The *Object* class
• Every base class implicitly extends the Object class

• The Object class provides some useful functionality for all classes

• The Object class is special in that the Java compiler and execution environment know about this.
public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:
JDK1.0

See Also:
class

Constructor Summary

Constructors

Constructor and Description
Object()

Method Summary

Methods

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected Object</td>
<td>close()</td>
</tr>
<tr>
<td></td>
<td>Creates and returns a copy of this object.</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(Object obj)</td>
</tr>
<tr>
<td></td>
<td>Indicates whether some other object is &quot;equal to&quot; this one.</td>
</tr>
<tr>
<td>protected void</td>
<td>finalize()</td>
</tr>
<tr>
<td></td>
<td>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.</td>
</tr>
<tr>
<td>Class&lt;?&gt;</td>
<td>getClass()</td>
</tr>
<tr>
<td></td>
<td>Returns the runtime class of this Object.</td>
</tr>
</tbody>
</table>
What do we inherit from Object?

protected Object clone() Creates and returns a copy of this object.

boolean equals(Object obj) Indicates whether some other object is "equal to" this one.

protected void finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Class<? extends Object> getClass() Returns the runtime class of an object.

int hashCode() Returns a hash code value for the object.

String toString() Returns a string representation of the object.
Overriding equals and hashcode


(javadoc) must define an equivalence relation (it must be reflexive, symmetric, and transitive). In addition, it must be consistent (if the objects are not modified, then it must keep returning the same value). Furthermore, `o.equals(null)` must always return false.

`hashCode()` (javadoc) must also be consistent (if the object is not modified in terms of `equals()`, it must keep returning the same value).

The relation between the two methods is:

*Whenever* `a.equals(b)`, then `a.hashCode()` must be same as `b.hashCode()`. (if `a` is null then a null pointer exception is thrown)

Use the same object fields to compute equals and hashcode.
What do we inherit from Object?

void *notify() Wakes up a single thread that is waiting on this object's monitor.

void *notifyAll() Wakes up all threads that are waiting on this object's monitor.

void *wait() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

void *wait(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

void *wait(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.
Making copies of objects

• Copy Constructors
• Clone methods
Copy Constructors

• You can create a constructor T(T t) to make copies.
  - T r = new T(tr);
  - Because Java never actually passes objects to functions (only references to objects), copy constructors are not used as much in C++ as in Java

• Java also supports a clone method.
Class User {
    public String name;
    public int age;
    public User(String str, int a) {
        name = str; age = a;
    }
}

Class Test {
    public static void main(String[] args) {
        User u1 = new User("Ralph", 112);
        System.out.println(u1.name); // Ralph
        User u2 = u1;
        u2.name = "Bond";
        System.out.println(u1.name); // Bond
    }
}
Class User {
    public String name;
    public int age;
    public User(String str, int a) {
        Name = str; age = a;
    }
}

Class Test {
    public static void main(String[] args) {
        User u1 = new User("Ralph", 112);
        System.out.println(u1.name); // Ralph
        User u2 = u1;
        u2.name = "Bond";
        System.out.println(u1.name); // Bond
    }
}
Class User {
    public String name;
    public int age;
    public User(String str, int a) {
        Name = str; age = a;
    }
}

Class Test {
    public static void main(String[ ] args) {
        User u1 = new User("Ralph", 112);
        System.out.println(u1.name); // Ralph
        User u2 = u1;
        u2.name = "Bond";
        System.out.println(u1.name); // prints Bond
    }
}

But what if we want u2 to reference a copy of what u1 references?
Sometimes a copy or *clone* of an object is desired

- Even though Java only copies references, it is sometimes desirable that an assignment put a reference to a new object in the left hand side (LHS) variable.
- *Cloning* is the Java mechanism for accomplishing this.
- The class for objects to be cloned *must* implement the interface *Cloneable*.
- Invoking the *clone* method on an object produces a clone of the object.
• Even though X is a very simple class, it cannot invoke clone because it does not implement cloneable.
• We can duplicate X and its state, as shown below:

```java
class X {
    int n;
    X() {n = 3;}
    X(int n) {
        this.n = n;
    }
    getN() {
        return n;
    }
}
X xobj = new X(4);
Y yobj = new X(xobj.getN());
```
• But . . .
  • What if X was a hashmap or something else complicated?
  • What about private fields?
  • Encapsulation implies objects should copy themselves.
class X implements Cloneable {
    int n;
    X() {n = 3;}
    X(in n) {
        this.n = n;
    }
    int getN() {
        return n;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone();
The `Object clone` function (super.clone( )) makes a byte-by-byte copy of the object referenced by `xobj` and returns a reference to it.

```java
class X implements Cloneable {
    int n;
    X( ) {n = 3;}
    X(int n) {
        this.n = n;
    }
    int getN( ) {
        return n;
    }
    public Object clone( ) throws CloneNotSupportedException {
        return super.clone( );
    }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone( );
```
class X implements Cloneable {
    int n;
    X( ) {n = 3;}
    X(int n) {
        this.n = n;
    }
    int getN( ) {
        return n;
    }
    public Object clone( ) throws CloneNotSupportedException {
        return super.clone( );
    }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone( );
The interface cloneable *is not a normal interface*

- *Cloneable* is empty -- a class implementing it doesn't have to actually implement anything (but can if it wants)
- but if a *public Object clone()* method is not implemented an error will result if an attempt is made to clone the object.
- *Implements Cloneable* is a signal to the *Object* class that it is ok for *Object’s clone* to clone this object w/a byte for byte copy.
- This leverages the fact that *Object* is not a normal class. It, and some other systems classes, perform functionality not expressible in Java.
Calling clone

import java.util.*;
class X implements Cloneable {
    public int n;
    public X( ) {n=3;}
    public Object clone( ) throws CloneNotSupportedException {
        return super.clone( );
    }
}

try {
    xobj_clone = (X) xobj.clone( );
} catch (CloneNotSupportedException e) { ... }
...
Consider class X

class X implements Cloneable {
    public int[] arr = new int[5]
    public X() {
        Random ran = new Random();
        int i = 0;
        while (i < 5) {
            arr[i++] = ran.nextInt(10);
        }
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

X xobj = new X();
X xobj = new X();

X xobjClone = xobj.clone();

The default clone called in super (Object) will do this

What if we also wanted arr cloned?
What if we want this?

```
X xobj = new X();
...
X xobjClone = xobj.clone();
```

We need to write our own clone function that does something useful.
// X is as before
public Object clone( ) throws CloneNotSupportedException {
    X xob = null;
    xob = (X) super.clone( );
    // now clone the array
    xob.arr = (int[ ]) arr.clone( );
    return xob;
}

... public static void main(String[ ] args) throws Exception {
    x xobj = new X( );
    x xobjClone = (X) xobj.clone( );
    System.out.println(xobj); // 0 4 5 2 5
    System.out.println(xobjClone); // 0 4 5 2 5
    xobj.arr[0] = 1000;
    System.out.println(xobj); // 1000 4 5 2 5
    System.out.println(xobjClone); // 0 4 5 2 5
}
public Object clone() throws CloneNotSupportedException {
    X xob = null;
    xob = (X) super.clone();
    // now clone the array
    xob.arr = (int[]) arr.clone();
    return xob;
}
public Object clone( ) throws CloneNotSupportedException {
    X xob = null;
    xob = (X) super.clone( );
    // now clone the array
    xob.arr = (int[ ]) arr.clone( );
    return xob;
}
What about arrays of references?

```
header | arr
0     | 1     | 2     | 3     | 4
obj_1 | obj_2 | obj_n
```
An example of cloning with arrays of objects

```java
public class I implements Cloneable {

    int i;
    public I() {i = 0;}
    public I(int i) {this.i = i;}
    public void print() {System.out.println("i: "+i);}
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

I is a class that holds an integer (could use Integer)
public class L implements Cloneable {

    I arryl[ ];

    public L( ) {
        arryl = new I[5];
        for (int i = 0; i < 5; i++) {
            arryl[i] = new I(i);
        }
    }

    void print(String s) {
        System.out.println("Printing L object "+s+":");
        for (int i = 0; i < arryl.length; i++)
            arryl[i].print( );
    }

    public void setElement(int i, int v) {
        arryl[i] = new I(v);
    }

    public Object clone( ) throws CloneNotSupportedException { // see next slide }
The clone method

public Object clone( ) throws CloneNotSupportedException {
    L lClone = (L) super.clone( );
    // lClone.arryI = arryI.clone( ); WRONG by itself and
    // inefficient in the best case (recopies contents of arryI which
    // are then written over
    I arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone( );
    lClone.arryI = arryClone;
    return lClone;
}
Let's see what this does pictorially

```java
public Object clone() throws CloneNotSupportedException {
    L lClone = (L) super.clone();
    // lClone.arryI = arryI.clone(); // Inefficient
    I arryClone[] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone();
    lClone.arryI = arryClone;
    return lClone;
}
```
Let's see what this does pictorially

```java
public Object clone() throws CloneNotSupportedException {
    L IClone = (L) super.clone();
    // IClone.arryI = arryI.clone(); // Inefficient
    I arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone();
    IClone.arryI = arryClone;
    return IClone;
}
```
Let's see what this does pictorially

```java
public Object clone() throws CloneNotSupportedException {
    L lClone = (L) super.clone();
    lClone.arryI = arryI.clone(); // Why is this suboptimal?
    // I arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone();
    lClone.arryI = arryClone;
    return lClone;
}
```

Creates storage for `arryI`, copies data from `this.arryI`, but doesn't clone the actual objects. There is a wasted copy.
public Object clone( ) throws CloneNotSupportedException {
    L IClone = (L) super.clone( );
    // IClone.arryI = arryI.clone( );; WRONG
    IarryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        IarryClone[i] = (I) arryI[i].clone( );
    IClone.arryI = arryClone;
    return IClone;
}

Creates storage for IarryClone.

IClone → header arryI

this → header arryI

IarryClone

null null null null null null
Let's see what this does pictorially

public Object clone( ) throws CloneNotSupportedException {
    L IClone = (L) super.clone( );
    // IClone.arryI = arryI.clone( );  WRONG
    IarryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        IarryClone[i] = (I) arryI[i].clone( );
    IClone.arryI = arryClone;
    return IClone;
}

clone the objects pointed to by the elements of IarryClone
Let's see what this does pictorially

```java
public Object clone() throws CloneNotSupportedException {
    L lClone = (L) super.clone();
    // lClone.arryl = arryI.clone(); WRONG
    IarryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        larryClone[i] = (I) arryI[i].clone();
    lClone.arryl = arryClone;
    return lClone;
}
```

clone the objects pointed to by the elements of larryClone
Let's see what this does pictorially

public Object clone( ) throws CloneNotSupportedException{
    L IClone = (L) super.clone( );
    // IClone.arryI = arryI.clone( ); WRONG
    arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone( );
    IClone.arryI = arryClone;
    return IClone;
}

clone the objects pointed to by the elements of arryClone
Let's see what this does pictorially

```java
public Object clone() throws CloneNotSupportedException {
    L lClone = (L) super.clone();
    // lClone.arryI = arryI.clone(); WRONG
    IarryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        larryClone[i] = (I) arryI[i].clone();
    lClone.arryI = arryClone;
    return lClone;
}
```

Make lClone.arryI reference larryClone
public Object clone( ) throws CloneNotSupportedException {
    L lClone = (L) super.clone( );
    // lClone.arrlyl = arryI.clone( ); WRONG
    arryClone[ ] = new L[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (L) arryI[i].clone( );
    lClone.arrlyl = arryClone;
    return lClone;
}
In summary . . .

public Object clone( ) throws CloneNotSupportedException {
    L IClone = (L) super.clone( );
    // IClone.arryI = arryI.clone( ); WRONG!
    I arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone( );
    IClone.arryI = arryClone;
    return IClone;
}
In summary . . .

public Object clone( ) throws CloneNotSupportedException {
    L lClone = (L) super.clone( );
    // lClone.arrayI = arrayI.clone( ); WRONG!
    I arrayClone[ ] = new I[arrayI.length];
    for (int i = 0; i < arrayI.length; i++)
        arrayClone[i] = (I) arrayI[i].clone( );
    lClone.arrayI = arrayClone;
    return lClone;
}
Clone each object in the original L object array and assign a reference to it to the new L object's array.

```java
public Object clone( ) throws CloneNotSupportedException {
    L IClone = (L) super.clone( );
    // IClone.arryI = arryI.clone( ); WRONG!
    I arryClone[ ] = new I[arryI.length];
    for (int i = 0; i < arryI.length; i++)
        arryClone[i] = (I) arryI[i].clone( );
    IClone.arryI = arryClone;
    return IClone;
}
```

In summary . . .
public class Test {

    public static void main(String[] args) throws Exception {
        L lobj = new L();
        lobj.print("lobj");
        L lobjCloned = (L) lobj.clone();
        lobjCloned.print("lobj Cloned");
        lobjCloned.setElement(2, 500);
        lobj.print("lobj");
        lobjCloned.print("lobjCloned");
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        L lobj = new L();
        lobj.print("lobj");
        L lobjCloned = (L) lobj.clone();
        lobjCloned.print("lobj Cloned");
        lobjCloned.setElement(2, 500);
        lobj.print("lobj");
        lobjCloned.print("lobjCloned");
    }
}
Change L.java's clone to ...

```java
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

Printing L object lobj:
```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Printing L object lobj Cloned:
```
i: 0
i: 1
i: 2
i: 3
i: 4
```

lobjCloned.arryl[2]=500
public class I implements Cloneable {

    int i;
    public I( ) {i = 0;}
    public I(int i) {this.i = i;}
    public void print( ) {System.out.println("i: "+i);}
    //    public Object clone( ) throws CloneNotSupportedException {
    //      return super.clone( );
    //    }
    }

javac Test.java
./L.java:16: clone() has protected access in java.lang.Object
    arryClone[i] = (I) arryI[i].clone( );
       ^
1 error

This comes from not implementing a public clone method in the class being cloned.