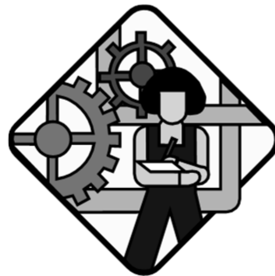
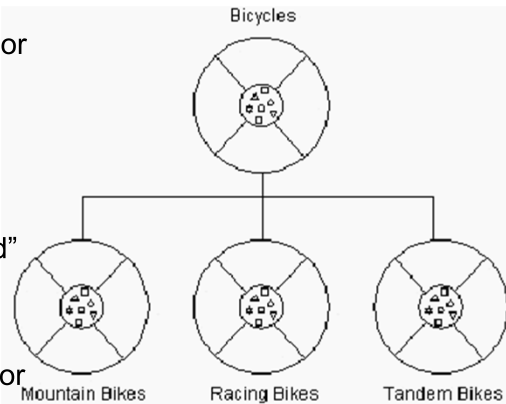
Inheritance: Reusing Objects' Structure & Methods



Understanding Inheritance

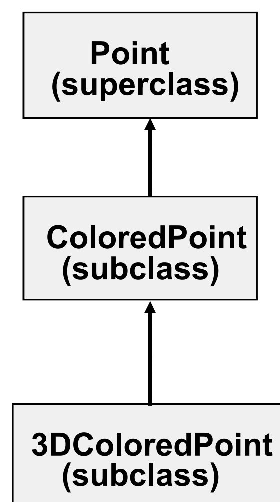
What Semantic Relationships Exist?

- What is classification?
  - Group related objects together
  - Based on attributes and behavior
- What is generalization?
  - Discovering common traits in different classes
  - Consider those classes "related"
- What is specialization?
  - Reuse all attributes and behavior of more abstract classes
  - Add additional functionality



What is inheritance?

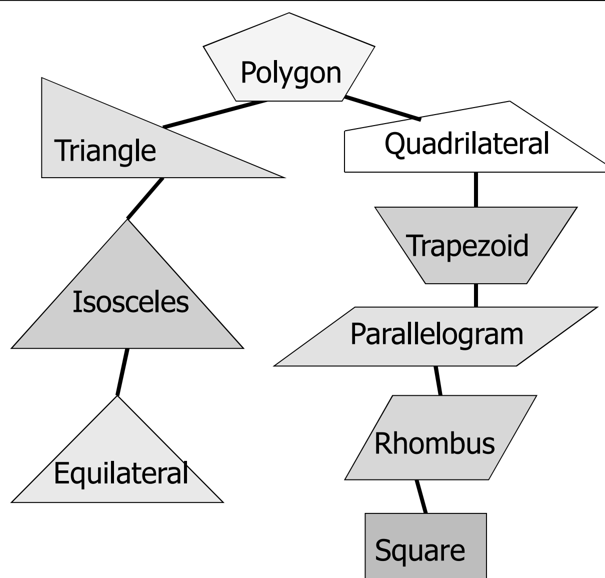
- Inheritance allows us to derive a new class from an existing one
  - The existing class is called the super-class or base-class or parent-class
  - The new class is called the subclass or derived-class or child-class
- Inherit: Instances of the derived class inherit all the properties and functionality that is defined in the base class
- Extend: Usually, the derived class adds more functionality and properties
  - In Java, the reserved word **`extends`** is used to establish an inheritance relationship between classes



Java Sub-Classing Example

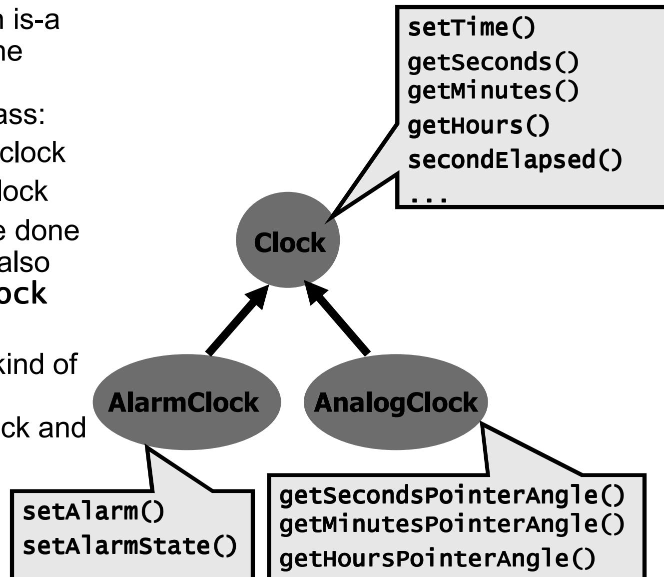
```
/**
* A ColoredPoint class represents a Point object
* with a new characteristic: its color
*/
public class ColoredPoint extends Point
// All members of super are automatically included!!
// Any additions and modifications are inserted
// explicitly (more variables and methods)
{
  private Color color;
  public void setcolor(Color color) {
     this.color = color;
  }
}
```


Inheritance Example: Shapes

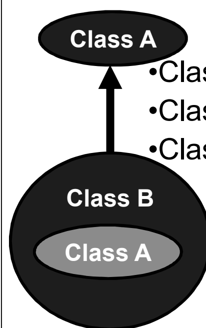


Inheritance and Is-A

- Inheritance creates an is-a relationship, meaning the child **is-a** more specific version of the parent class:
  - *AnalogClock* is a clock
  - *AlarmClock* is a clock
- Everything that can be done with a clock object can also be done with `AlarmClock` and `AnalogClock`
  - They are a special kind of clock, have all the functionality of a clock and some more
  - They are more specific than clock



Inheritance and Is-A



- Class B inherits from Class A, or
- Class B extends Class A
- Class A is the more general form while Class B is a specialization

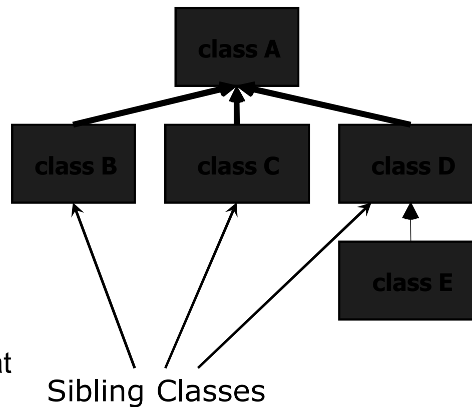
Class B contains:
- All of the properties of A
- Its own methods, not found in A (EXTENDING class A)
- Methods of A that were redefined in B (OVERRIDING)

Class B must belong to the same family as A
You must be able to say:

"B **is a** A"

Inheritance Hierarchy

- Several classes can be declared as subclasses of the same superclass
  - Good class design puts all common features as high in the hierarchy as is reasonable
- These subclasses share some structures and behaviors - what they inherit from their common superclass
  - Class hierarchies often have to be extended and modified to keep up with changing needs
- Inheritance can also extend over several "generations" of classes, creating a hierarchy of classes
- There is no single class hierarchy that is appropriate for all situations

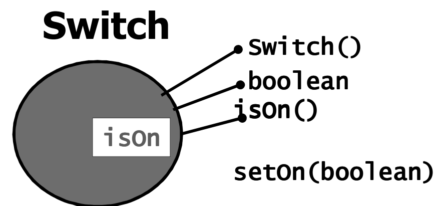


Sibling Classes

Private Members

- When you derive a class from a given base class:
  - The subclass inherits all the fields of the base class
  - It inherits all the methods of the base class
- `Private` members in the base class are inherited but they cannot be accessed directly from the code of the subclass
  - They are private and encapsulated in the base class itself

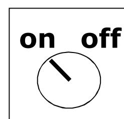
Switch



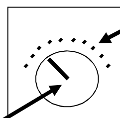
Refining the Class Switch

```
/**
 * An electronic switch that can be on/off.
 */
public class Switch {
   // Records the state of the switch
    private boolean isOn;
   /**
    * Sets the state of the switch on/off.
    */
    public void setOn(boolean state) {
       isOn = state;
    }
    /**
     * Checks if this switch is on.
     * @return true if the switch is on.
     */
    public boolean isOn() {
        return isOn;
    }
}
```

Switch vs. Adjustable Switch

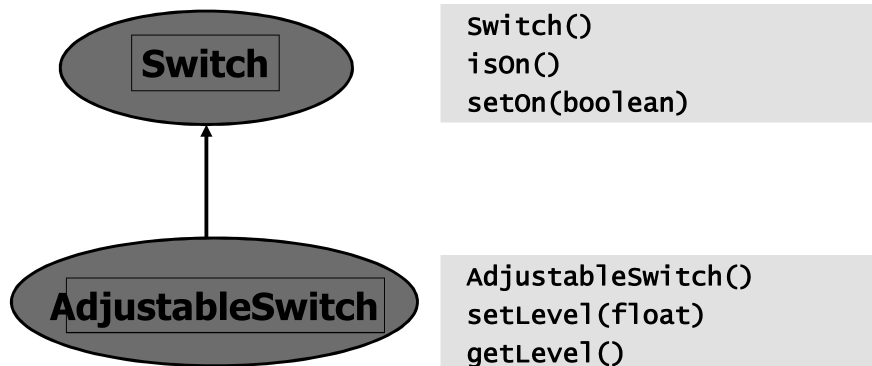


an adjustable switch has a "level of current" dial



pressing the adjustable switch turns it on

Inheriting Adjustable Switch

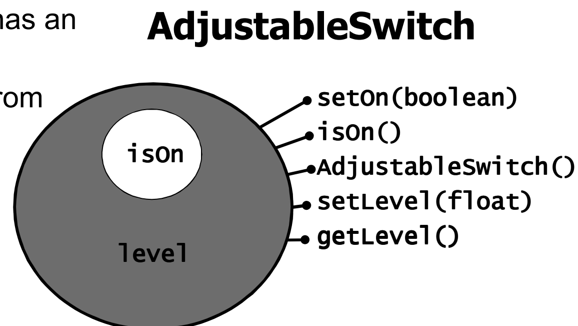


AdjustableSwitch Code

```
/**
 * An adjustable electronic switch.
 */
public class AdjustableSwitch extends Switch {
   // The level of current (0-100)
   private float level;
   /**
    * Sets the level of current
    * @param level the required level of current (0%-100%)
    */
    public void setLevel(float level) {
        if (level < 0.0f || level > 100.0f) {
             throw new IllegalArgumentException();
        }
        this.level = level;
    }
    /**
     * Returns the level of current of the switch.
     */
    public float getLevel() {
        return level;
    }
}
```

AdjustableSwitch Example

- Notice that an `AdjustableSwitch` object has an instance variable `isOn`
  - This variable is inherited from `Switch`



- However, it cannot be accessed directly from the code of `AdjustableSwitch`
  - because it is defined as private in `Switch` - i.e. it is encapsulated

Example: Changing the Base Class Switch

- Suppose that we want to add the property of '**maximal power**' to our representation of a switch
  - This property is suitable for adjustable switches as well
- Inheritance allows us to add this property to the base class
  - The change will automatically occur in the inherited class

```
/**
 * An electronic switch
 */
public class Switch {
    // Records the state of the switch
    private boolean isOn;
    // The maximal power of this switch
    private float maxPower;
…
}
```

Changing the Switch Class Methods

```
/**
 * Constructs a new switch.
 * @param power the maximal power of the switch.
 */
 public Switch(float maxPower) {
 if (maxPower <= 0.0f) {
     throw new IllegalArgumentException();
    }
    this.maxPower = maxPower;
 }
/**
 * Returns the maximal power of this switch.
 */
 public float getMaxPower() {
     return maxPower;
 }
}
```

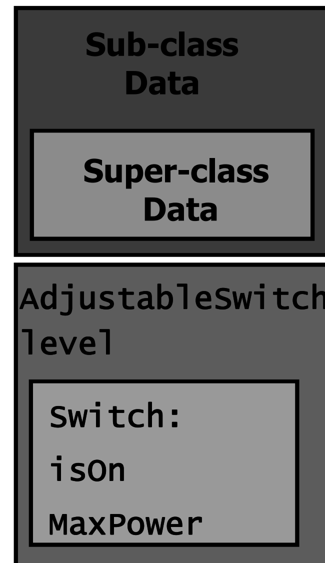
Constructors Must Be Redefined!

- Note that the constructor of `Switch` receives a parameter -`maxPower`- and initializes the private field with the same name
- In any `AdjustableSwitch` constructor we have to initialize the `maxPower` field
- Problem?
  - `maxPower` field is private in `Switch`, so there is no direct access to it from `AdjustableSwitch`
- Constructors are not inherited even though they have public visibility!
  - Constructors must be redefined!
- In general, when we invoke a constructor of a subclass, we first have to construct the superclass “part” of the subclass
  - We do this by using the `super` keyword, which calls the superclass constructor

9

Initialization of a Subclass

- Construction tasks
  - Construct parent object and initialize its fields
  - Initialize any fields having intializers
  - Perform the statements in the body of the constructor
- Constructing parent
  - Java will call default version if you don't
  - You may call overloaded constructor



Redefinition of Subclass Constructor

```
/**
 * Constructs an adjustable switch.
 * @param power the maximal power of the switch.
 * @exception IllegalArgumentException
 * if the power is negative.
 */
 public AdjustableSwitch(float power) {
    super(power);
    level = 0.0;
 }
```

In the "first line" we call the constructor of the superclass

Class Constructors and Inheritance

- If a given class does not have any constructor a **default constructor** is defined by the compiler
  - The default constructor is the **empty constructor**
  - Rules: declared values, zero, false, null
- If a subclass does not have any constructor then
  - The empty constructor of the superclass is **implicitly invoked** inside the empty constructor of the subclass itself
- Sometimes the subclass has an empty constructor, but the superclass does not (it may have at least one non-default constructor)
  - Inside the empty constructor we must **explicitly call** other superclass constructors with some default arguments
- Sometimes we can directly instantiate the superclass "part" in a subclass (whenever superclass members are not private)
  - Inside the subclass constructor we can use the keyword **this**

Subclass Constructors Examples

```
public AdjustableSwitch(){
    setLevel(0.0);
  }

public AdjustableSwitch(float power){
   this.power = power;
   this.level = 0.0;
  }

public AdjustableSwitch(float power) {
      super(power);
      level = 0.0;
   }
```

Implicit **call to** `Switch()`

None of Super's constructors will be called from here

Explicit **call to** `Switch(float)`

More on "super" and "this" Keywords

super(xxx)	calls a superclass constructor	***must be first line*** of a subclass constructor
super.xxx	accesses superclass variables	usable anywhere in the code of a sub class
super.xxx(xxx)	calls a superclass method	usable anywhere in the code of a sub class
this(xxx)	calls a current-class constructor	***must be first line*** of current class
this.xxx	accesses a current-class variable	usable anywhere in the code
this.xxx(xxx)	calls a current-class method	usable anywhere in the code
super.super.xxx	invalid statement	invalid statement

Defined vs. Inherited Class Members

- A subtle feature of inheritance is the fact that even if a method or an instance variable is not **inherited** by a child class, it is still **defined** for that class
  - An inherited member can be referenced directly in the child class, as if it were declared in that class (e.g., instance variables and methods)
  - But even members of the parent class that are not inherited exist for the child class, and can be referenced indirectly through parent class methods (e.g., constructors, overrided methods)

CSD Univ. of Crete — Fall 2012

Using Methods & Variables under Inheritance

- When a **method** is invoked using a reference **it is the class of the current object**, not the reference that determines which method to run

The code of `display()` in `Adjustable Switch will be executed`

```
Switch my_switch;
my_switch = new AdjustableSwitch();
my_switch.display();
```

- However, in the case of an instance **variable**, **it is the class of the reference**, not the class of the current object denoted by the reference, that determines which variable will actually be accessed

The variable `level` **is not** `defined at the` **Adjustable** `Switch class`

```
Switch my_switch;
my_switch = new AdjustableSwitch();
System.out.println(my_switch.level);
```

25

CSD Univ. of Crete — Fall 2012

Overriding

- A derived class inherits all methods of the base class (except constructors)
- It can then do one of the following with each inherited method:
  - Use the method as it is
  - Change it's implementation while keeping it's signature a.k.a. **Override**
- Overriding can be done only for methods!
  - A subclass can override a method implementation to provide a different version then the base class (e.g., `display()`)
  - It can add some information to the base version of the method, or it can completely re-define the base class method
- A subclass cannot override the variables of a superclass: but it can **shadow** them
  - That means, it can have its own variable of the same type as the superclass without a conflict with that of the superclass
  - A subclass can always use `super` to access the inherited member variable

26

Shadowing Members

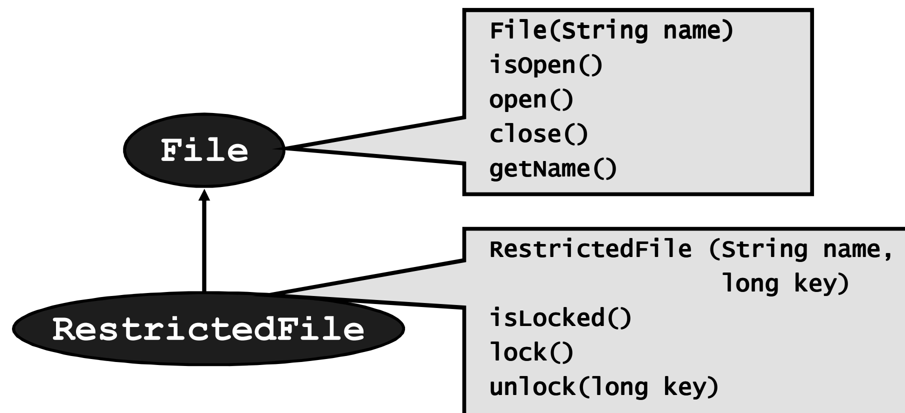
```
class B { int i = 6;  int f() { return i + i; }}
public class D extends B {
int i = 66;
int f() { return -i; };
int superf() {return super.f();}
int superi() {return super.i; }
int bcasti() {return ((B)this).i; }

public static void main(String args[])
{ D d = new D();
System.out.println(d.i);        // 66
System.out.println(d.f());      // -66
System.out.println(d.superf()); // 12
System.out.println(d.superi()); // 6
System.out.println(d.bcasti()); // 6
B b = (B) d;
System.out.println(b.i);        // 6
System.out.println(b.f()); } }  // -66
```

Name Scope and Inheritance

- Variables (e.g. `maxPower`):
  - First examines current method, looks for local variable or parameter;
  - Then examines current class (e.g. `AdjustableSwitch`);
  - Then examines superclass (e.g. `Switch`);
  - Continues up the class hierarchy until no more superclasses to examine
- Methods (e.g. `setPower(float)`):
  - First examines current class (e.g. `AdjustableSwitch`);
  - Then examines superclass (e.g. `Switch`);
  - Continues up inheritance hierarchy until no more superclasses to examine

Overriding Example

- We want clients to be able to open a protected file only if it is unlocked



Class File Example

```
public class File {
   // The name of the file
   private String name;
   // true if the file is opened
   private boolean isOpen;
   //Construct a file with a given name.
   public File(String name) {
       this.name = name;
   }
   // Returns the name of the file
   public String getName() {
      return name;
   }
   // Checks if the file is open
   public boolean isOpen() {
      return isOpen;
   }
```

Class File Example

```
// Opens the file
public void open() {
// … other operations
   isOpen = true;
}
// Closes the file
public void close() {
// … other operations
   isOpen = false;
}
```


RestrictedFile Example

```
/**
 * Represents a restricted file, which can be
 * opened only if it is unlocked. In order to
 * unlock the file a key is needed
 */
public class RestrictedFile extends File {
   // Password for unlocking the file
   private long key;
   // The state of the file - locked/unlocked
   private boolean isLocked;
```

RestrictedFile Example

```
// Constructs a new restricted file.
// The key is used to unlock the file
public RestrictedFile (String name, long key) {
       super(name);
       this.key = key;
       isLocked = true;
   }
//Checks if the file is locked
public boolean isLocked() {
        return isLocked;
   }
// Locks the file
public void lock() {
       isLocked = true;
   }
```

RestrictedFile Example

```
// Unlock the file
// The file will be unlocked only
// if the given key matches
public void unlock(long key) {
   if (this.key == key) {
       isLocked = false;
      }
  }
```

Locking?

- So far the implementation is useless if we do not change the implementation of **open()** !

```
// Open the file. The file will be
// opened only if it is unlocked
public void open() {
   if (!isLocked()) {
       super.open();
     }
   }
```

- **RestrictedFile** inherits the interface of **File**, but changes the functionality of the method **open()**
  - ◆We say that **RestrictedFile** overrides the method **open()**
- Notice the call to **super.open()** - we invoke the method **open()** of the superclass on this object

Two Kinds of Method Overriding

- Replacement
  - ◆A method completely replaces the method of the superclass that is overridden (e.g., a **toString()** routine in every subclass)
- Refinement
  - ◆The superclass method is not replaced but rather refined, that is, code is added to the superclass method.
    - This is accomplished by first calling the superclass method (e.g., **super.**abc())
  - ◆All subclass constructors use the refinement method: This is called constructor chaining.
    - Each subclass constructor begins its execution by first calling its superclass constructor (i.e., super())

Rules of Overriding

- When you derive a class B from a class A, the interface of class B will be a superset of that of class A (except for constructors)
- You cannot remove a method from the interface by subclassing (why?)
- However, class B can override some of the methods that it inherits and thus change their functionality
- The contract of a method states what is expected from an overriding implementation of the method

Flexibility in Overriding Methods

- A method can only override a method in its superclass if the superclass method is not private
- There is some flexibility in the visibility of an overriding method, whether the arguments are final or not, and the exceptions thrown by it
  - An overriding method may be more visible than that in the superclass
  - Arguments that are final in the superclass version do not have to be marked as such in the subclass version
  - Any checked exception thrown by the subclass version must match the type of the one thrown by the superclass version, or be a subclass of such an exception. However, the subclass version does not have to throw any exceptions that are thrown by the superclass version

Overriding vs. Overloading

- Don't confuse the concepts of overloading and overriding
  - Overloading deals with multiple methods in the same class with the same name but different signatures
  - Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for similar data
- Overriding lets you define a similar operation in different ways for different but compatible object types

Inheritance: a Basis for Code Reusability

- Fast implementation
  - we need not write the implementation of classes from scratch, we just implement the additional functionality
- Ease of use
  - If someone is already familiar with the base class, then the derived class will be easy to understand
- Less debugging
  - debugging is restricted to any additional functionality
- Ease of maintenance
  - if we need to correct/improve the implementation of a class, the sub-class is automatically corrected as well
- Compactness
  - our code is more compact and is easier to understand

Costs of Inheritance

- Execution speed
  - Inherited methods dealing with arbitrary subclasses are often slower than specialized code
- Program size
  - The use of any software library frequently imposes a size penalty not imposed by systems constructed for a specific code
- Message Passing Overhead
  - Passing messages is more costly than simply invoking procedures
- Program Complexity
  - Although OOP is often touted as a solution to software complexity, overuse of inheritance can often simply replace one form of complexity with another
- This does not mean you should not use inheritance, but rather than you must understand the benefits, and weigh the benefits against the costs

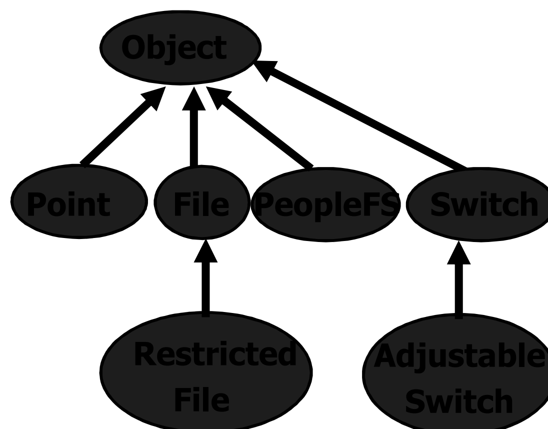
Inheritance Pluses and Minuses

- Plus
  - Reusability (design and code)
  - Protocol Consistency (e.g., in Collections and Numbers)
  - Software ICs
  - Rapid Prototyping
  - Frameworks (e.g., Swing, JCF)
  - Information Hiding
  - Lightweight Methods
- Minuses
  - Speed (generally overrated)
    - premature optimization is deadly
  - Size (if must carry many base classes)
  - Message passing overhead (higher in untyped languages)
  - Complexity (yo-yo problem)
  - Restricted visibility into foundation classes

Java Object and Wrapper Classes

The `Object` Class

- Java defines the class **`java.lang.Object`** that is defined as a superclass for all classes!
- If a class doesn't specify explicitly which class it is derived from, then it will be implicitly derived from class **`Object`** Object
  - The **`Object`** class is therefore the ultimate root of all class hierarchies
  - Any object is-a(n) **Object**



All Classes Extend `Object`

- The following two class definitions are equivalent:

```
class SomeClass
{
// some code
}
```

```
class SomeClass extends Object
{
// some code
}
```


The `Object` Class Methods

- The **`Object`** class defines a set of methods that are inherited by all classes
- This makes easier many functionalities since all objects (i.e., class instances) have some common interface

java.lang.Object

Method	Description
`clone()`	an identical copy
`equals()`	object equality
`finalize()`	release resources
`hashCode()`	a computed *hash code*
`toString()`	`"class@hashCode"`
`getClass()`	`Class` object for type
`wait(),notify()`	for threads
...	

23

The `clone()` Method

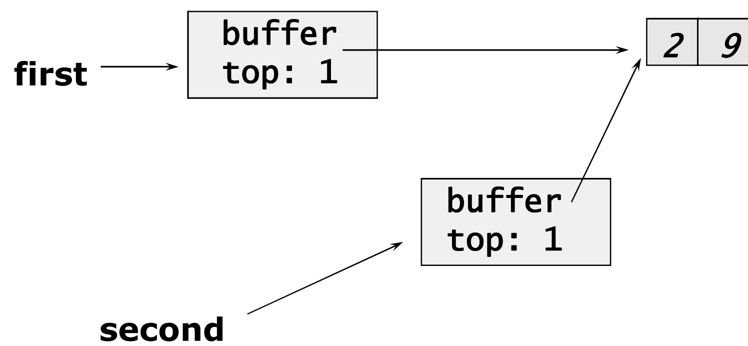
- The `clone()` method has no parameters an returns an `Object` type
  - returns a new object whose initial state is a copy of the current state of the object
- Note that what is actually returned by each class does not match the return type on the method header verses
  - Since all classes are derived from `Object`, every class “is an” `Object`
- In many classes, the default implementation of (protected) `clone()` will be wrong because it duplicates a reference to an object that shouldn’t be shared
  - A shallow copy, by default
  - A deep copy is often preferable

Creating a Shallow Copy of an Object

```
Public class IntegerStack {
      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--];
      }
}
```

```
IntegerStack first =
      new IntegerStack(2);
first.push(2);
first.push(9);
IntegerStack second =
   (IntegerStack)first.clone();
```

Creating a Shallow Copy of an Object

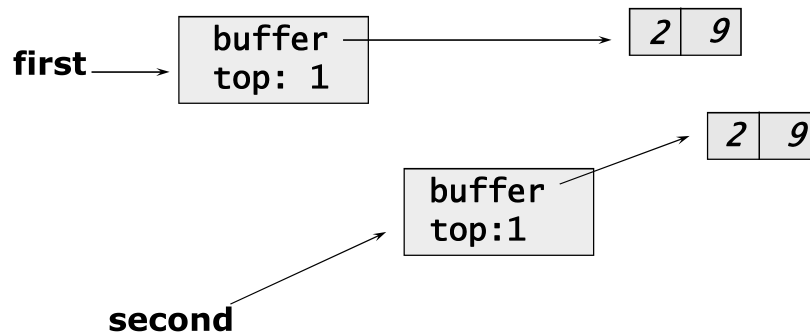


Creating a Deep Copy of an Object

Object clone method is used

```
public Object clone() {
   try {
    IntegerStack nObj = (IntegerStack)super.clone();
    nObj.buffer = (int[])buffer.clone();
   }catch (CloneNotSupportedException e){
     throw new InternalError(e.toString());
   }
}
```

Creating a Deep Copy of an Object



The `equals()` Method

- The `equals()` method receives an `Object` as a parameter
  - Since all classes are derived from `Object`, every class "is an" `Object`
- The default implementation just checks if the two references are the same
  - We can override this method in order to test object state equality

```
public class Box{
    …
    public boolean equals(Object o){
        if (o instanceof Box){
            Box b = (Box) o;
            return ((length == b.getLength()) &&
           (width == b.getWidth()) &&
           (height == b.getHeight()));
        }
    }
```

possible *ClassCastException* avoided by checking

The `instanceof` Operator

- You can restore the narrow point of view for objects, by using casting
- However at runtime you must know the true type of object referenced by some wider (base) reference (why?)
- The `instanceof` operator can be used to query about the true type of the `Object` you are referencing
  - E.g., Is this particular `Object` an instance of Class `Box`?
- Question: If Java can determine that a given `Object` is or is not a `Box` (via `instanceof`), then:
  - Why the need to cast it to a `Box` object before Java can recognize that it can `getWidth()` or `getHeight()`?
  - Why can't Java do it for us?

The `instanceof` Operator and Casting

```
if (o instanceof Box){
    Box b = (Box) o;
    return ((length == b.getLength()) &&
            (width == b.getWidth()) &&
            (height == b.getHeight()));
```

- 1st statement is legal
- 2nd statement isn't (*unless* o is Box)
- *We* can see that 1st line guarantees 2nd and 3rd are legal

- *Compiler* cannot see inter-statement dependencies… unless compiler runs whole program with all possible data sets!
- *Runtime system* could tell easily
  - We want most checking at compile-time for performance and correctness

The `instanceof` Operator and Casting

- Here, legality of each line of code can be evaluated at compile time
- Legality of each line discernable without worrying about inter-statement dependencies, i.e., each line can stand by itself
- Can be sure that code is legal (not sometimes-legal)
- A Good Use for Casting*:*
  - *Resolving polymorphic ambiguities for the compiler*

The `finalize()` Method

- The `finalize()` method is useful to release system resources
- Who runs it?
  - The garbage collector
- Unlike constructors, you can only have one `finalize` method
- A finalizer receives no parameters and return no value (e.g., its return type is **void**)
- Example:

```
public Class Box
    private static int count;
    ...
    protected void finalize(){
       --count
    }
```

The `hashCode()` Method

- **`hashCode()`** returns distinct integers for distinct objects
  - If two objects are equal according to the **`equals()`** method, then the **`hashCode()`** method on each of the two objects must produce the same integer result
  - When **`hashCode()`** is invoked on the same object more than once, it must return the same integer, provided no information used in equals comparisons has been modified
  - It is *not* required that if two objects are unequal according to **`equals()`** that **`hashCode()`** must return distinct integer values

The `toString()` Method

- The **`toString()`** method is used whenever we want to get a String representation of an object
- When you define a new class, you can override the **`toString()`** method in order to have a suitable representation of the new type of objects as Strings
- Example:

```
System.out.println (new Random() );
```

printed

```
Java.util.Random@fd78d6b6
```

shortcut for

```
System.out.println (new Random().toString() );
```

The `Class` Class

- We can navigate the type system in a program
  - instance of the class **`Class`** represents each class or interface in a running Java application
- Provide a tool to manipulate classes (**`defineClass()`** in **`Classloader`**)
  - creating objects of types specified in strings
  - loading classes using specialized techniques, such as across the network
- Two way to get a **`Class`** object
  - use **`this.getClass`** method of **`Object`**
  - fully qualified name using the static method **`Class.forName`**`(clsname)`
- The **`Class`** Class Methods
  - **`getName()`** -> a string class/interface name
  - **`getSuperclass()`** -> a **`Class`** object
  - **`getInterfaces()`** -> an array of objects representing all interfaces implemented/extended by the class/interface

Printing the Type Hierarchy of a `Class` Object

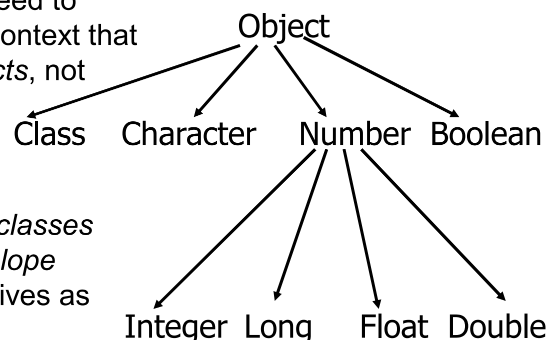
```
public class ClassInfo {
    // We expect class names as command line arguments
    public static void main(String[] args) {
    ClassInfo info = new ClassInfo();
        for(int i = 0; i < args.length; i++) {
            try {
                info.printInfo(Class.forName(args[i]), 0);
            } catch(ClassNotFoundException e) {
                System.err.println(e); // report the error
            }
   }
   // by default print on standard output
   private java.io.PrintStream out = System.out;
   // used in printInfo() for labeling type names
   private static String[]
           basic    = {“class”,   “interface”},
           extended = {“extends”, “implements”};
```

Printing the Type Hierarchy of a `Class` Object

```
  public void printInfo(Class type, int depth) {
    // Object's supertype is null
    if(type == null)
       return;
    // print out this type
    for(int i=0; i < depth; i++)
       out.print(" ");
    String[] labels = (depth == 0 ? Basic : extended);
    out.print(labels[type.isInterface() ? 1 : 0] + " ");
    out.println(type.getName());
    // print out all interface this class implements
    Class[] initerfaces = type.getInterfaces();
    for(int i = 0; i < interfaces.length ; i++)
        printInfo(interfaces[i], depth + 1);
    // recurse on the superclass
    printInfo(type.getSuperClass(), depth + 1);
  }
}
```


Wrapper Classes

- Primitive types (e.g., *int*) are not classes
- But sometimes, we may have need to make use of *primitive* types in a context that requires that we manipulate *objects*, not *primitives*
  - e.g., many collection classes are collections of Objects
- Java provides a set of *wrapper classes* (a.k.a. *type wrappers*, a.k.a. *envelope classes*) to support treating primitives as objects
- It does this by providing a specific class that corresponds to each primitive data type
- They are in `java.lang`, so the names are universally available



Wrapper Classes

Class	corresponds to	Primitive
Boolean		boolean
Character		char
Byte		byte
Short		short
Integer		int
Long		long
Float		float
Double		double

- Each one:
- allows us to *manipulate primitives as objects*
- contains useful *conversion methods*
  - E.g. Integer contains
  - `static Integer valueOf(String s)`
  - `Integer.valueOf("27")`
  - is the object corresponding to int 27
- contains useful *utility methods* (e.g. for hashing)

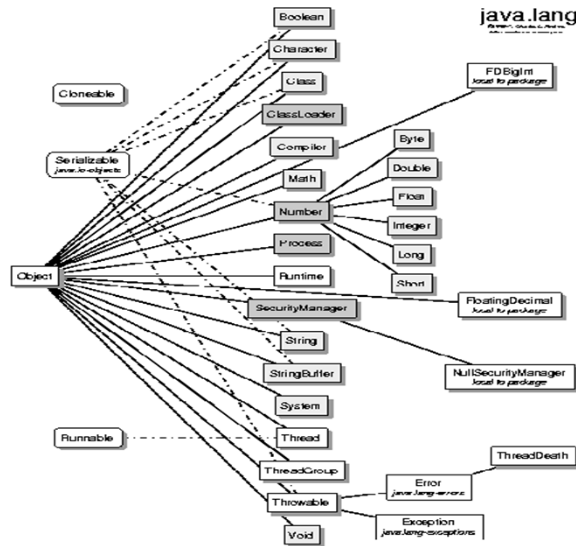
Wrapper Classes

- Using wrappers to bridge between objects and primitives:

```
// create and initialize an int
   int i = 7;

// create an Integer object and convert the int to it
   Integer intObject = new Integer( i );

// retrieve the int by unwrapping it from the object
   System.out.println( intObject.intValue );

// convert a string into an Integer object
   String strS = "27";
   Integer intObject
   intObject = new Integer (Integer.valueOf(strS) );
// then to an int
   i = intObject.intValue;
```

A class method

The `Java.Lang` Hierarchy



Understanding Polymorphism

The Notion of Polymorphism

- The Webster definition:
  - pol-y-mor-phism n. : a genetic variation that produces differing characteristics in individuals of the same population or species
  - pol-y-mor-phous adj. : having, assuming, or passing through many or various forms, stages, or the like
- The basic principle and aims of the generality and abstraction are:
  - Reuse
  - Interoperability
- The basic idea:
  - A programmer uses a single, general interface, Java selects the correct method

Principle of Substitutability

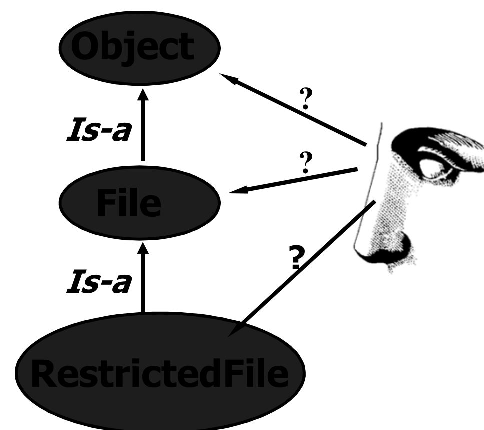
- If B is a subclass of A, then we can replace any instance of A with an instance of B in any situation with no observable effect
- If A responds to message m, then B responds to m
  - All Numbers respond to + aNumber
- Strongly typed languages require this property in many more situations than just adhering to a protocol
  - E.g., in Java, we may insist on parameters of a certain type – subtypes may be used here.
- Strongly typed languages require casts to adhere to typing rules

Subclass, Subtype, Substitutability

- Substitutability: type of a variable does not have to match the type of the actual value, subclasses are OK, too
- A subclass is usually substitutable, because:
  - Subclass instances have all parent fields
  - Subclasses implement all parent methods
  - Thus, subclasses support parent protocol
- But: not always true (see later)
- Interfaces can also be used for subtyping
- Subtype <> subclass, "stronger notion"

The Principle of Substitutability: Example

- Recall the is-a relationship induced by inheritance:
  - A `RestrictedFile` is a `File`, which is an `Object`
- `RestrictedFile` can be used by any code designed to work with `File` objects
- If a method expects a `File` object you can hand it a `RestrictedFile` object and it will work (think why?)
- This means that `RestrictedFile` object can be used as both a `File` object and a `RestrictedFile` object (and an `Object`)



The object can have many forms!

3 Views of `RestrictedFile`

- We can view a `RestrictedFile` object from 3 different points of views:
  - As a `RestrictedFile`
    - This is the most narrow point of view (the most specific)
    - This point of view 'sees' the full functionality of the object
  - As a `File`
    - This is a wider point of view (a less specific one)
    - We forget about the special characteristics the object has as a `RestrictedFile` (we can only open and close the file)
  - As a plain `Object`
    - What can we do with it?

Referencing a Subclass

- We "view" and use an object by holding and referring to the object's reference

- A variable of type 'reference to `File`' can only refer to an object which is a `File`

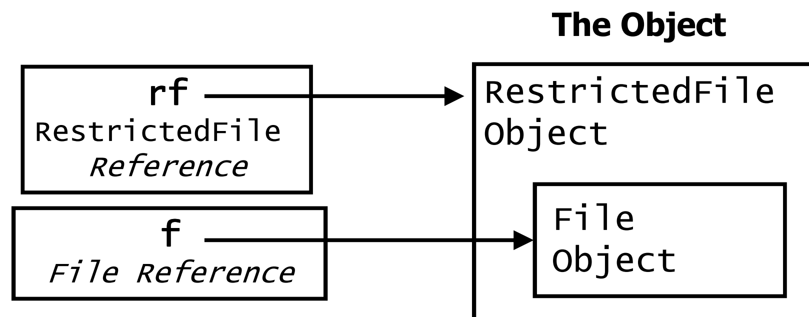
```
File f = new File("story.txt");
```

- But a `RestrictedFile` is also a `File`, so `f` can also refer to a `RestrictedFile` object

```
File f = new RestrictedFile("mydoc.txt");
```

The Reference Determines the View

- The type of the reference we use determines the point of view we will have on the object



View 1

- If we refer to a **`RestrictedFile`** object using a **`RestrictedFile`** reference we have the RestrictedFile point of view - we see all the methods that are defined in RestrictedFile and up the hierarchy tree

```
RestrictedFile rf = new
  RestrictedFile(
  "visa.dat", 12345);
✓ rf.lock();
✓ rf.unlock(12345);
✓ rf.close();
✓ String s =
  rf.toString();
```

View 2

- If we refer to a `RestrictedFile` object using a `File` reference we have the File point of view - which lets us use only methods that are defined in class File and up the hierarchy tree.

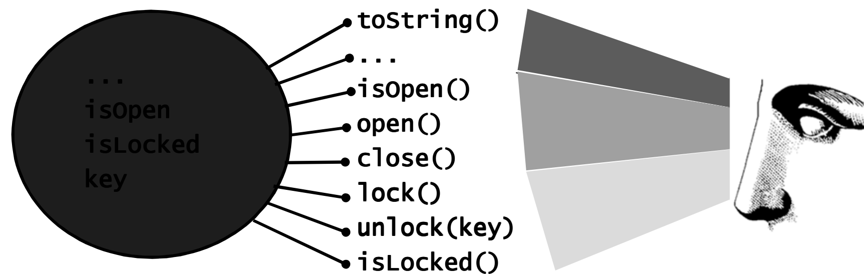
```
File f = new
 RestrictedFile(
 “visa.dat”, 12345);
✗f.lock();
✗f.unlock(12345);
✓f.close();
✓String s =
 f.toString();
```


View 3

- If we refer to a `RestrictedFile` object using an **Object** reference we have the Object point of view - which let us see only methods that are defined in class Object.

```
Object o = new
 RestrictedFile(
 “visa.dat”, 12345);
✗o.lock();
✗o.unlock(12345);
✗o.close();
✓String s =
 o.toString();
```

Multiple Views



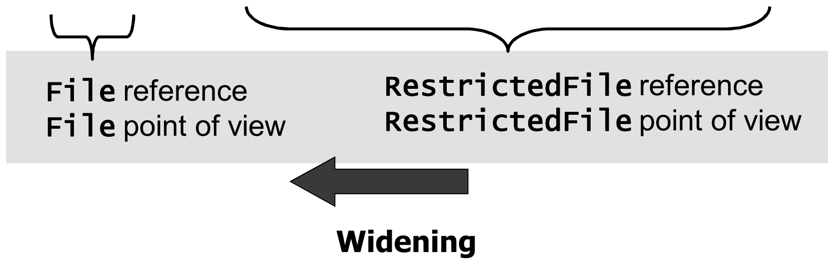
Object References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- Assigning a predecessor object to an ancestor reference is considered to be a **widening conversion** (upcast), and can be performed by simple assignment (implicit)
  - It is always possible to perform an upcast (relation *is-a* of inheritance)
- Assigning an ancestor object to a predecessor reference can also be done, but it is considered to be a **narrowing conversion** (downcast) and must be done with a cast (explicit)
  - A downcast is not always possible; at runtime, the type of the object will be checked subclass (`instanceof`)

Widening

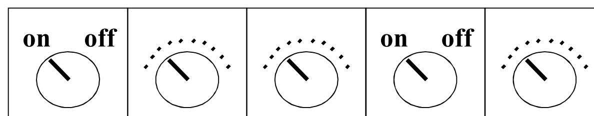
- Changing our point of view of an object, to a wider one (a less specific one) is called widening

```
File file;
file = new RestrictedFile(("visa", 1234);
```



Heterogeneous Collections

- Widening is especially useful when we want to create a heterogeneous collection of objects
- For example, we want a class `SwitchPanel` that represents a panel of switches
  - Some of the switches in the panel are simple switches and some are adjustable switches
- When implementing our `SwitchPanel` class, we would like to store switches of both kinds in a single array, so we can easily write code that operates on both kinds of switches



SwitchPanel Example

```
/**
 * Represents a panel of switches
 */
public class SwitchPanel {
    // Holds all the switches in this panel
    private Switch[] switches;
    // constants representing the types of switches
    private final static int ADJUSTABLE_SWITCH = 1;
    private final static int REGUALR_SWITCH = 0;
    // ...here come SwitchPanel() constructor
    // and getConsumption() method
}
```

getConsumption() Method in Switch

- Suppose that we add the method **getConsumption()** in class **Switch** that returns the electrical consumption of the switch (which is **0** if the switch is off and **maxPower** if the switch is on)
- This method is overridden in class **AdjustableSwitch** (the consumption is **0** if the switch is off, or **level*maxPower/100** if it is on)

```
/**
 * An electronic switch that can be on/off.
 */
public class Switch {
  // ... same implementation as before
  // Returns the electrical consumption of the switch
  public float getConsumption() {
        return (isOn() ? maxPower : 0.0f);
    }
}
```


41

getConsumption() Override

```
/**
 * An adjustable electronic switch
 */
public class AdjustableSwitch extends Switch {
  // ... same implementation as before
  // Returns the electrical consumption of the switch
  public float getConsumption() {
     return
          super.getConsumption()*level/100;
    }
}
```

- We want to implement a method **getConsumption()** in class **SwitchPanel** that will compute the total electrical consumption of all the switches in the panel, whether they are adjustable switches or regular switches

SwitchPanel Constructor

```
// model is the name of the file that describes
// the panel
public SwitchPanel(String model) {
  int numberOfSwitches =
  // ...code for reading the numbers of switches
  switches = new Switch[numberOfSwitches];
  for (int i = 0; i < numberOfSwitches; i++) {
    int switchType = // ...read the switch type
    float maxPower = //...read maxPower
    if (switchType == REGULAR_SWITCH) {
      switches[i] = new Switch(maxPower);
    } else if (switchType == ADJUSTABLE_SWITCH {
      switches[i] = new AdjustableSwitch(maxPower);
    }
  }
}
```

getConsumption() Implementation

```
/**
 * Computes the total electricity consumption
 * of all the switches in the panel.
 */
public float getConsumption() {
   float total = 0.0f;
   for (int i = 0; i < switches.length; i++) {
       total += switches[i].getConsumption();
     }
   return total;
}
```

- We do not care if `switches[i]` refers to a regular `Switch` or an `AdjustableSwitch`, they both know how to compute their current consumption

Static versus Dynamic Binding

- In OOP calling a method is often referred to as sending a message to an object
  - At runtime, the **object** responds to the message by executing the appropriate code of the method
- Binding refers to the association of a method invocation and the code to be executed on behalf of the invocation
  - When we **invoke a method** on an object we always do it through a reference
- In static binding, all the associations are determined at compile time
  - conventional function calls are statically bound
- In dynamic binding, the code to be executed in response to a method invocation (i.e., a message) will not be determined until runtime
  - method invocations to objects are dynamically bound

Polymorphic Methods

- A method is polymorphic if the action performed by the method depends on the actual **type of the object** to which the method is applied
- In Java the code of the method which will be executed is dependent on the type of object (and not on the type of reference), and this type is determined at **runtime**

```
System.out.println(ref);
```

- is a message to the object referred to by `ref` (Which message?)
- Since the object knows of what type it is, it knows how to respond to the message:
  - If a `Switch` is referenced by `ref`, then `toString` method of `Switch` class is invoked
  - If `AdjustableSwitch` is referenced by `ref`, then `toString` method of `AdjustableSwitch` class is invoked

Polymorphic Methods: Example 1

If `pt` *refers to* `Switch`

```
public String toString() {
 return
  "(" + isOn + "," + maxPower ")";
}
```

```
System.out.println(pt);
```

If `pt` *refers to* `AdjustableSwitch`

```
public String toString() {
 return
  "(" + isOn + "," + maxPower ","
      + level + ")";
 }
```

Polymorphic Methods: Example 2

```
File file;
if (Math.random() >= 0.5) {
    file = new File();
} else {
    file = new RestrictedFile("visa.dat", 76543);
    // Recall that a RestrictedFile is
    // locked by default
}
file.open();
```

Will the file be opened if the number tossed < 0.5? - NO!

Polymorphic Methods: Example 2

```
public void workaroundAttempt(File file) {
    File workaroundReference = file;
    workaroundReference.open();
    //...
 }
...
...
workaroundAttempt(file);
```

Since file is of type `RestrictedFile`, this will invoke `open()` which is defined on restricted files and not on regular files. The workaround attempt will fail!!!

Static Methods Are Not Polymorphic!

- When we invoke a method on an object we always do it through a reference
- Static methods can be invoked using a class name! (but can also using a reference)
- They are resolved at **compile time**, when we do not know which type of object is actually referenced
  - Therefore they are NOT virtual, they depend on the type of **reference**

Narrowing

- We have seen how widening can be used in heterogeneous collections
- But if we look at all objects from a wide point of view, and would not be able to restore a more narrow view for some of them, there would not have been much point in having an heterogeneous collection in the first place

Narrowing Example

```
/**
 * Locks all the restricted files in the array
 * @param files The array of files to be locked
 */
public static void lockRestrictedFiles(File[] files)
{
  for (int i = 0; i < files.length; i++) {
    if (files[i] instanceof RestrictedFile) {
      RestrictedFile file = (RestrictedFile)files[i];
      file.lock();
    }
  }
}
```

`RestrictedFile` point of view ← **Narrowing** `File` point of view

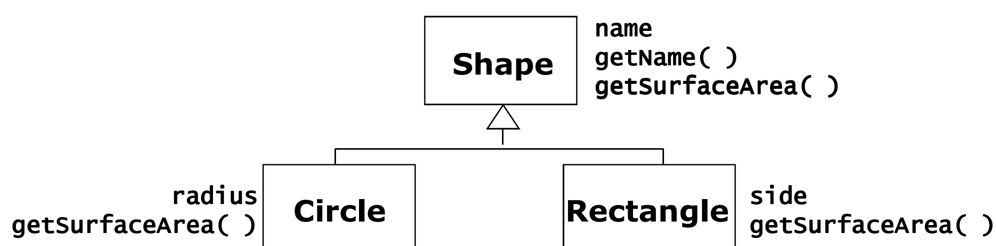
Narrowing - Equivalent Example

```
/**
 * Locks all the protected files in the array
 * @param files The array of files to be locked
 */
public static void lockRestrictedFiles(File[] files)
{
  for (int i = 0; i < files.length; i++) {
    if (files[i] instanceof RestrictedFile) {
      ((RestrictedFile)files[i]).lock();
    }
  }
}
```

Rules of Narrowing

- Narrowing let us restore a more specific point of view of an object using cast
- Casting a reference should be done only after verifying the object is indeed of the type we cast to!
- You cannot refer to an object through a `RestrictedFile` reference unless this object is indeed a **RestrictedFile** object!
- If we try to cast a `File` reference referring to a regular `File` object, into a `RestrictedFile`, a `ClassCastException` will be thrown

A More Complex Example: Geometric Shapes




```
class Shape {

   public final double PI=3.14159;
   protected String name;

   public String getName ()  {
        return (this.name);
   }  // getName
```

```
public int
     getSurfaceArea () {
       return (0);
   }  // area

} // Shape
```

A More Complex Example: Geometric Shapes

```
class Rectangle extends Shape {
   private int length, width;

   Rectangle () {
      this(0, 0); } // constructor

   Rectangle (int l, int w) {
      this( l, w, “rectangle”);
   } // constructor

   Rectangle (int l, int w,
              String n) {
      length = l; width = l;
      name = n; } // constructor

   public int getSurfaceArea () {
      return (length * width);
     } // area
```

```
 public String getName () {
  if (length == width)
     return ("square");
  else
     return (super.getName());
   }  // getName

 public String toString () {
   String s;
   s = new String ("A " +
      getName() +
      " with length " + length
      + " and width " + width);
   return (s);
  } // toString

} // Rectangle
```

Polymorphism is Possible Because of

- Inheritance: subclasses inherit attributes and method of the superclass.

```
public class Circle extends Shape {
  … …
}
```

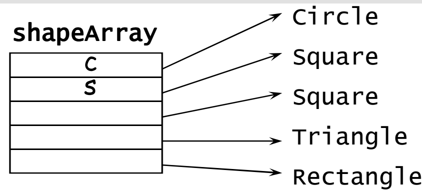
- Method overriding: subclasses can redefine methods that are inherited from the superclass

```
public class Shape {
  public float getSurfaceArea( ) {return 0.0f;}
  … …
}
public class Circle extends Shape {
  public float getSurfaceArea( ) {return (float)
  3.14f*radius*radius; }
  … …
}
```

Polymorphism is Possible Because of

- Polymorphic variables: variables that can hold value of different types (classes)

```
Circle c = new Circle(“Circle C”);
Square s = new Square(“Square S”);
Shape shapeArray[ ] = {c, s};
```

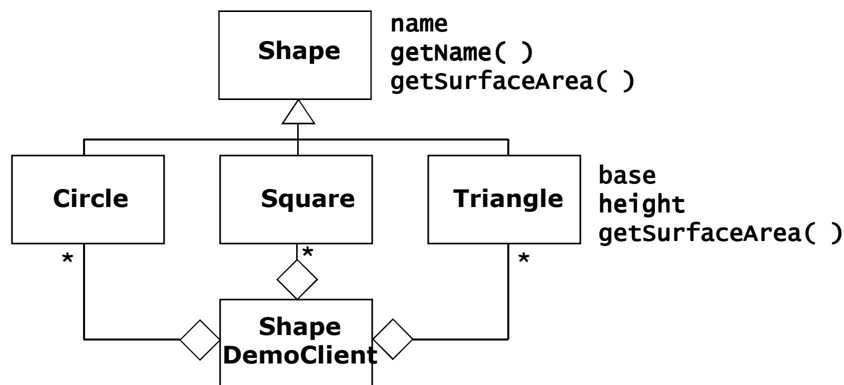


- Dynamic binding: method invocations are bound to methods during execution time

```
for(int i = 0; i < shapeArray.lenth; i++)
    shapeArray[i].getSurfaceArea( ) ;
```

Impact of polymorphism on Software Development

- Incremental development
  - adding new class is made easy with inheritance and polymorphism



Impact of polymorphism on Software Development

```
public class Triangle extends Shape {
  private float base, height;
  public Triangle(String aName) {super(aName); base = 1.0f;
                                 height = 1.0f; }
  public Triangle(String aName, float base, float height)
    { super(aName); this.base = base; this.height = height; }
  public float getSurfaceArea( ) {return (float) 0.5f*base*height;}
} // End Triangle class
```

```
public class ShapeDemoClient {
  public static void main(String argv[ ]) {
    … …
    Triangle t = new Triangle("Triangle T", 4.0f, 5.0f);
    Shape shapeArray[ ] = {c1, c2, s1, s2, t};
    … …
    }
  } // End main
} // End ShapeDemoClient class
```

Impact of polymorphism on Software Development

- Increased code readability
  - polymorphism also increases code readability since the same message is used to call different objects to perform the appropriate behavior

```
for (i = 0; i < numShapes; i++)
  switch (shapeType[i]) {
    'c': calculateCircleArea(circles[c++] ); break;
    's': calculateSquareArea(squares[s++] ); break;
}
```

versus

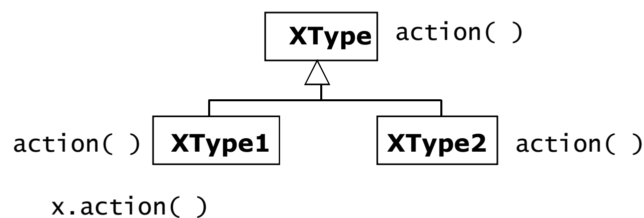
```
for(int i = 0; i < shapeArray.lenth; i++)
  shapeArray[i].calculateArea( );
```

Polymorphism Design Hints

- use polymorphism, not type information whenever you find the code of the form

```
if (x is of type 1)
  action1(x);
else if (x is of type 2)
  action2(x);
```

think polymorphism!



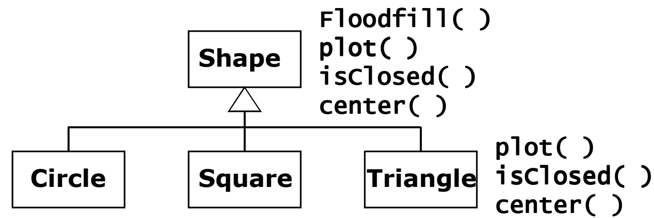
Polymorphism Design Hints

- move common behavior to the superclass
  - ◆to flood fill a shape means to do the following:
    - plot the outline of the shape
    - if it isn't a closed shape, give up
    - find an interior point of the shape
    - fill the shape
  - ◆these common behaviors can be put into the superclass `Shape`:

```
class Shape {
  ...
  public boolean
        floodfill(GraphicsPanel aPanel, Color aColor) {
    plot(aPanel);
    if ( ! isClosed( ) ) return false;
    Point aPoint = center( );
    aPanel.fill(aPoint.getX( ), aPoint.getY( ), aColor);
    return true;
  }
}
```

Polymorphism Design Hints

- Subclasses merely need to redefine: **plot( )**, **isClosed( )**, and **center( )**




```
Circle c = new Circle("Circle C");
Square s = new Square("Square S");
Triangle t = new Triangle("Triangle T");
Shape shapeArray[ ] = {c, s, t}
… …
for (int i = 0; i < shapeArray.length; i++)
  shapeArray[i].floodfill(aPanel, aColor);
...
```


Java Heterogeneous Collections of Objects

The Class `Vector`

- One of the most useful classes in the Java API is **`java.util.Vector`**
  - An array that dynamically resizes itself to whatever size is needed
  - You may add or remove objects from the vector at any position and its size grows and shrinks as needed to accommodate adding and removing items
- `Vector` is a class
  - Must have an *object* of type `Vector` instantiated via **`new( )`** to get an instance of `Vector`
- A `Vector` is designed to store `Object` references
  - Note that, because all classes inherit from the `Object` class, an `Object` reference can refer to any type of object
- All rules of good *OO* programming apply
  - Thus, access by *requesting services via methods*, not via *direct access* (such an array)

The Class `Vector` Contract

```
class Vector {
   public void     addElement( Object obj ) // adds to end
   public boolean  contains(Object elem)
   public Object   elementAt(int index) // returns reference to
                                        element at specified index
   public Object   firstElement()
   public int      indexOf(Object elem)
   public void     insertElementAt(Object obj, int index)
                                       // insertion into linked list
   public boolean  isEmpty()
   public Object   lastElement()
   public int      lastIndexOf(Object elem)
   public void     removeAllElements()
   public boolean  removeElement(Object obj)
   public void     removeElementAt(int index)
   public void     setElementAt(Object obj, int index)
   public int      size() // returns current number of elements
}
```

Java Vectors

- Can be populated only with *objects* and not with *primitives*
- Can be populated with objects that *contain* primitives
  - If you need to populate them with primitives, use *type wrapper* classes e.g., `Integer` for `int`, etc.
- Will allow you to populate them with *any* type of object . . .
  - Thus, good programming *requires* that the programmer enforce typing within a Vector, because Java *doesn't*
- Capacity is *dynamic*
  - Capacity *can* grow and shrink to fit needs
  - Capacity grows *upon demand*
- Capacity shrinks *when you tell it to do so* via method **`trimToSize( )`**
  - It implies performance costs upon subsequent insertion
- When extra capacity is needed, then it grows by *how much*?
  - Depends on which of three constructors is used . . .

Java Vectors Capacity

- Three Vector constructors:
  - `public Vector (int initialCapacity, int capacityIncrements);`
  - `public Vector (int initialCapacity);`
  - `public Vector( );`
- First constructor (2 parameters):
  - begins with **`initialCapacity`**
  - if/when it needs to grow, it grows by size `capacityIncrements`
- Second constructor (1 parameter):
  - begins with **`initialCapacity`**
  - if/when needs to grow, it grows by doubling current size
- Third constructor (no parameters):
  - begins with capacity of 10
  - if/when needs to grow, it grows by doubling current size

CSD Univ. of Crete

Fall 2012

Vectors Example: Cats & Dogs…

```
class Cat {
    public String toString() {
        return new String("meaw");
    }
}
```

```
class Dog {
    public String toString() {
        return new String("bark");
    }
}
```

```
class Mouse {
    public String toString() {
        return new String("squeak");
    }
}
```

CSD Univ. of Crete

Fall 2012

Vectors Example: Cats & Dogs…

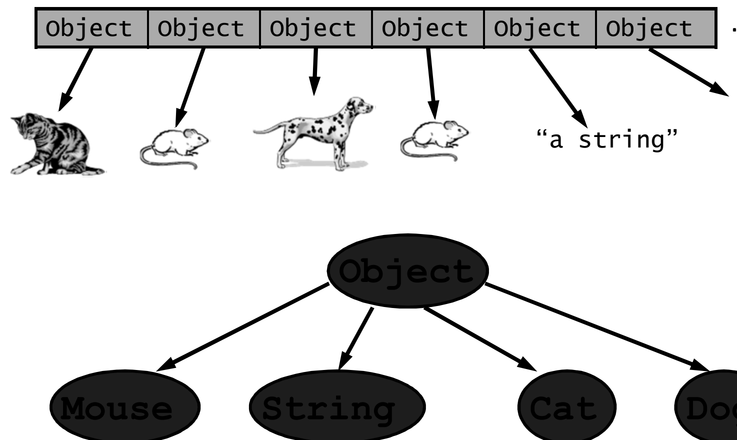
"its raining"

```
class MouseTrap {
   public static void main(String[] args) {
       Vector v = new Vector();
       v.addElement(new Cat());
       v.addElement(new Mouse());
       v.addElement(new Dog());
       v.addElement(new Mouse());
       v.addElement(new String("its raining"));
       for (int i = 0; i < v.size(); i++)
          System.out.println(v.elementAt(i));
       catchTheMice(v);
   }
```

CSD Univ. of Crete

Fall 2012

The `Vector` Holds `Object` References



113

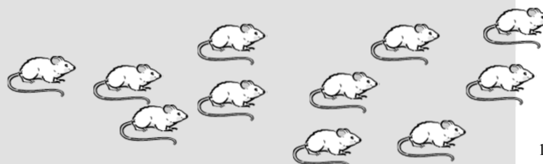
CSD Univ. of Crete

Fall 2012

Vectors and Casting

- Vectors are a subclass of class **`Object`**
  - Thus, vectors can handle *any* class of object (i.e., no *type checking*)
- Thus, *must* cast *any* object obtained from a **`Vector`** *before* invoking any methods not defined in class **`Object`**

```
private static catchTheMice(Vector v) {
   int i = 0;
   while (i < v.size()) {
       if (v.elementAt(i) instanceof Mouse) {
           v.removeElementAt(i);
       } else {
           i++;
       }
   }
}
```



114

Vectors versus Arrays

- Arrays: *statically* sized
- Vectors: *dynamically* sized
- Arrays: can directly access, e.g., `myArray[6]`
  - but *shouldn't*
    (except *maybe* within the class in which they're declared *if* efficiency concerns; *or* for testing purposes.)
- Vectors: *must use* methods to access
- Vector services provide a *good model* for the Array services you *should* implement

Vectors versus Linked Lists

- Can use Vectors to simulate a Linked List:
  - Don't want direct access to data, so . . .
  - Provide methods for `getPrevious( )`,`getNext( )`, etc. that do the standard Linked List things
  - While the list is implemented as a Vector, the client uses it *as if* it's a Linked List
- *BUT . . .*
  - There are performance implications (that may or may not matter for a given instance)
  - What is the cost of:
    - insertion?
    - deletion?

58

Vectors versus Linked Lists

- For ordered Linked Lists:
  - cost of traversal to locate target: *O(N)*
  - cost of insert/delete: *O(1)*
  - total cost: *O(N)*
- For ordered Vector:
  - cost of traversal to locate target: *O(N)* (if accessible via direct access, then *O(1)*)
  - insertion or deletion of element implies (average case), moving *O(N)* elements
  - total cost: *O(N)*
- Thus, at first glance, equivalent…
- *But what does* Big Oh hide here?
  - Linked Lists: search thru *N/2,* plus insert/delete
  - Vectors: search thru *N/2,* plus moving *N/2*

Thus, Vectors imply *twice* the work

Java vs. C++ Object Oriented Programming

- Similarities
  - User-defined classes can be used the same way as build-in types
  - Basic Syntax
- Differences
  - Methods (i.e., member functions) are the only function type
  - Object is the topmost ancestor for all classes
  - All methods use run-time and not compile-time types of objects (i.e., Java methods are like C++ virtual functions)
  - The type of all objects are known at run-time
  - It is always safe to return objects from methods
  - Single inheritance only