

Advanced Object-Oriented Programming

Inheritance, Interfaces and Polymorphism

Kulwadee Somboonviwat

International College, KMITL

kskulwad@kmitl.ac.th

The software crisis

- **software engineering:** The practice of developing, designing, documenting, testing large computer programs.
- Large-scale projects face many issues:
 - getting many programmers to work together
 - getting code finished on time
 - avoiding redundant code
 - finding and fixing bugs
 - maintaining, improving, and reusing existing code
- **code reuse:** The practice of writing program code once and using it in many contexts

I. Inheritance

Inheritance: facility for code reuse

- Syntax
- Overriding
- Inheritance Hierarchy
- Polymorphism
- Interacting with super class
- Inheritance and constructor
- Inheritance and fields
- The Object class
- Abstract class

Case Study: Employee regulations

- Consider the following employee regulations:
 - Employees work 40 hours / week.
 - Employees make \$40,000 per year, except legal secretaries who make \$5,000 extra per year (\$45,000 total), and marketers who make \$10,000 extra per year (\$50,000 total).
 - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
 - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.
- Each type of employee has some unique behavior:
 - Lawyers know how to sue.
 - Marketers know how to advertise.
 - Secretaries know how to take dictation.
 - Legal secretaries know how to prepare legal documents.

An Employee class

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;          // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";    // use the yellow form
    }
}
```

- Exercise: Implement class `Secretary`, based on the previous employee regulations. (Secretaries can take dictation.)

Redundant Secretary class

```
// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;          // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";    // use the yellow form
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`.

- We'd like to be able to say:

```
// A class to represent secretaries.
```

```
public class Secretary {
```

```
    copy all the contents from the Employee class;
```

```
    public void takeDictation(String text) {
```

```
        System.out.println("Taking dictation of text: " + text);
```

```
    }
```

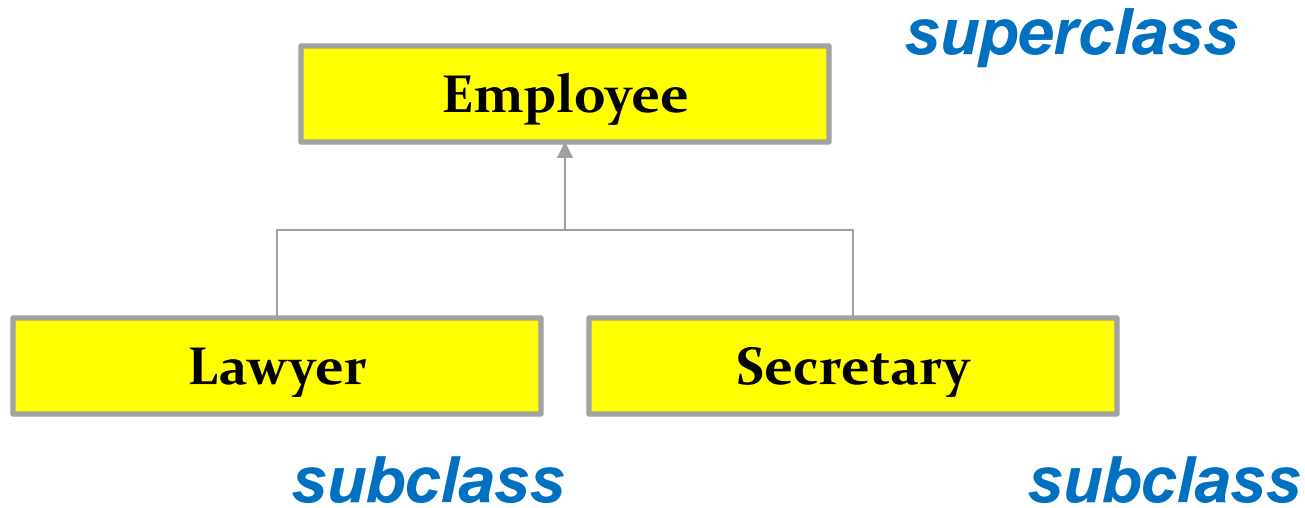
```
}
```


Inheritance

- **inheritance:** A way to form new classes based on existing classes, taking on their attributes/behavior.
 - a way to group related classes
 - a way to share code between two or more classes
- One class can *extend* another, absorbing its data/behavior.
 - **superclass:** The parent class that is being extended.
 - **subclass:** The child class that extends the superclass and inherits its behavior.
 - Subclass gets a copy of every field and method from superclass

Inheritance in Java

- Defined with the **extends** keyword
- Unlike C++, Java does not allow **multiple inheritance**
 - In Java, we use interface to implement multiple inheritance
- Why using inheritance ?
 - Model “is-a” relationship
 - Code Reuse



```
class Employee {  
...  
}
```

```
class Lawyer extends Employee {  
...  
}
```

```
class Secretary extends Employee {  
....  
}
```

Inheritance Syntax

```
public class name extends superclass {  
    ....  
}
```

– Example:

```
public class Secretary extends Employee {  
    ...  
}
```

- By extending Employee, each Secretary object now:
 - receives a getHours, getSalary, getVacationDays, and getVacationForm method automatically
 - **can be treated as an Employee** by client code (see later in polymorphism)

Improved Secretary code

```
// A class to represent secretaries.  
public class Secretary extends Employee {  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Now we only write the parts unique to each type.
 - Secretary **inherits** getHours, getSalary, getVacationDays, **and** getVacationForm **methods** from Employee.
 - Secretary **adds** the takeDictation method.

Overriding

Implementing Lawyer

- Consider the following lawyer regulations:
 - Lawyers who get an extra week of paid vacation (a total of 3).
 - Lawyers use a pink form when applying for vacation leave.
 - Lawyers have some unique behavior: they know how to sue.
- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.

Consider the Employee class

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```


Overriding methods

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
 - No special syntax required to override a superclass method.
Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {  
    // overrides getVacationForm method in Employee  
    class  
        public String getVacationForm() {  
            return "pink";  
        }  
        ...  
}
```

- Exercise: Complete the `Lawyer` class.
 - (3 weeks vacation, pink vacation form, can sue)

Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    @Override
    public int getVacationDays() {
        return 15;           // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Complete the `Marketer` class. Marketers make \$10,000 extra (\$50,000 total) and know how to advertise.

Marketer class

// A class to represent marketers.

```
public class Marketer extends Employee {  
    public void advertise() {  
        System.out.println("Act now while supplies last!");  
    }  
  
    public double getSalary() {  
        return 50000.0;           // $50,000.00 / year  
    }  
}
```

Note: Overloading vs. Overriding

- **Overloading** means to define multiple methods with the same name but different signatures
- **Overriding** means to provide a new implementation for a method (already defined in the superclass) in the subclass

Which code uses overriding (a or b) ?

(a)

```
public class Test1
{
    public static void main(String[] args)
    {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}
class B
{
    public void p(double i)
    {
        System.out.println( i * 2 );
    }
}
class A extends B
{
    public void p(double i)
    {
        System.out.println(i);
    }
}
```

(b)

```
public class Test2
{
    public static void main(String[] args)
    {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}
class B
{
    public void p(double i)
    {
        System.out.println( i * 2 );
    }
}
class A extends B
{
    public void p(int i)
    {
        System.out.println(i);
    }
}
```

Overriding vs Hiding

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

The distinction between hiding and overriding has important implications. The version of the overridden method that gets invoked is the one in the subclass. The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass. Let's look at an example that contains two classes.

```
public class Animal {  
    public static void testClassMethod() {  
        System.out.println("The class" + " method in Animal.");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance " + " method in Animal.");  
    }  
}
```

```
public class Cat extends Animal {  
    public static void testClassMethod() {  
        System.out.println("The class method" + " in Cat.");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method" + " in Cat.");  
    }  
}
```

```
public static void main(String[] args) {  
    Cat myCat = new Cat();  
    Animal myAnimal = myCat;  
    Animal.testClassMethod();  
    myAnimal.testInstanceMethod();  
}
```

The output from this program is as follows:

The class method in Animal.
The instance method in Cat.

Preventing Extending and Overriding

- Use the **final** modifier to indicate that
 - A class cannot be a superclass
 - A method cannot be overridden by its subclasses

```
public final class C {  
}
```

```
public class Test {  
    public final void m() { }  
}
```


Inheritance Hierarchy

Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
 - Example: A legal secretary is the same as a regular secretary but makes more money (\$45,000) and can file legal briefs.

```
public class LegalSecretary extends  
Secretary {  
    ...  
}
```

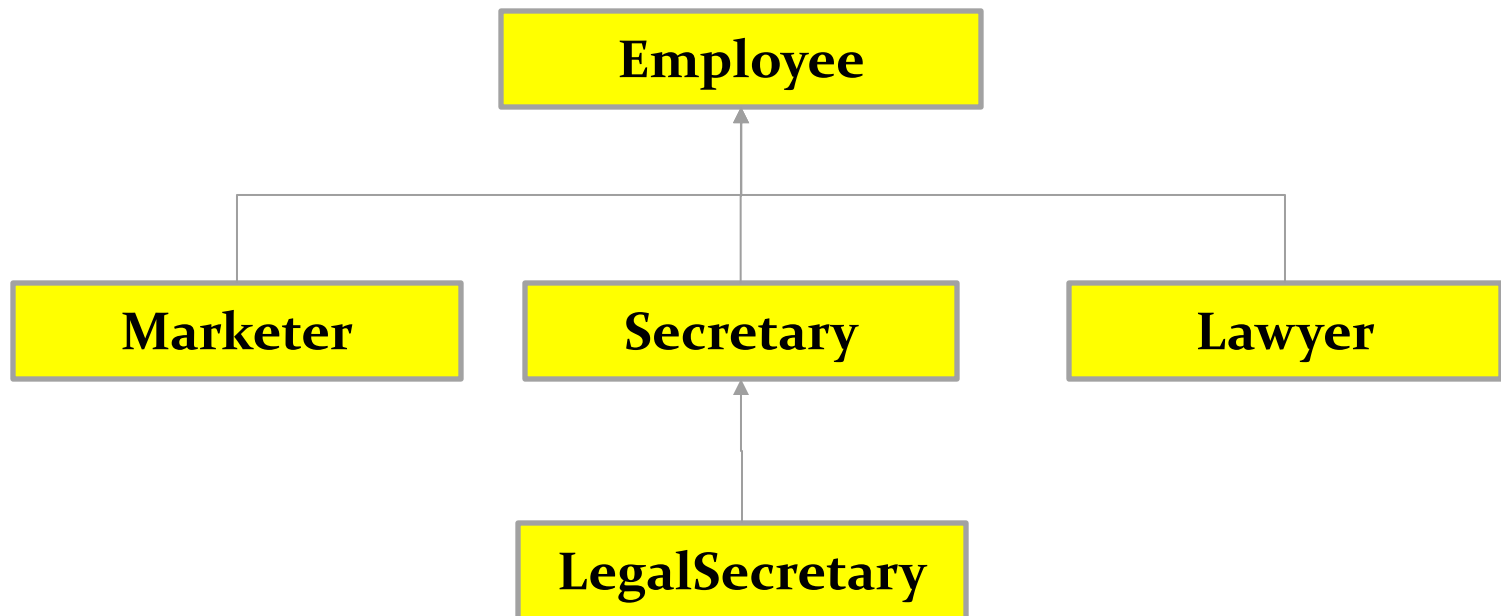
- Exercise: Complete the `LegalSecretary` class.

LegalSecretary class

```
// A class to represent legal secretaries.  
public class LegalSecretary extends Secretary {  
    public void fileLegalBriefs() {  
        System.out.println("I could file all day!");  
    }  
  
    public double getSalary() {  
        return 45000.0;           // $45,000.00 / year  
    }  
}
```

Inheritance Hierarchy

- Inheritance needs not stop at deriving one layer of class
- The collection of all classes extending from a common superclass is called an **inheritance hierarchy**



Interacting with the superclass

Changes to common behavior

- Let's return to our previous company/employee example.
- Imagine a company-wide change affecting all employees.

Example: Everyone is given a \$10,000 raise due to inflation.

- The base employee salary is now \$50,000.
 - Legal secretaries now make \$55,000.
 - Marketers now make \$60,000.
- We must modify our code to reflect this policy change.

Modifying the superclass

```
// A class to represent employees (20-page manual).
public class Employee {
    public int getHours() {
        return 40;                // works 40 hours / week
    }

    public double getSalary() {
        return 50000.0;           // $50,000.00 / year
    }

    ...
}
```

- Are we finished?
- The `Employee` subclasses are still incorrect.
 - They have overridden `getSalary` to return other values.

An unsatisfactory solution

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        return 55000.0;  
    }  
    ...  
}  
  
public class Marketer extends Employee {  
    public double getSalary() {  
        return 60000.0;  
    }  
    ...  
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.

Calling overridden methods

- Subclasses can call overridden methods with `super`

`super.method(parameters)`

– Example:

```
public class LegalSecretary extends Secretary {  
    public double getSalary() {  
        double baseSalary = super.getSalary();  
        return baseSalary + 5000.0;  
    }  
    ...  
}
```

- Exercise: Modify Lawyer and Marketer to use `super`.

Improved subclasses

```
public class Lawyer extends Employee {  
    public String getVacationForm() {  
        return "pink";  
    }  
  
    public int getVacationDays() {  
        return super.getVacationDays() + 5;  
    }  
  
    public void sue() {  
        System.out.println("I'll see you in court!");  
    }  
}
```

```
public class Marketer extends Employee {  
    public void advertise() {  
        System.out.println("Act now while supplies  
last!");  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 10000.0;  
    }  
}
```

Summary: The super keyword

- Constructors of a superclass are not inherited into the subclass, but can be invoked only from the constructors of the subclasses using the keyword ***super***
- The keyword “**super**” refers to the **superclass** of the class in which it appears
 - Used to call a superclass constructor : **super()**
 - Used to call a superclass method :
super.methodName(parameters)

Inheritance and Constructor

Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company.
 - For each year worked, we'll award 2 additional vacation days.
 - When an Employee object is constructed, we'll pass in the number of years the person has been with the company.
 - This will require us to modify our `Employee` class and add some new state and behavior.
 - Exercise: Make necessary modifications to the `Employee` class.

Modified Employee class

```
public class Employee {  
    private int years;  
  
    public Employee(int initialYears) {  
        years = initialYears;  
    }  
  
    public int getHours() {  
        return 40;  
    }  
  
    public double getSalary() {  
        return 50000.0;  
    }  
  
    public int getVacationDays() {  
        return 10 + 2 * years;  
    }  
  
    public String getVacationForm() {  
        return "yellow";  
    }  
}
```

Problem with constructors

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol   : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
      ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
- The long explanation: (next slide)

The detailed explanation

- Constructors are not inherited.
 - Subclasses don't inherit the Employee(int) constructor.
 - Subclasses receive a default constructor that contains:

```
public Lawyer() {  
    super();      // calls Employee() constructor  
}
```
- But our Employee(int) replaces the default Employee().
 - The subclasses' default constructors are now trying to call a non-existent default Employee constructor.

Calling superclass constructor

```
super (parameters) ;
```

– Example:

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years); // calls Employee constructor  
    }  
    ...  
}
```

- The `super` call must be the first statement in the constructor.
- Exercise: Make a similar modification to the `Marketer` class.

Modified Marketer class

// A class to represent marketers.

```
public class Marketer extends Employee {  
    public Marketer(int years) {  
        super(years);  
    }  
  
    public void advertise() {  
        System.out.println("Act now while supplies last!");  
    }  
  
    public double getSalary() {  
        return super.getSalary() + 10000.0;  
    }  
}
```

- Exercise: Modify the Secretary subclass.
 - Secretaries' years of employment are not tracked.
 - They do not earn extra vacation for years worked.

Modified Secretary class

// A class to represent secretaries.

```
public class Secretary extends Employee {  
    public Secretary() {  
        super(0);  
    }  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Since Secretary doesn't require any parameters to its constructor, LegalSecretary compiles without a constructor.
 - Its default constructor calls the Secretary() constructor.

Inheritance and fields

Inheritance and fields

- Try to give lawyers \$5000 for each year at the company:

```
public class Lawyer extends Employee {  
    ...  
    public double getSalary() {  
        return super.getSalary() + 5000 * years;  
    }  
    ...  
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private access in Employee  
    return super.getSalary() + 5000 * years;  
                                   ^
```

- Private fields cannot be directly accessed from subclasses.
 - One reason: So that subclassing can't break encapsulation.
 - How can we get around this limitation?

Improved Employee code

Add an accessor for any field needed by the subclass.

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getYears() {
        return years;
    }
    ...
}

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * getYears();
    }
    ...
}
```

Revisiting Secretary

- The `Secretary` class currently has a poor solution.
 - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
 - If we call `getYears` on a `Secretary` object, we'll always get 0.
 - This isn't a good solution; what if we wanted to give some other reward to *all* employees based on years of service?
- Redesign our `Employee` class to allow for a better solution.

Improved Employee code

- Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```
public class Employee {  
    private int years;  
  
    public Employee(int initialYears) {  
        years = initialYears;  
    }  
  
    public int getVacationDays() {  
        return 10 + getSeniorityBonus();  
    }  
  
    // vacation days given for each year in the company  
    public int getSeniorityBonus() {  
        return 2 * years;  
    }  
    ...  
}
```

- How does this help us improve the Secretary?

Improved Secretary code

- Secretary can selectively override `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version.
 - Choosing a method at runtime is called *dynamic binding*.

```
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }

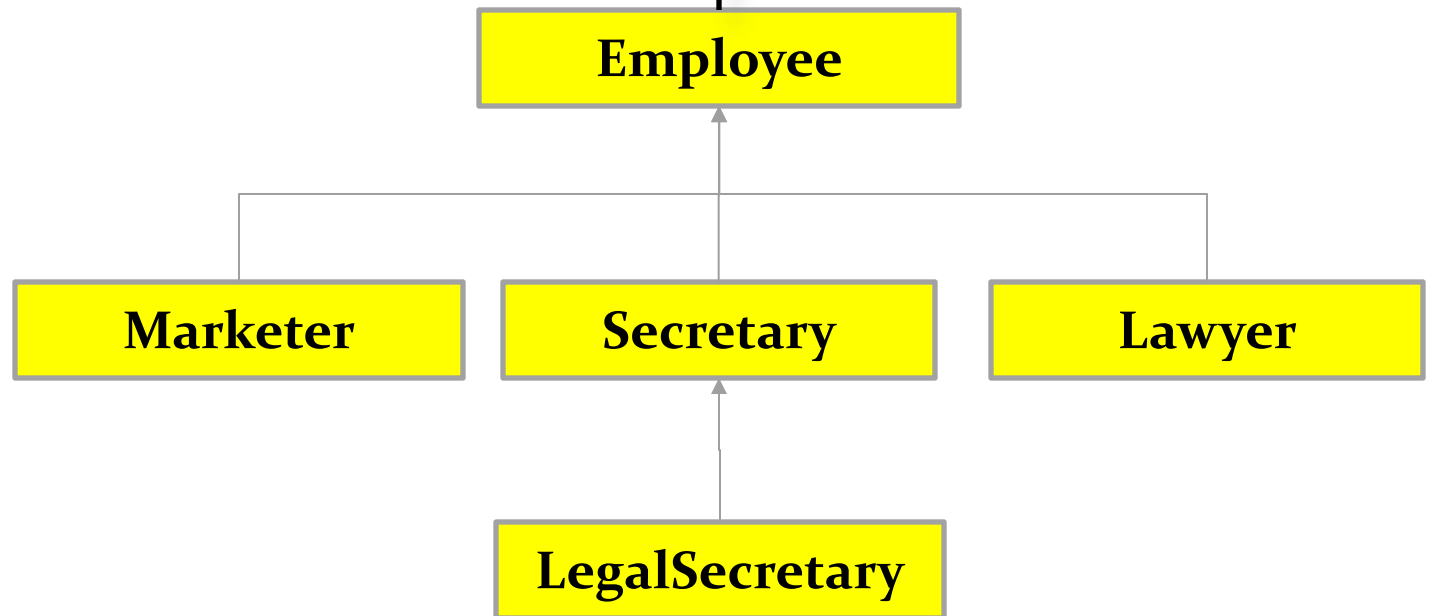
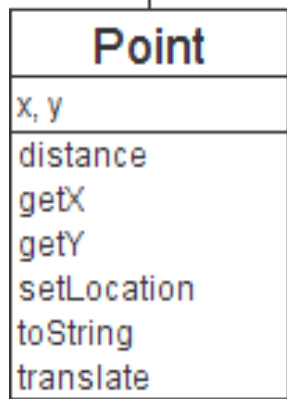
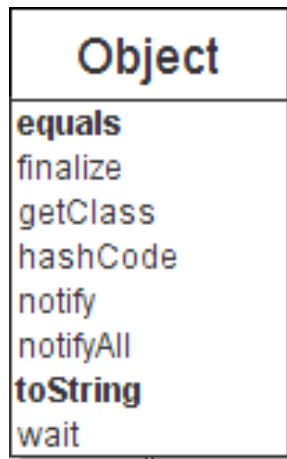
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

The Object class

The Object Class

- Every class in Java is descended from the **java.lang.Object** class
- Therefore it is important to be familiar with the methods available in the Object class so that you can use them in your classes
 - Study the Object class's methods from the Java API documentation

<http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html>



Class Object [\[source \]](#)

java.lang.Object

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

[Object\(\)](#)

Method Summary

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 <?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the 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.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the 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 the 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.

Object variables

- You can store any object in a variable of type `Object`.

```
Object o1 = new Point(5, -3);  
Object o2 = "hello there";  
Object o3 = new Scanner(System.in);
```

- An `Object` variable only knows how to do general things.

```
String s = o1.toString();           // ok  
int len = o2.length();              // error  
String line = o3.nextLine();        // error
```

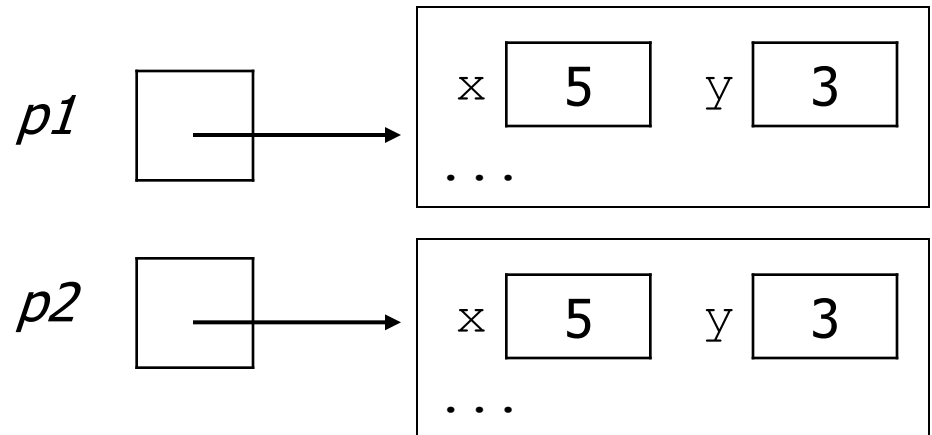
- You can write methods that accept an `Object` parameter.

```
public void checkForNull(Object o) {  
    if (o == null) {  
        throw new IllegalArgumentException();  
    }  
}
```

Comparing objects

- The `==` operator does not work well with objects.
 `==` compares references to objects, not their state.
 It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
if (p1 == p2) {    // false  
    System.out.println("equal");  
}
```



The equals method

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {  
    System.out.println("the strings are  
equal");  
}
```

- But if you write a class, its `equals` method behaves like `==`

```
if (p1.equals(p2)) {    // false :- (  
    System.out.println("equal");  
}
```

- This is the behavior we inherit from class `Object`.
- Java doesn't understand how to compare `Points` by default.

equals and Object

```
public boolean equals(Object name) {  
    statement(s) that return a boolean value ;  
}
```

- The parameter to `equals` must be of type `Object`.
- `Object` is a general type that can match any object.
- Having an `Object` parameter means *any* object can be passed.
 - If we don't know what type it is, how can we compare it?

- Another flawed equals implementation:

```
public boolean equals(Object o) {  
    return x == o.x && y == o.y;  
}
```

- It does not compile:

```
Point.java:36: cannot find symbol  
symbol   : variable x  
location: class java.lang.Object  
return x == o.x && y == o.y;  
           ^
```

- The compiler is saying,
"o could be any object. Not every object has an x field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a `Point`.

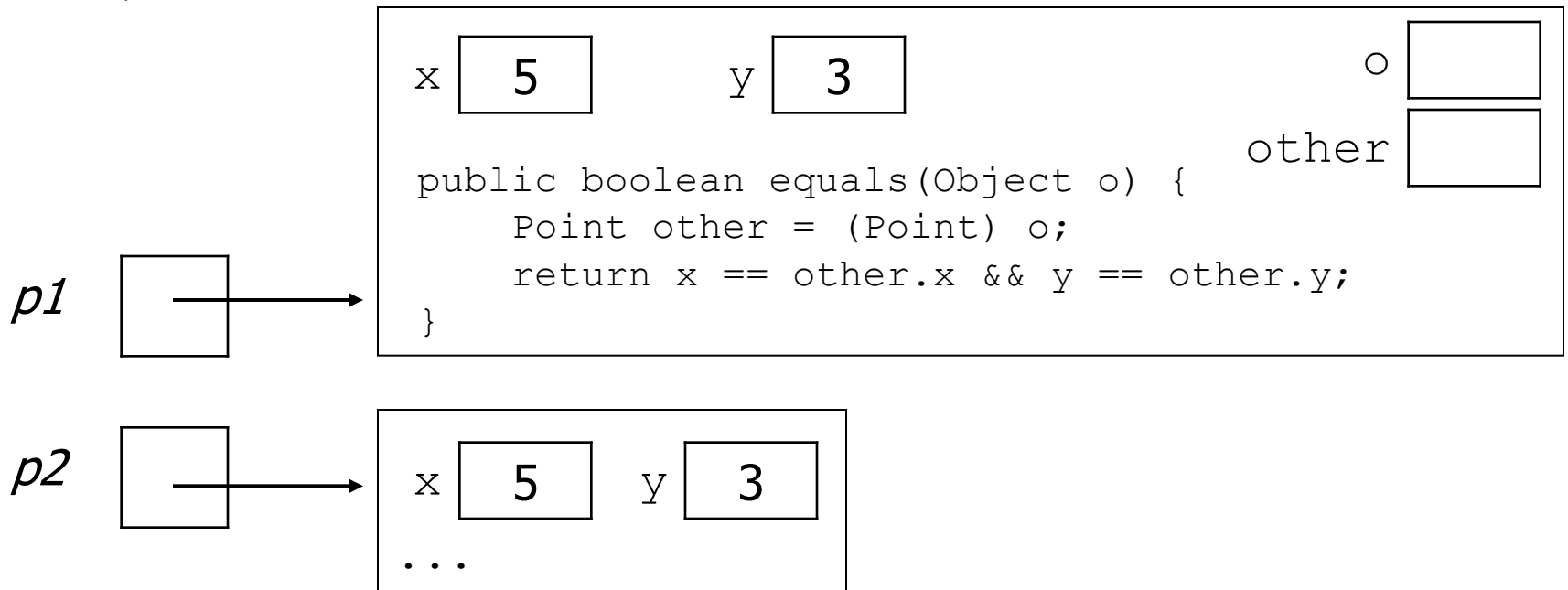
```
public boolean equals(Object o) {  
    Point other = (Point) o;  
    return x == other.x && y == other.y;  
}
```

- Casting objects is different than casting primitives.
 - Really casting an `Object` reference into a `Point` reference.
 - Doesn't actually change the object that was passed.
 - Tells the compiler to *assume* that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
if (p1.equals(p2)) {  
    System.out.println("equal");  
}
```



Comparing different types

```
Point p = new Point(7, 2);  
if (p.equals("hello")) {    // should be false  
    ...  
}
```

- Currently our method crashes on the above code:

```
Exception in thread "main"  
java.lang.ClassCastException: java.lang.String  
    at Point.equals(Point.java:25)  
    at PointMain.main(PointMain.java:25)
```

- The culprit is the line with the type-cast:

```
public boolean equals(Object o) {  
    Point other = (Point) o;
```

The instanceof keyword

```
if (variable instanceof type) {  
    statement(s);  
}
```

- Asks if a variable refers to an object of a given type.
 - Used as a boolean test.

```
String s = "hello";  
Point p = new Point();
```

expression	result
s instanceof Point	false
s instanceof String	true
p instanceof Point	true
p instanceof String	false
p instanceof Object	true
s instanceof Object	true
null instanceof String	false
null instanceof Object	false

Final equals method

```
// Returns whether o refers to a Point object with  
// the same (x, y) coordinates as this Point.
```

```
public boolean equals(Object o) {  
  
    if (o instanceof Point) {  
        // o is a Point; cast and compare it  
        Point other = (Point) o;  
        return x == other.x && y == other.y;  
  
    } else {  
        // o is not a Point; cannot be equal  
        return false;  
    }  
}
```

Abstract Class

Abstract Classes

- As you move up the inheritance hierarchy, classes become more general and probably more abstract.
- At some point, the ancestor class becomes *so general that you think of it* more as a basis for other classes than as a class with specific instances you want to use.

Abstract Class

superclass

GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

sign
indicates
protected
modifiers

#GeometricObject()
#GeometricObject(color: String, filled:
boolean)
+getColor() : String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double

Abstract method are
italized

Circle

-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled:
boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

subclass

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double,
color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

subclass

```
public abstract class GeometricObject {  
    private String color = "white";  
    private boolean filled;  
    private java.util.Date dateCreated;  
    /** Construct a default geometric object */  
    protected GeometricObject() {  
        dateCreated = new java.util.Date();  
    }  
    /** Construct a geometric object with the specified color  
    * and filled value */  
    protected GeometricObject(String color, boolean filled) {  
        this.color = color;  
        this.filled = filled;  
    }  
    /** Return color */  
    public String getColor() {    return color;    }  
    public void setColor(String color) {    this.color = color;    }  
    public boolean isFilled() {    return filled;    }  
    public void setFilled(boolean filled) {    this.filled = filled;    }  
    public java.util.Date getDateCreated() { return dateCreated;    }  
    public String toString() {  
        return "created on " + dateCreated + "\n" + "color: " + color +  
        " and filled: " + filled;  
    }  
    /** Abstract method getArea */  
    public abstract double getArea();  
  
    /** Abstract method getPerimeter */  
    public abstract double getPerimeter();  
}
```

```
public class Circle extends GeometricObject {
    private double radius;
    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }
    public Circle(double radius, String color, boolean filled) {
        this.radius = radius;
        setColor(color);
        setFilled(filled);
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getDiameter() {
        return 2 * radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() +
            " and the radius is " + radius);
    }
}
```

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() { }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(double width, double height, String color, boolean filled) {
        this(width, height);
        setColor(color);
        setFilled(filled);
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double getArea() {
        return width * height;
    }
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

```
public class TestGeometricObject
{
    public static void main(String[] args)
    {
        // create two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);

        System.out.println("The two objects have the same area? " +
            equalArea(geoObject1, geoObject2) );

        displayGeometricObject(geoObject1);
        displayGeometricObject(geoObject2);
    }

    public static boolean equalArea(GeometricObject object1,
        GeometricObject object2)
    {
        return object1.getArea() == object2.getArea();
    }

    public static void displayGeometricObject(GeometricObject object)
    {
        System.out.println();
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }
}
```

Interesting Points on Abstract Classes

- An abstract method cannot be contained in a non-abstract class
- An abstract class cannot be instantiated using the **new** operator
- A class that contains abstract method must be abstract
- An abstract class can be used as a data type

```
GeometricObject[] objects = new GeometricObject[5];  
objects[0] = new Circle();  
objects[1] = new Rectangle();  
objects[2] = new Circle();
```

II. Polymorphism

Polymorphism

- Types of Polymorphism
- Implementation mechanism
- Coding with Polymorphism
 - Method parameters
 - Array
 - Common usage pattern
 - Casting Reference

Polymorphism

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
 - `System.out.println` can print any type of object.
 - Each one displays in its own way on the console.
 - `EmployeeMain` can interact with any type of employee.
 - Each one is paid in different way.

Type of Polymorphism

– Coercion (implicit type conversion)

E.g. **int** a = 10;
 double d1 = 9.0;
 double d2 = a / d1; // a is coerced to double

– Overloading

same operator symbol or method name can be used in different contexts.

E.g. + can be used to perform integer division, floating-point division, or string concatenation, depending on the types of its operands.

Type of Polymorphism

– Subtype

When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype's version of that operation executing.

** Java supports subtype polymorphism via inheritance and interfaces*

Type of Polymorphism

– Parametric

Within a class declaration, a field name can associate with different types and a method name can associate with different parameter and return types.

** Java supports parametric polymorphism via **generics***

Subtype Polymorphism

```
// File TestGeo
public class TestGeo
{
    public static void main(String[] args)
    {
        GeometricObject[] gobjs = new GeometricObject[] {new Circle(),
                                                            new Rectangle(), new GeometricObject()};

        for (GeometricObject g : gobjs)
            g.draw();
    }
}
```

```
// File: GeometricObject.java
public class GeometricObject
{
    void draw() { System.out.println("Geo"); }
}

class Circle extends GeometricObject
{
    void draw() { System.out.println("Circle"); }
}

class Rectangle extends GeometricObject
{
    void draw() { System.out.println("Rectangle"); }
}
```

Subtype Polymorphism is made possible by

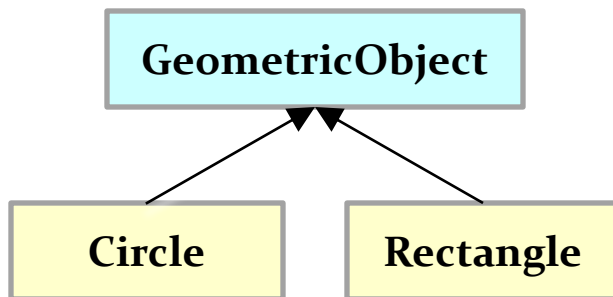
- Upcasting
- Method Overriding
- Dynamic Binding

*** Dynamic Binding :**

the method being called upon an object is looked up by name at runtime

Upcasting

- Inheritance enables a **subclass** to inherit features from its **superclass**
- A class defines a type
 - Superclass → supertype
 - Subclass → subtype



Every instance of a subclass is also an instance of its superclass, but not vice versa

E.g. every circle is a geometric object, but not every geometric object is a circle

Upcasting:

```
GeometricObject G = new Circle();    // OK
```

But,

```
Circle C = new GeometricObject(); // ERROR!
```

Coding with polymorphism

- A variable of type T can hold an object of any subclass of T .

```
Employee ed = new Lawyer();
```

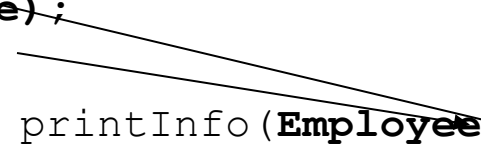
- You can call any methods from the `Employee` class on `ed`.
- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary());           // 50000.0  
System.out.println(ed.getVacationForm());    // pink
```


Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
public class EmployeeMain {  
    public static void main(String[] args) {  
        Lawyer lisa = new Lawyer();  
        Secretary steve = new Secretary();  
        printInfo(lisa);  
        printInfo(steve);  
    }  
  
    public static void printInfo(Employee empl) {  
        System.out.println("salary: " + empl.getSalary());  
        System.out.println("v.days: " +  
empl.getVacationDays());  
        System.out.println("v.form: " +  
empl.getVacationForm());  
        System.out.println();  
    }  
}
```

A diagram consisting of two lines. One line starts from the **printInfo(steve)** call in the `main` method and points to the `printInfo` method signature. The other line starts from the `printInfo(Employee empl)` signature and points to the `Employee` parameter, illustrating that the `Secretary` object `steve` is passed to a method that expects an `Employee`.

OUTPUT:

```
salary: 50000.0  
v.days: 15  
v.form: pink
```

```
salary: 50000.0  
v.days: 10  
v.form: yellow
```

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {  
    public static void main(String[] args) {  
        Employee[] e = { new Lawyer(),    new Secretary(),  
                        new Marketer(), new LegalSecretary() };  
  
        for (int i = 0; i < e.length; i++) {  
            System.out.println("salary: " + e[i].getSalary());  
            System.out.println("v.days: " +  
                               e[i].getVacationDays());  
            System.out.println();  
        }  
    }  
}
```

Output:

salary: 50000.0

v.days: **15**

salary: 50000.0

v.days: 10

salary: **60000.0**

v.days: 10

salary: **55000.0**

v.days: 10

Polymorphism problems

- 4-5 classes with inheritance relationships are shown.
- A client program calls methods on objects of each class.
- You must read the code and determine the client's output.

A polymorphism problem

- Suppose that the following four classes have been declared:

```
public class Foo {  
    public void method1() {  
        System.out.println("foo 1");  
    }  
  
    public void method2() {  
        System.out.println("foo 2");  
    }  
  
    public String toString() {  
        return "foo";  
    }  
}  
  
public class Bar extends Foo {  
    public void method2() {  
        System.out.println("bar 2");  
    }  
}
```

```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

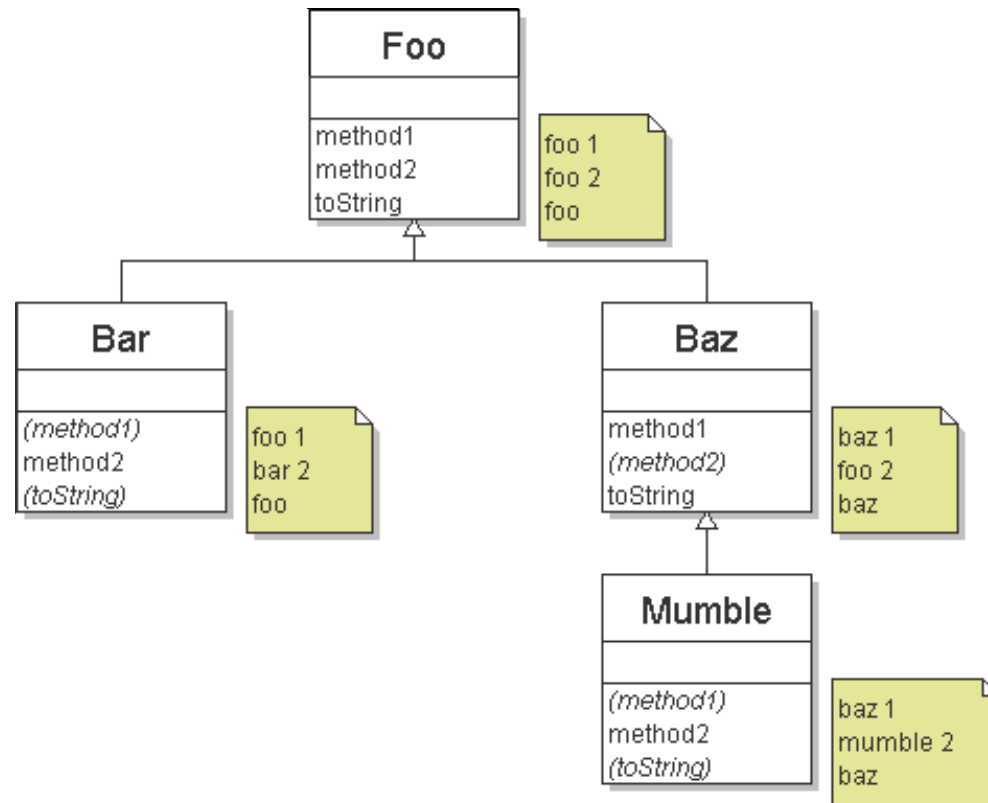
public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

- What would be the output of the following client code?

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new
    Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Diagramming the classes

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.



Finding output with tables

method	Foo	Bar	Baz	Mumble
method1	foo 1	<i>foo 1</i>	baz 1	<i>baz 1</i>
method2	foo 2	bar 2	<i>foo 2</i>	mumble 2
toString	foo	<i>foo</i>	baz	<i>baz</i>

Polymorphism answer

```
Foo[] pity = {new Baz(),new Bar(),new Mumble(),new Foo()};  
for (int i = 0; i < pity.length; i++) {  
    System.out.println(pity[i]);  
    pity[i].method1();  
    pity[i].method2();  
    System.out.println();  
}
```

- **Output:**

```
baz  
baz 1  
foo 2  
  
foo  
foo 1  
bar 2  
  
baz  
baz 1  
mumble 2  
  
foo  
foo 1  
foo 2
```


Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();  
int hours = ed.getHours();    // ok; it's in  
Employee  
ed.sue();                     // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue`.

- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;  
theRealEd.sue();           // ok
```

```
((Lawyer) ed).sue();       // shorter version
```

More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();  
((Secretary) eric).takeDictation("hi");           // ok  
((LegalSecretary) eric).fileLegalBriefs();       // exception  
// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();  
((Secretary) linda).takeDictation("hi");         // error
```

- Casting doesn't actually change the object's behavior.
It just gets the code to compile/run.

```
((Employee) linda).getVacationForm() // pink (Lawyer's)
```

III. Interfaces

Relatedness of types

Write a set of Circle, Rectangle, and Triangle classes.

- Certain operations that are common to all shapes.
 - perimeter - distance around the outside of the shape
 - area - amount of 2D space occupied by the shape
- Every shape has them but computes them differently.

Shape area, perimeter

- Rectangle (as defined by width w and height h):

$$\text{area} = w h$$

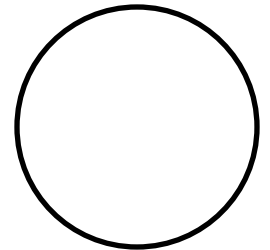
$$\text{perimeter} = 2w + 2h$$



- Circle (as defined by radius r):

$$\text{area} = \pi r^2$$

$$\text{perimeter} = 2\pi r$$

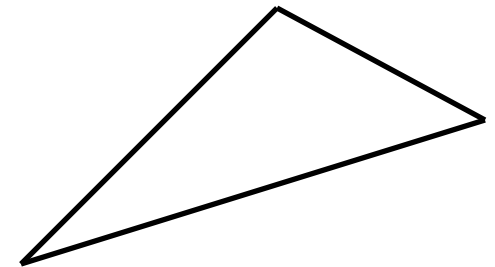


- Triangle (as defined by side lengths a , b , and c)

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

$$\text{where } s = \frac{1}{2}(a+b+c)$$

$$\text{perimeter} = a + b + c$$



Common behavior

- Write shape classes with methods `perimeter` and `area`.
- We'd like to be able to write client code that treats different kinds of shape objects in the same way, such as:
 - Write a method that prints any shape's area and perimeter.
 - Create an array of shapes that could hold a mixture of the various shape objects.
 - Write a method that could return a rectangle, a circle, a triangle, or any other shape we've written.
 - Make a `DrawingPanel` display many shapes on screen.

Interfaces

- **interface:** A list of methods that a class can implement.
 - Inheritance gives you an is-a relationship *and* code-sharing.
 - A `Lawyer` object can be treated as an `Employee`, and `Lawyer` inherits `Employee`'s code.
 - Interfaces give you an is-a relationship *without* code sharing.
 - A `Rectangle` object can be treated as a `Shape`.

Declaring an interface

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
}
```

Example:

```
public interface Vehicle {  
    public double speed();  
    public void setDirection(int direction);  
}
```

- **abstract method**: A header without an implementation.
 - The actual body is not specified, to allow/force different classes to implement the behavior in its own way.

Shape interface

```
public interface Shape {  
    public double area();  
    public double perimeter();  
}
```

- This interface describes the features common to all shapes.
(Every shape has an area and perimeter.)

Implementing an interface

```
public class name implements interface {  
    ...  
}
```

– Example:

```
public class Bicycle implements Vehicle  
{  
    ...  
}
```

- A class can declare that it *implements* an interface.
 - This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

Interface requirements

- If a class claims to be a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.

- Example:

```
public class Banana implements Shape {  
    ...  
}
```

- The compiler error message:

```
Banana.java:1: Banana is not abstract and  
does not override abstract method area() in  
Shape
```

```
public class Banana implements Shape {
```

^

Complete Circle class

```
// Represents circles.
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

Complete Rectangle class

```
// Represents rectangles.
public class Rectangle implements Shape {
    private double width;
    private double height;

    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Returns the area of this rectangle.
    public double area() {
        return width * height;
    }

    // Returns the perimeter of this rectangle.
    public double perimeter() {
        return 2.0 * (width + height);
    }
}
```

Complete Triangle class

// Represents triangles.

```
public class Triangle implements Shape {  
    private double a;  
    private double b;  
    private double c;
```

// Constructs a new Triangle given side lengths.

```
public Triangle(double a, double b, double c) {  
    this.a = a;  
    this.b = b;  
    this.c = c;  
}
```

// Returns this triangle's area using Heron's formula.

```
public double area() {  
    double s = (a + b + c) / 2.0;  
    return Math.sqrt(s * (s - a) * (s - b) * (s - c));  
}
```

// Returns the perimeter of this triangle.

```
public double perimeter() {  
    return a + b + c;  
}  
}
```

Interface & Polymorphism

Interfaces + polymorphism

- Interfaces don't benefit the class so much as the *client*.
 - Interface's is-a relationship lets the client use polymorphism.

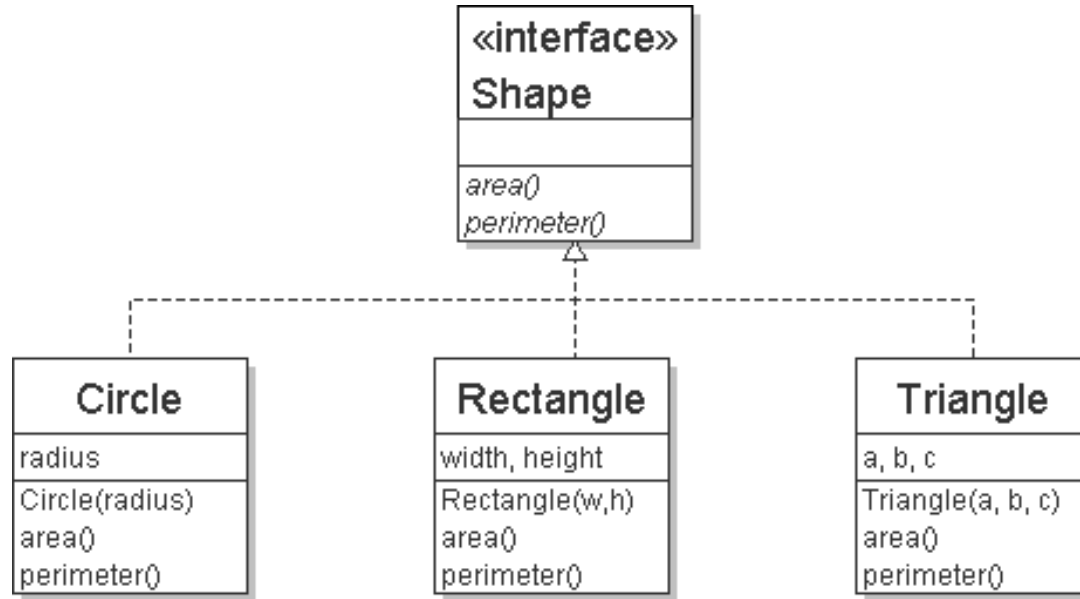
```
public static void printInfo(Shape s) {  
    System.out.println("The shape: " + s);  
    System.out.println("area : " + s.area());  
    System.out.println("perim: " + s.perimeter());  
}
```

- Any object that implements the interface may be passed.

```
Circle circ = new Circle(12.0);  
Rectangle rect = new Rectangle(4, 7);  
Triangle tri = new Triangle(5, 12, 13);  
printInfo(circ);  
printInfo(tri);  
printInfo(rect);
```

```
Shape[] shapes = {tri, circ, rect};
```


Interface diagram



- Arrow goes up from class to interface(s) it implements.
 - There is a supertype-subtype relationship here; e.g., all Circles are Shapes, but not all Shapes are Circles.
 - This kind of picture is also called a *UML class diagram*.