Guidelines for desingning classes
Interfaces and Polymorphism
Chap 3 and 4

Motivation

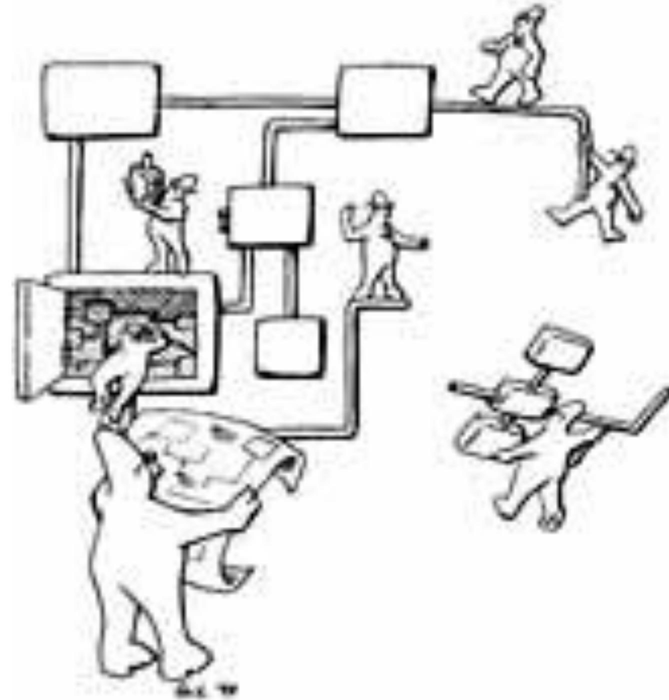
Focus of this course: Use object oriented techniques to program reusable modules that can be used (and reused) for building

larger applications.

Reuse is good: reused software components get tested in many

different programs! They

become more reliable.

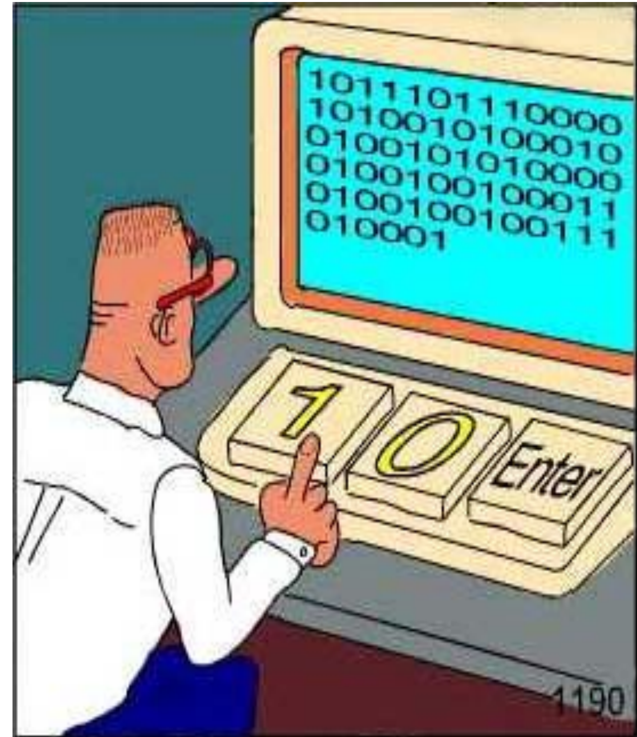


How to

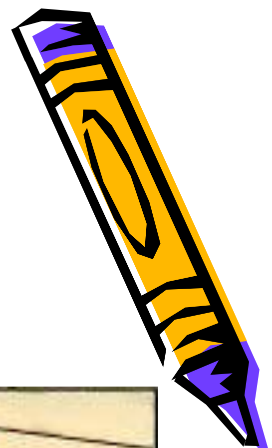
- Bottom up

We will learn how to design libraries of useful classes and objects that can help in many programs.

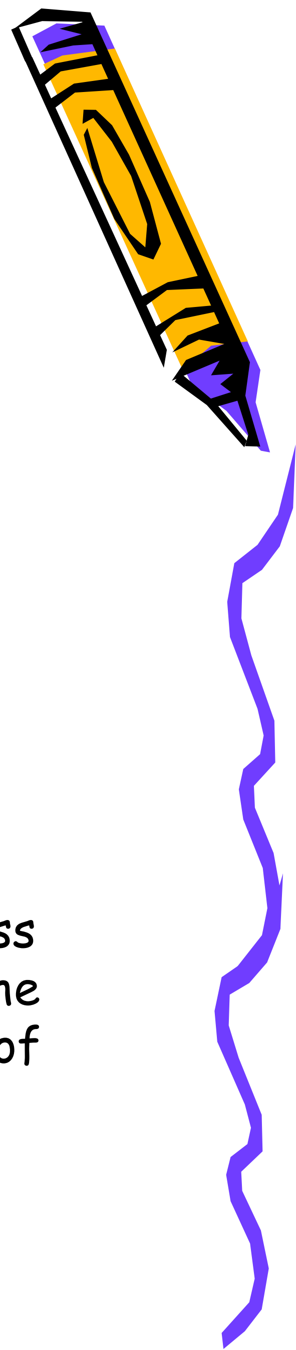
- This is difficult, we have to make life easy for programmers that will use our classes!



Real programmers code in binary.

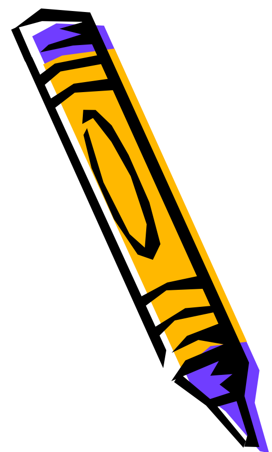


Topics

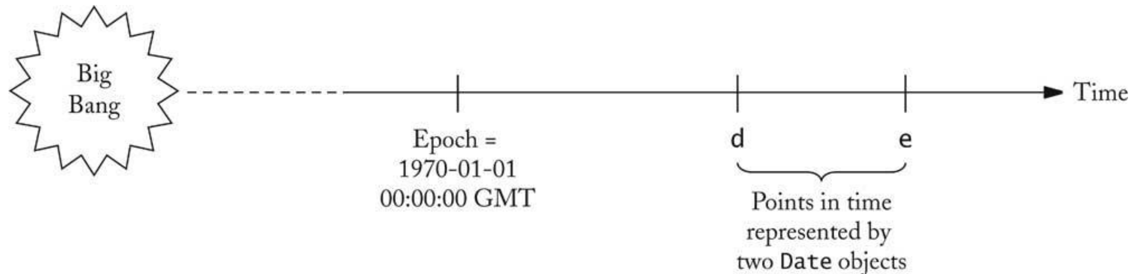


- Encapsulation, Users of a class should be kept unaware of implementation details (implementations might change,users should not need to change their own code!)
- Contracts, Specify in detail responsabilities of the implementor and the caller to increase reliability and efficiency. (comming later in the course, when we discuss Unit Testing)
- Qualities, Cohesion, completeness, conveinience, clarity, consistency.
- Interfaces & Polymorphism Separating the interface of a class from the code that implements it. Many classes can implement the same interface and we can write programs that can work on any of these (polymorphism)

Dates and Calenders- class Day



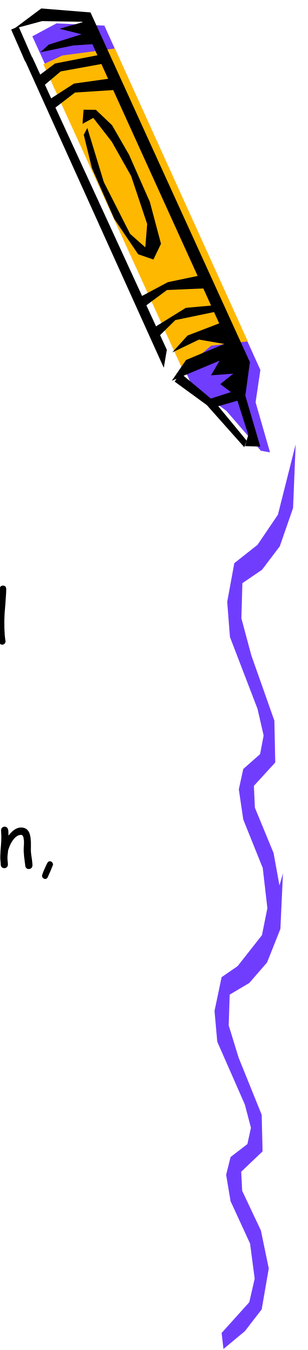
Time in programs



boolean after(Date other)	This date after other?
boolean before(Date other)	This date before other?
int compareTo(Date other)	Which date came before?
long getTime()	Milliseconds gone since epoch
void setTime(long n)	Set to milliseconds since epoch

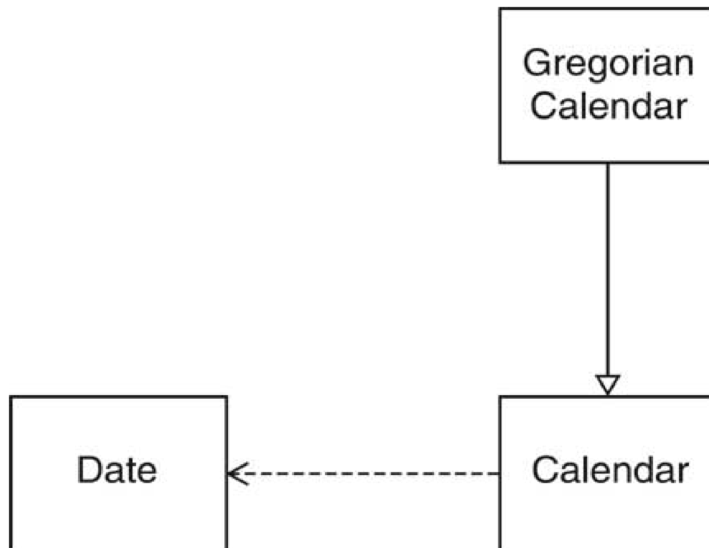
http://download.oracle.com/javase/1.5.0/docs/api/

Calenders in the word



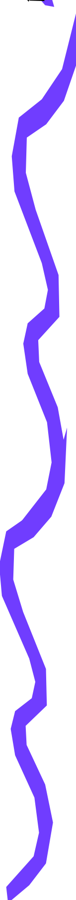
- There are different calendar systems aroud the world. Programs might be required to deal with years, months and days in one or more of them.
- Calendars in use nowadays are Gregorian, Chinese, Hebrew and Islamic.

Dates and Calenders



The (abstract) class java.util.Calendar
Other classes will take care of translating back and forth from points in time to calendar descriptions. A lot of functions are already dened in Calendar.

This separation of concerns is good design! The class Date can be used for other purposes too!



Designing a Day class

- Many programs work with days

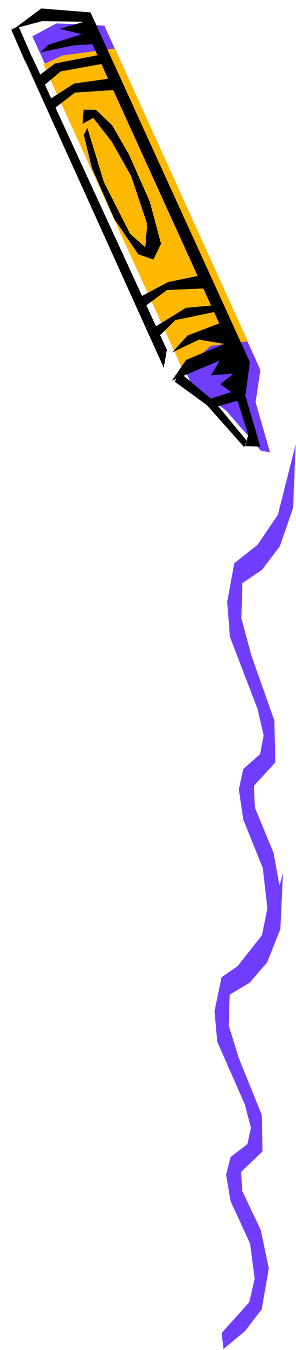
for example March 31, 2010

For the Date class the most useful operation was before, efter

What should be the responsabilies of Day class

1) How many days from now to…?
2) What day is 100 days from nu ?

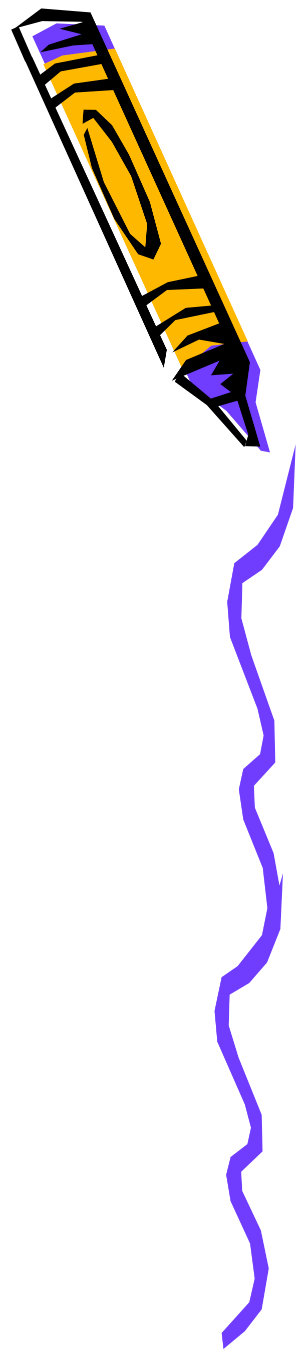
What users of the class expect (Designing the class)



```
public class Day
public Day(int aYear, int aMonth, int aDate)
public int getYear()
public int getMonth()
public int getDate()
public Day addDays(int n)
public int daysFrom(Day other)
...
d.addDays(n).daysFrom(d) == n
d1.addDays(d2.daysFrom(d1)) == d2
```

As you see, there is one constructor, 3 accessors and 2 ways of using days. But there are no mutators!

Test the Day class



```
public class DayTester {
public static void main(String[] args){

Day today = new Day(2001, 2, 3);
Day later = today.addDays(999);
System.out.println( later.getYear()
+ "-" + later.getMonth()
+ "-" + later.getDate());

System.out.println( later.daysFrom( today ));

}
}
```

Designing classes-a first implementation

```
public class Day {

private int year;
private int month;
private int date;
```

addDays and daysFrom are tedious to implement:

April, June, September, November have 30 days
February has 28 days, except in leap years it has 29 days
All other months have 31 days
Leap years are divisible by 4, except after 1582, years divisible by 100 but not 400 are not leap years
There is no year 0; year 1 is preceded by year -1
In the switchover to the Gregorian calendar, ten days were dropped: October 15, 1582 is preceded by October 4

Day- first implementation

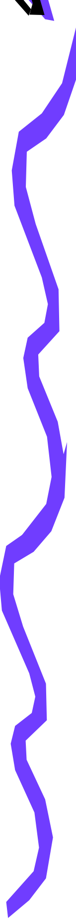
```
public Day addDays(int n){

Day result = this;
while (n > 0){
result = result. nextDay() ;
n--;
}
while (n < 0){
result = result. previousDay() ;
n++;
}
return result;
}
```

Private methods that might use other private filds.
We do not make them public because
for other implementations they might
no be even dened!

Day- first implementation

```
public int daysFrom(Day other){
int n = 0;
Day d = this;
while (d.compareTo(other) > 0){
d = d. previousDay() ;
n++;
}
while (d.compareTo(other) < 0){
d = d. nextDay() ;
n--;
}
return n;
}
```

complete implementation in the labb



A second implementaion

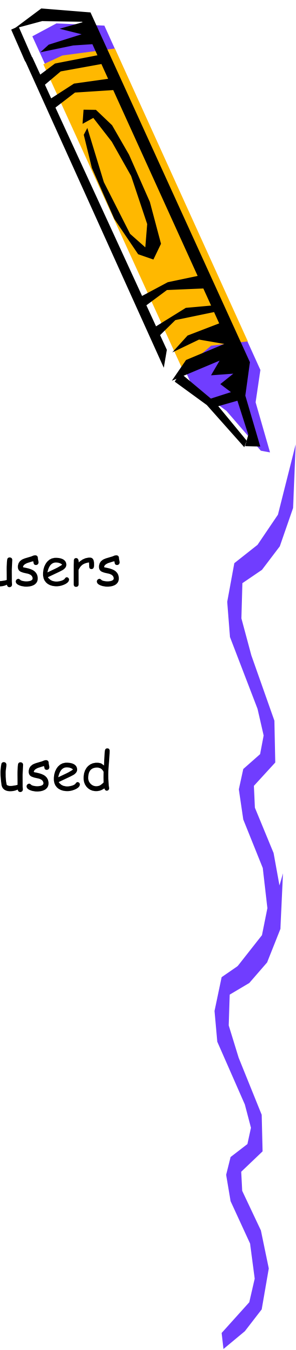
- For greater effciency, use *Julian day number*
- Used in astronomy
- Number of days since Jan. 1, 4713 BCE
- May 23, 1968 = Julian Day 2,440,000
- Greatly simplies date arithmetic

A second implementation

```
public Day(int aYear, int aMonth, int aDate){
julian = toJulian(aYear, aMonth, aDate);
}
public Day addDays(int n){
return new Day( julian + n);
}
public int daysFrom(Day other){
return julian - other.julian;
}
public int getYear(){
return fromJulian(julian)[0];
}
private int julian;
```

Arithmetic becomes efficient, but not the constructor and the selectors!

Encapsulation (information hiding)



- Even a simple class can benefit from different implementations!
- Hiding the representation and auxiliary methods from users of the class makes the user unaware of the differnt implementations!
- Imagine having the fields public. The users might have used

  d.year

  d.year++

It is not easy to change the user code everywhere if the implementation changes.

Encapsulation

- Do not supply a mutator for every accessor! They might lead to inconsistent states! (In our example, not all days are valid!)

Having a setDate might lead to

```
Day deadline = new Day(2001, 1, 31);
deadline.setMonth(2); // ERROR
deadline.setDate(28);
```

or

```
Day deadline = new Day(2001, 2, 28);
deadline.setDate(31); // ERROR
deadline.setMonth(3);
```

Sharing mutable references

```
class Employee{
...
public String getName() return name;
public double getSalary() return salary;
public Date getHireDate() return hireDate;

private String name;
private double salary;
private Date hireDate;

}
```

In a program dealing with employees

```
Employee harry = . . .;
Date d = harry. getHireDate() ;
```

final instance fields

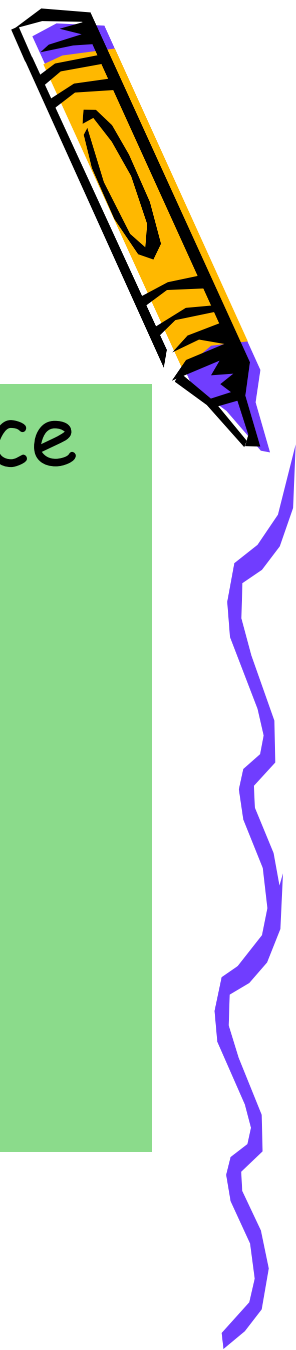
```
public class Day {

private final int year;
private final int month;
private final int date;

}
```

To make sure the object is imutable but….

```
class DataStructure
{
 private final ArrayList datalist;
}
```

If the objects in the datalist are not imutable the datalist will became mutable anyway

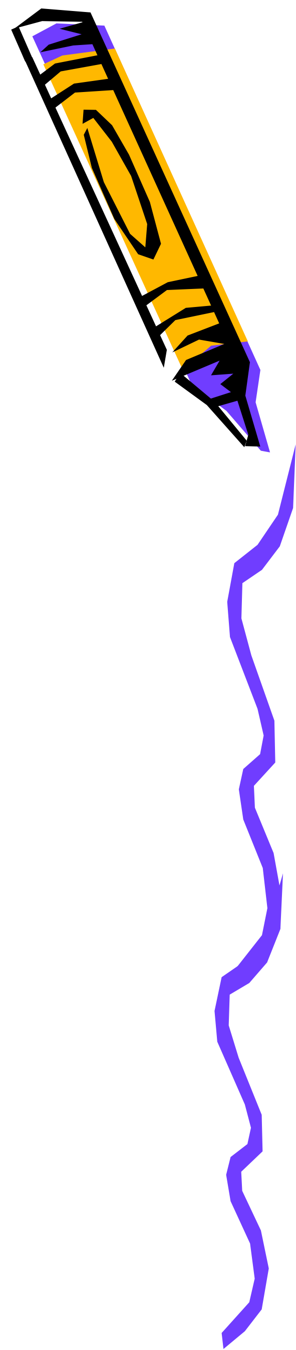
Qualities of a class interface



What criteria should a class interface meet?

- 1 Cohesion
- 2 Completeness
- 3 Convenience
- 4 Clarity
- 5 Consistency

Cohesion



- Class describes a single abstraction
- Methods should be related to the single abstraction

```
public class Mailbox {
public addMessage(Message aMessage) { ... }
public Message getCurrentMessage() { ... }
public Message removeCurrentMessage() { ... }
! public void processCommand(String command) { ... }
```

Completeness

Support all operations that are a part of abstraction

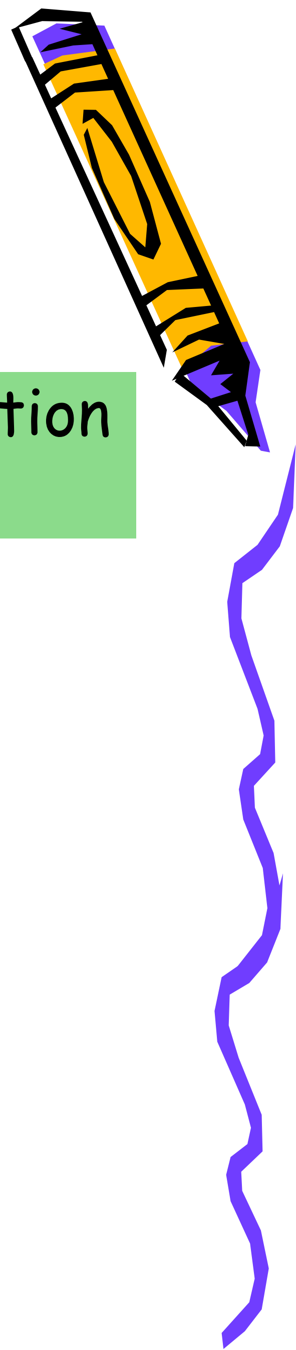
- Why is there no method for calculating how many milliseconds have ellapsed between to dates? (there are methods before, after and getTime)

  Date start = new Date();

  // do some work

  Date end = new Date()

I may be wrong about it !



Convenience

- A good interface makes all tasks possible . . . and common tasks simple

Bad example

- Reading from System.in before Java 5.0:
- BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
- Why doesn't System.in have a readLine method? After all,
- System.out has println!

Clarity and Consistency

Related features should have related
- 1 names
- 2 parameters
- 3 return values
- 4 behavior

Bad example
- In the class GregorianCalendar the constructor

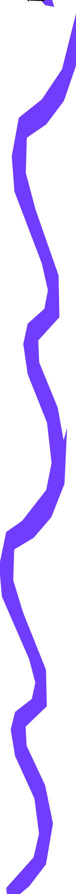
public GregorianCalendar(int year, int month, int dayOfMonth)

expects for year and dayOfMonth values from 1 but for month from 0!

A user interface

- **Displaying an Image**
- Can specify arbitrary image file
- JOptionPane. showMessageDialog(null, "Hello, World!", "Message", JOptionPane.INFORMATION_MESSAGE, new ImageIcon("globe.gif"));

Icon



Use Icon interface

- What if we don't want to generate an image from *file*?
- Fortunately, can use any class that implements Icon *interface type*

```
JOptionPane.showMessageDialog(
 null,
"Hello, Mars!",
"Message",
JOptionPane.INFORMATION_MESSAGE,
new MarsIcon(50)) ;
```

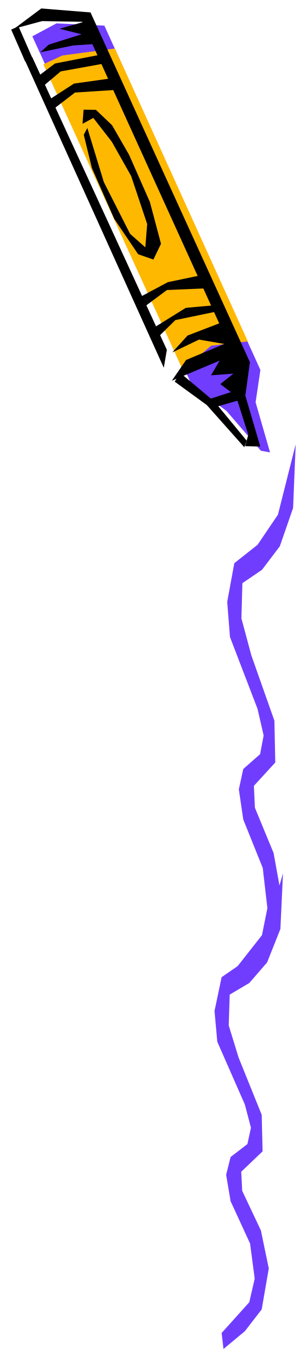
The last argument is of type Icon

The Icon interface create Icon objects

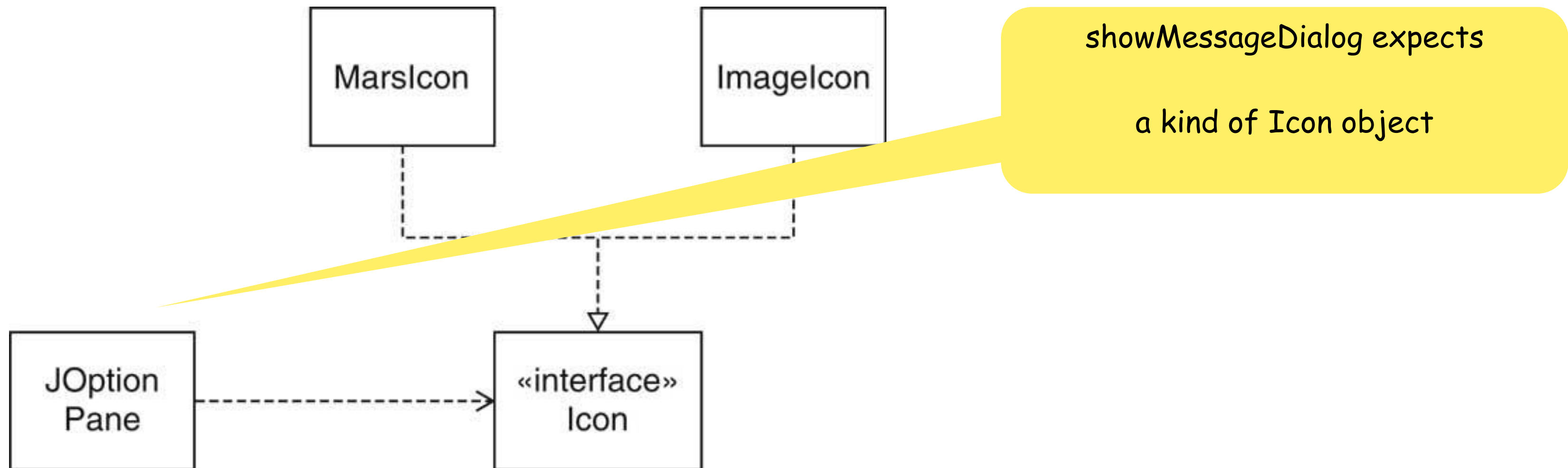
```
public interface Icon{
int getIconWidth();
int getIconHeight();
void paintIcon(Component c, Graphics g,int x,int y);
}
```

Icons objects

```
public class MarsIcon implements Icon{

public MarsIcon( int aSize ){size = aSize;}
public int getIconWidth() {return size;}
public int getIconHeight(){return size;}
public void paintIcon (Component c, Graphics g, int x, int y){
Graphics2D g2 = (Graphics2D) g;
Ellipse2D.Double planet = new Ellipse2D.Double(x, y, size, size);

g2.setColor(Color.RED);
g2.fill(planet);
}

private int size;
}
```



The Icon Interface Type and Implementing Classes



Polymorphism is the capability of an action or ***method*** to do different things based on the object that it is acting …

Benefit of polymorhism
-loose coupling and extensibility-

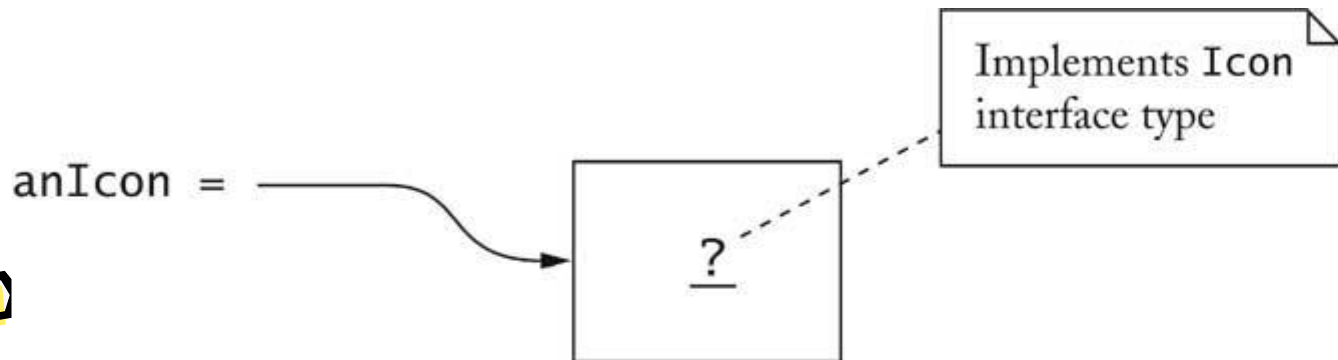
public static void showMessageDialog(......., **Icon anIcon**)

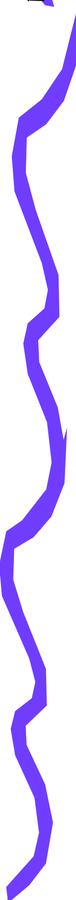
- There are no objects of type Icon!

showMessageDialog doesn't know which icon is passed

- anIcon belongs to a class that implements Icon and that class

defines a getIconWidth method!





The Comparable and Comparator Interface Type

- Collections has static sort method:
  sort(List<T> list)
  Sorts the specified list into ascending order, according to the *natural ordering* of its elements (Comparable)

sort(List<T> list, Comparator<? super T> c)
Sorts the specified list according to the order induced by the specified comparator.