `Z` use.

In Java, `X`, `Y` and `W` can be interfaces:

then there is only single implementation provided by `Z`.

# Interfaces

Interfaces define types in an abstract form as a collection of methods or other types that form the contract for that type.

A reference to an object of a subtype can be used polymorphically anywhere a reference to an object of any of its supertypes (extended class or implemented interface) is required.

Some simple "ability" interfaces:

- `Cloneable` Objects of this type support cloning,
- `Comparable` have an ordering that allows them to be compared,
- `Runnable` represent a unit of work that can execute in an independent thread of control,
- `Serializable` can be written to an object byte stream for shipping to a new virtual machine, or for storing persistently.

```
public interface Comparable<T> { int compareTo(T obj); }
```

The natural ordering for points could be their distance from the origin:

```
class Point implements Comparable<Point> {
    /** Origin reference that never changes */
    private static final Point ORIGIN = new Point();
    private int x, y;
    // ... definition of constructors, setters and accessors
    public double distance(Point p) {
        int xdiff = x - p.x;
        int ydiff = y - p.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
    }
    public int compareTo(Point p) {
        double pDist = p.distance(ORIGIN);
        double dist  = this.distance(ORIGIN);
        if (dist > pDist)
            return 1;
        else if (dist == pDist)
            return 0;
        else
            return -1;
    }
}
```

```
Comparable<Point> obj = new Point(); // OK: interface as
type
double dist = obj.distance(p1); // INVALID: Comparable has
                                // no distance method
String desc = obj.toString(); // OK: interfaces implicitly
                              // extend Object
```

An interface can declare three kinds of members:

- constants (fields)
- methods
- nested classes and interfaces

All interface members are implicitly public.

Interface constants are fields – implicitly `public`, `static`, and `final`.

Interface methods are implicitly `abstract`.

```
interface Verbose {
    int SILENT  = 0;
    int TERSE   = 1;
    int NORMAL  = 2;
    int VERBOSE = 3;

    void setVerbosity(int level);
    int getVerbosity();
}
```

Above better to define an `Enum` type inside the interface.

If you need shared, modifiable data in your interface you can achieve this by using a named constant that refers to an object that holds the data.

Extending more than one interface:

```
public interface SerializableRunnable
                 extends java.io.Serializable, Runnable
{
    // ...
}
```

## Inheriting and Hiding Constants

```
interface X {
    int val = 1;
}
interface Y extends X {
    int val = 2;
    int sum = val + X.val;
}
class Z implements Y { }
```

```
interface C {
    String val = "Interf
C";
}
interface D extends X, C {
}
```

you can do

```
System.out.println("Z.val=" + Z.val + ", Z.sum=" + Z.sum);
```

but inside `D` you would have to explicitly use `X.val` or `C.val`.

## Inheriting, Overriding, and Overloading Methods

If a class implements several interfaces containing a method with the same signature, then the return type of one of them must be a subtype of all the others, and the method returns this common subtype. (What about `throws`?)

The problem is whether a single implementation of the method can honor all the contracts implied by that method being part of the different interfaces.

```
interface CardDealer {
    void draw();            // flip top card
    void deal();            // distribute cards
    void shuffle();
}
interface GraphicalComponent {
    void draw();            // render on default device
    void draw(Device d);    // render on 'd'
    void rotate(int degrees);
    void fill(Color c);
}
interface GraphicalCardDealer
    extends CardDealer, GraphicalComponent { }
```

## Using Interfaces

We will create an `Attributed` type to be used for objects that can be attributed by attaching `Attr` objects to them.

```
public interface Attributed {
    void add(Attr newAttr);
    Attr find(String attrName);
    Attr remove(String attrName);
    java.util.Iterator<Attr> attrs();
}
```

`Iterator` is a generic interface defined in `java.util` for collection classes to use to provide access to their contents.

```
import java.util.*;

class AttributedImpl implements Attributed, Iterable<Attr> {
    protected Map<String, Attr> attrTable =
        new HashMap<String, Attr>(); // String has good hashCode method

    public void add(Attr newAttr) {
        attrTable.put(newAttr.getName(), newAttr);
    }
    public Attr find(String name) {
        return attrTable.get(name);
    }
    public Attr remove(String name) {
        return attrTable.remove(name);
    }
    public Iterator<Attr> attrs() {
        return attrTable.values().iterator();
    }
    public Iterator<Attr> iterator() {
        return attrs();
    }
}
```

## Forwarding

By composition and forwarding, *IsA* can become a *HasA*:

```
import java.util.Iterator;

class AttributedBody extends Body
    implements Attributed
{
    private AttributedImpl attrImpl = new AttributedImpl();
    public AttributedBody() {
        super();
    }
    public AttributedBody(String name, Body orbits) {
        super(name, orbits);
    }

    // Forward all Attributed methods to the attrImpl object
    public void add(Attr newAttr)
        { attrImpl.add(newAttr); }
    public Attr find(String name)
        { return attrImpl.find(name); }
    public Attr remove(String name)
        { return attrImpl.remove(name); }
    public Iterator<Attr> attrs()
        { return attrImpl.attrs(); }
}
```

"Pros" and "cons" of forwarding:

- both straightforward and much less work than implementing `Attributed` from scratch
- enables you to quickly change the implementation you use, should a better implementation of `Attributed` become available
- must be set up manually and that can be tedious and sometimes error prone

Any major class you expect to be extended, whether abstract or not, should be an implementation of an interface.

## Marker Interfaces

- do not declare any methods nor values but simply mark a class as having some general property
- all their contract is in the documentation that describes the expectations you must satisfy if your class implements that interface
- examples: `Cloneable`, `Serializable`, `java.rmi.Remote`, `java.util.EventListener`

# Nested Classes and Interfaces

- nested classes and nested interfaces allow types to be structured and scoped into logically related groups (static nested types)
- nested classes can be used to connect logically related objects simply and effectively (for example, event frameworks like GUIs)
- a nested type is considered a part of its enclosing type and each can access all members of the other
  - a static nested class can access private members of the enclosing class (through an object reference)
- nested interfaces are always static

## Static Nested Types

- A nested class or interface that is declared as a `static` member of its enclosing class or interface acts just like any non-nested, or top-level, class or interface. The name of a nested type is expressed as `EnclosingName.NestedName`.
- Because static nested types are members of their enclosing type, the same accessibility rules apply to them as for other members. The nested type is accessible only if the enclosing type is accessible.
- A static nested class can extend any other class (including the class it is a member of),[1] implement any interface and itself be used for further extension by any class to which it is accessible.

```
public class BankAccount {
    private long number;    // account number
    private long balance;   // current balance (in cents)

    public static class Permissions {
        public boolean canDeposit,
                       canWithdraw,
                       canClose;
        public Permissions (BankAccount account) {
            canWithdraw = account.balance > 0;
            canDeposit = true; canClose = true;
        }
    }
    // ...
}
```

later:

```
BankAccount.Permissions perm = acct.permissionsFor(owner);
```

# Inner Classes

- Non-static nested classes are called inner classes.
- An instance of an inner class is associated with an instance of its enclosing class (called the enclosing instance or enclosing object).
  - Usually, inner class objects are created inside instance methods of the enclosing class.
  - In general, we create an `InnerClass` object by treating `new InnerClass` as a method of the enclosing class.
  - The enclosing class members are said to be in scope: e.g. inner class method can refer to enclosing class fields by name.
  - **Qualified-`this`**: can refer to enclosing object of a specific class by `OuterClass.this`. There's **qualified-`super`** as well.
- Inner classes cannot have static members (including static nested types), except for (final static fields initialized to) constants.

```
public class BankAccount {
    private long number;    // account number
    private long balance;   // current balance (in cents)
    private Action lastAct; // last action performed

    public class Action {
        private String act;
        private long amount;
        Action(String act, long amount) {
            this.act = act;
            this.amount = amount;
        }
        public String toString() {
            // identify our enclosing account
            return number + ": " + act + " " + amount;
        }
    }
    public void deposit(long amount) {
        balance += amount;
        lastAct = new Action("deposit", amount);
    }
    public void withdraw(long amount) {
        balance -= amount;
        lastAct = new Action("withdraw", amount);
    }
    // ...
}
```

A transfer needs to update the `lastAct` field of both account objects:

```
public void transfer(BankAccount other, long amount) {
    other.withdraw(amount);
    deposit(amount);
    lastAct = this.new Action("transfer", amount);
    other.lastAct = other.new Action("transfer", amount);
}
```

## Extending Inner Classes

Extended inner class is often declared within an extension of the outer class:

```
class Outer {
    class Inner { }
}
class ExtendedOuter extends Outer {
    class ExtendedInner extends Inner { }
    Inner ref = new ExtendedInner();
}
```

If the enclosing class of the inner subclass is not a subclass of `Outer`, or if the inner subclass is not itself an inner class, then an explicit reference to an object of `Outer` must be given:

```
class Unrelated extends Outer.Inner {
    Unrelated(Outer ref) {
        ref.super();
    }
}
Outer ref = new Outer();
Unrelated u = ref.new Unrelated(); // INVALID -- not an inner class
```

An inner class's own fields and methods (and nested types), either declared or inherited, can hide those of the enclosing object. The enclosing object's field or method must be accessed explicitly using a qualified-`this` expression.

```
class Host {
    int x;
    class Helper extends Unknown {
        void increment() { x++; }  // Not what you think!
    }
}
```

**Confusing**: the reference to `x` should be explicitly qualified as either `this.x` or `Host.this.x`.

An inner class method with the same name as an enclosing class method hides all overloaded forms of the enclosing class method.

```
class Outer {
    void print() { }
    void print(int val) { }

    class Inner {
        void print() { }
        void show() {
            print();
            Outer.this.print();
            print(1);    // INVALID: no Inner.print(int)
        }
    }
}
```

Overloading is supposed to implement the same contract for different types.

# Local Inner Classes

- You can define inner classes in code blocks, such as a method body, constructor, or initialization block. They are local to that block, just as a local variable is.
- Because local inner classes are inaccessible, they can't have access modifiers.
- A local inner class can access all the variables that are in scope where the class is defined: local variables, method parameters, instance variables (assuming it is a non-static block), and static variables, but:
- a local variable or method parameter can be accessed only if it is declared `final`.
  - If needed, you can copy a non-final variable into a final one that is subsequently accessed by the local inner class.

`Iterator` from the `java.util` package defines a way to iterate through a group of objects:

```
package java.util;

public interface Iterator<E> {
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove() throws UnsupportedOperationException,
                         IllegalStateException;
}
```

- The `hasNext` method returns `true` if `next` can return another `E`.
- The `remove` method removes from the collection the last element returned by `next`.
- If `remove` is invoked before `next`, or is invoked multiple times after a single call to `next`, then `IllegalStateException` is thrown.
- `NoSuchElementException` is thrown if `next` is invoked when there are no more elements.

Example: return an `Iterator` to walk through an array of `Object` instances:

```
public static Iterator<Object>
    walkThrough(final Object[] objs) {
    class Iter implements Iterator<Object> {
        private int pos = 0;
        public boolean hasNext() {
            return (pos < objs.length);
        }
        public Object next() throws NoSuchElementException {
            if (pos >= objs.length)
                throw new NoSuchElementException();
            return objs[pos++];
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    return new Iter();
}
```

## Anonymous Inner Classes

Anonymous classes are defined at the same time they are instantiated with `new`. The `walkThrough` method could use an anonymous inner class:

```
public static Iterator<Object> walkThrough(final Object[] objs) {

    return new Iterator<Object>() {
        private int pos = 0;
        public boolean hasNext() {
            return (pos < objs.length);
        }
        public Object next() throws NoSuchElementException {
            if (pos >= objs.length)
                throw new NoSuchElementException();
            return objs[pos++];
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
```

- Because `Iterator` is an interface, the anonymous class in `walkThrough` implicitly extends `Object` and implements `Iterator`.
- An anonymous class cannot have an explicit `extends` or `implements` clause, and cannot have any modifiers.
- The `new` operator that constructs an anonymous class corresponds to explicitly calling `super` in a constructor; remaining initialization can be done in an initialization block.

```
Attr name = new Attr("Name") {
    public Object setValue(Object nv) {
        System.out.println("Name set to " + nv);
        return super.setValue(nv);
    }
};
```

- In the `walkThrough` example the sole purpose of the method is to return that object, but when a method does more, anonymous classes should be kept small (e.g. up to six lines), for clarity.

# Inheriting Nested Types

Consider a framework that models devices that can be connected by different ports. During construction of a concrete device class, references to the concrete ports of that class are initialized:

```
abstract class Device {
    abstract class Port {
        // ...
    }
    // ...
}
class Printer extends Device {
    class SerialPort extends Port {
        // ...
    }
    // problem: will not be overriden!
    Port serial = new SerialPort();
}
```

A concrete device can itself be extended, as can the concrete inner port classes, to specialize their behavior. Abstracting construction of the inner class objects into a factory method allows overridding in a subclass to construct the right kind of inner class:

```
class Printer extends Device {
    class SerialPort extends Port {
        // ...
    }
    Port serial = createSerialPort();
    protected Port createSerialPort() {
        return new SerialPort();
    }
}
class HighSpeedPrinter extends Printer {
    class EnhancedSerialPort extends SerialPort {
        // ...
    }
    protected Port createSerialPort() {
        return new EnhancedSerialPort();
    }
}
```