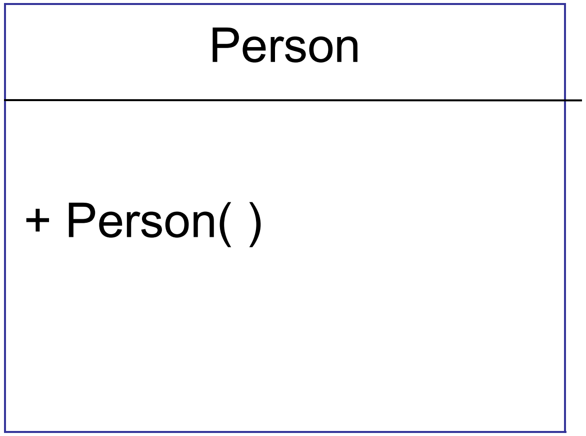
Review of O-O Concepts and UML

James Brucker

Objects and Classes

- Describe the relation between an object `person` and the class `Person`.

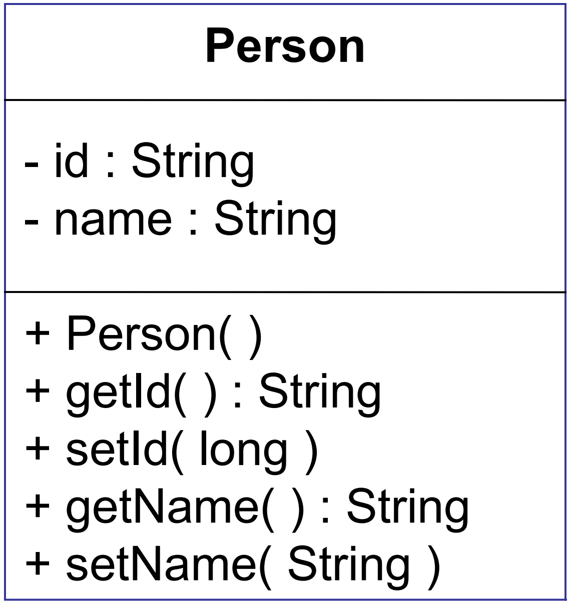


Person

+ Person( )

Objects and Classes

- How do you create a Person object?
- Are there any other ways to create objects?



Person

- id : String
- name : String

+ Person( )
+ getId( ) : String
+ setId(long)
+ getName( ) : String
+ setName(String)

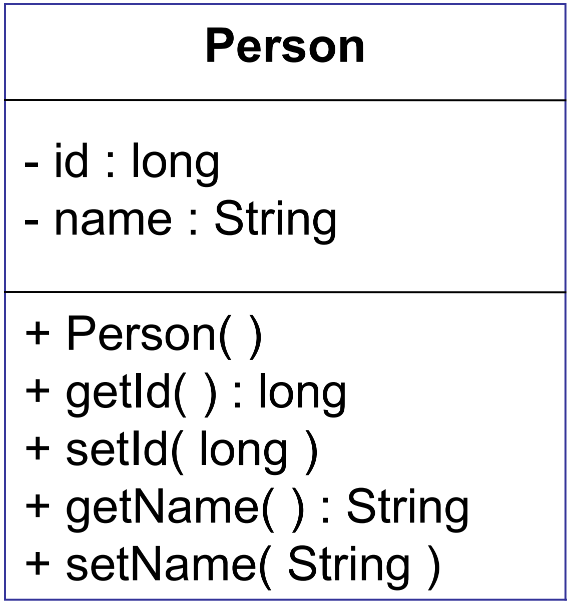
Objects and Classes

- How do you create a Person object?

```
Person p = new Person( );
```

- Are there any other ways to create objects?

Not really, but there are 3 special cases...



Person

- id : long
- name : String

+ Person( )
+ getId( ) : long
+ setId(long)
+ getName( ) : String
+ setName(String)

Objects and Classes

- Are there any other ways to create objects?

  3 Special Cases

  1, String constants.

  String s = "hello"; // new String("hello");

  2. factory methods

  Calendar cal = Calendar.getInstance(Locale);

  3. reflection

  String cl = "Double";

  Object obj = Class.forName(cl).newInstance( );

Constructors

What is the purpose (or job) of a constructor?

The job of the constructor is to prepare a new object for use.

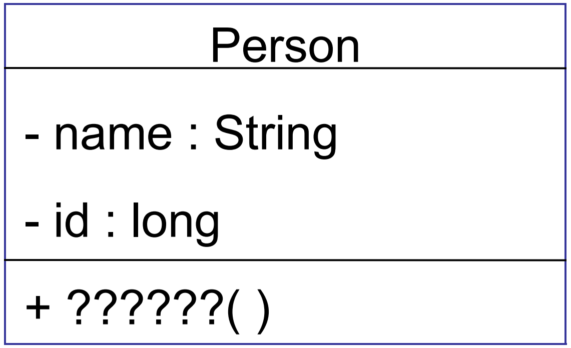
A constructor should initialize all the object's attributes.

Constructor may have other tasks, too.

Constructors

- Must every class have a constructor?

- What happens if a class does not have a constructor?



Person

- name : String

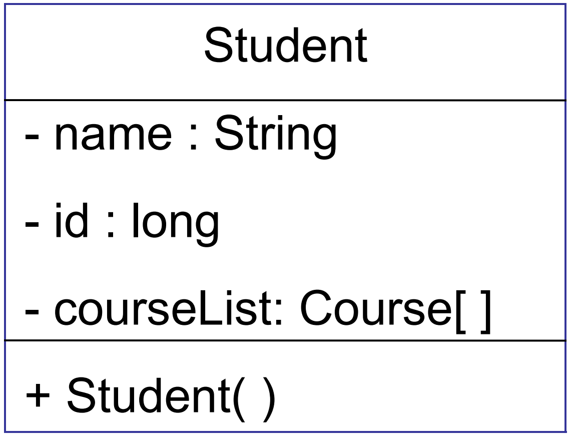
- id : long

+ ??????( )

Constructors

- Must every class have a constructor?
  - No
- What happens if a class does not have a constructor?
  - Java creates a default constructor that sets all primitives to 0 (or false) and sets all references to null.

```
name = null

id = 0L

courseList = null
```



Student

- name : String

- id : long

- courseList: Course[ ]

+ Student( )

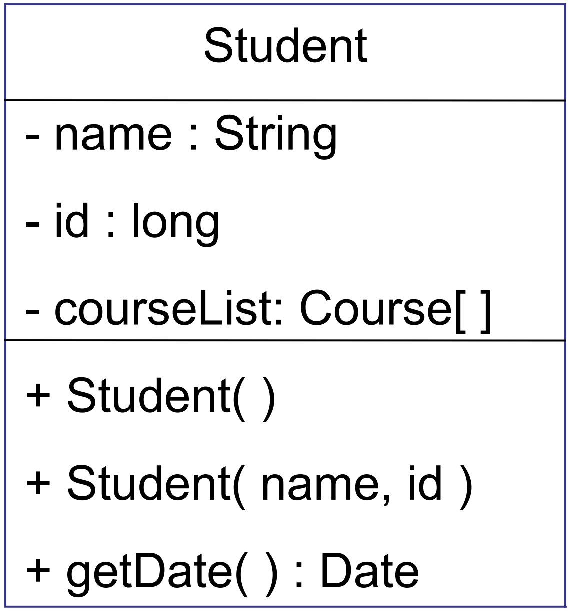
Constructors

- Can a class have more than one constructor?
  - Yes
- Can one constructor call another constructor? How?

  Yes - use "this(...)"

  Rule: **`this()`** must be the FIRST statement in the constructor

```
Student( ) {
   this( "", 99999999 );
}
Student(String name, long id) {
   this.name = name;
   this.id = id;
   courseList = new ArrayList<Course>();
}
```



Student

- name : String
- id : long
- courseList: Course[ ]

+ Student( )
+ Student(name, id)
+ getDate( ) : Date

Object Creation

- Is there any way to *prevent* users from creating objects of a class?

Example:

```
Math m = new Math( );
```

Error!

Object Creation

- Is there any way to prevent users from creating objects of a class?

declare all constructors "private" (you must provide at least one constructor)

Creating Objects: special cases

- Are there any other ways to create objects?

1. String constants: Java creates a "pool" of constants.

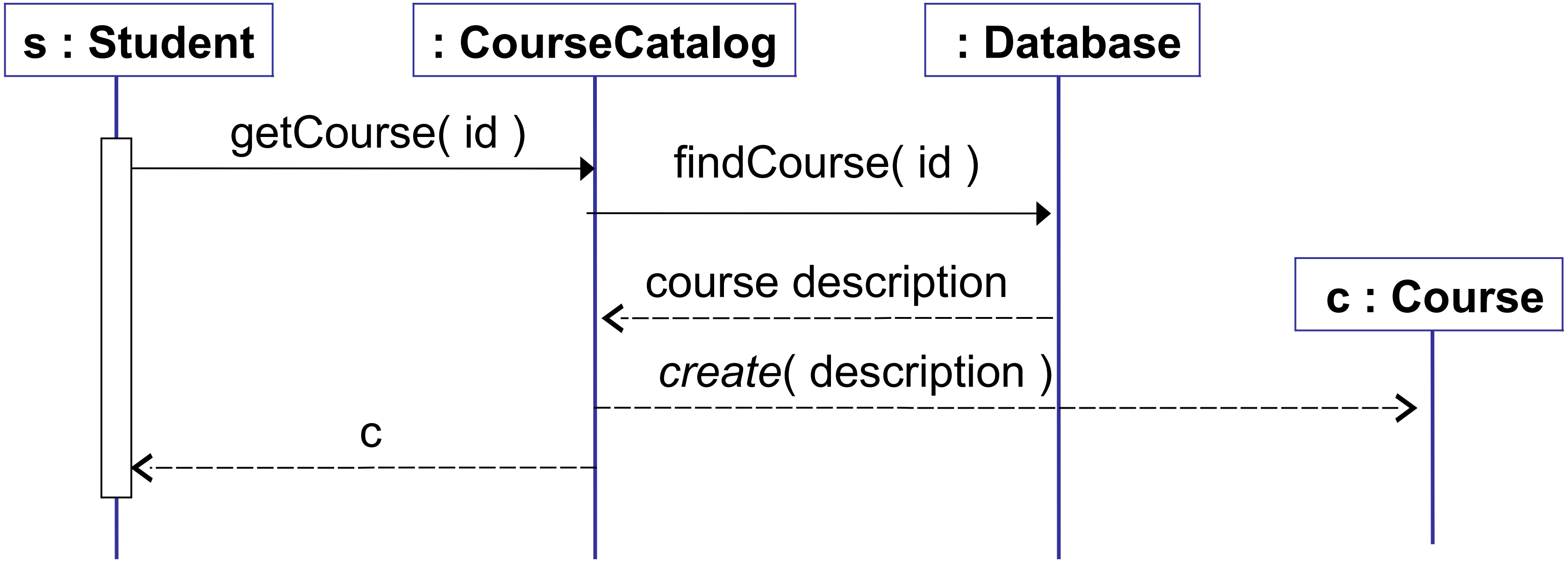
   public final String PROMPT = "Press enter...";

2. Factory Methods: a method that creates an object and returns it.

```
// create a new calendar object
Calendar cal = Calendar.getInstance( );
```

Creating Objects: Factory Methods

- See *Horstmann* for explanation and examples.
- Use factory methods to ...
  - hide complexity of object creation
  - verify parameter before creating object
  - record the object someplace



Object Identity

Suppose we have two identical objects:

```
Person a = new Person("Nok", 12345678);
Person b = new Person("Nok", 12345678);
```

What is the output?

```
if ( a == b )
     out.println("a and b are same");
else out.println("a and b are not same");
```

Object References

- What do these statements do?

```
Person a = new Person("Nok", 12345678);
Person b = a;
```

(a) Person b is a **copy** of Person a.

(b) a and b ***refer*** to the same Person.

(c) this is a programming error! You didn't create Person b first ("`b = new Person(...)`").

Object References

- Consider this:

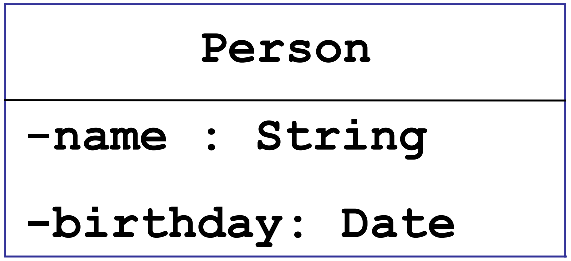
```
Person a = new Person("Nok", 12345678);
Person b = a;
b.setName("Maew");
System.out.println( a.getName() );
```

What value is returned by **`a.getName( )`** ??

Accessor Methods

- Provide access to attributes or "properties".
- Naming:
  - getName( ) - returns value of "name" attribute
  - isOn( ) - returns a boolean "on" attribute
  - isEmpty( ) - accessor returns a *computed* value
- How to write:

```
public String getName( ) {
   return name;
}
```



Person

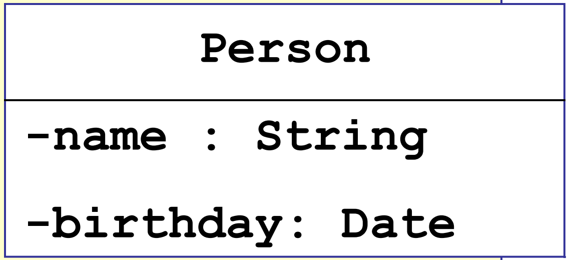
-name : String

-birthday: Date

Accessor Methods for Computed Value

By encapsulation, we can't tell if an accessor returns an attribute or a computed value.

```
/**
 * Get the person's age (approximately).
 * @return age in years.
 */
public int getAge( ) {
   Date now = new Date();
   return now.getYear() - birthday.getYear();
}
```

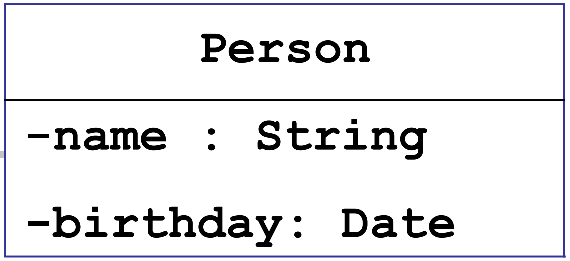


Person

-name : String

-birthday: Date

Mutator Methods



Person

-name : String

-birthday: Date

- Provide ability to change an attribute.
- Don't write them unless they are necessary.
- Naming:
  - setName(String name) - set the "name" attribute
  - setOn(boolean on) - set the boolean "on" attribute
- return **void** (Sun's convention).

```
public void setName(String name) {
   this.name = name;
}
```

Mutator Methods can Validate Data

- Doesn't have to simply assign a value.
- Can perform data validation or manipulation.
- Example:

```
public void setBirtdhay(Date bday) {
   Date now = new Date( );
   // validate the data
   if ( bday == null )
    throw new RuntimeException("can't be null");
   if ( bday.after(now) )
    throw new RuntimeException("not born yet");
   this.birthday = bday;
}
```

Comparing Objects

- `a == b`

  true only if a and b *refer* to the same object.

- `a.equals( b )`

  compare a and b according to the equals method of the class.

1. If a class doesn't supply its own **`equals(Object)`** method, it will *inherit one* from a superclass.
2. The **`equals(Object)`** method of the Object class is the same as ==.

Writing an equals method

Write an equals for Person that is true if two Person's have the same name.

```java
public boolean equals(Object obj) {
   // 1. check for null reference
   if ( obj == null ) return false;
   // 2. verify it is same type as this object
   if ( obj.getClass() != this.getClass() )
      return false;
   // 3. cast it to this class type
   Person other = (Person)obj;
   // 4. perform the comparison logic.
   // Assume we don't allow name to be null.
   return name.equals(other.name);
}
```

Variations on equals method (1)

Sometimes you will see `instanceof` in step 2.

```
public boolean equals(Object obj) {
   // 1. check for null reference
   if ( obj == null ) return false;
   // 2. verify it is same type as this object
   if ( !(obj instanceof Person) ) return false;
   ...
```

`a instanceof B` is true if:

1. `a` is an object from class `B` *or any subclass* of `B`

2. a *implements* B (in case B is an interface). For example:

   `if ( a instanceof Comparable )` /* a has compareTo */

Variations on equals method (2)

So why don't we use `instanceof`?

Sometimes `instanceof` is what you want.

But it has a potential problem: *asymmetry.*

`a.equals(b)` *should return same as* `b.equals(a)`

Using "instanceof", this symmetry may not be true.

Another problem:

if two objects come from different classes,
should we consider them "equals"?

Call by Value

- Java uses **call by value** to pass parameters to methods and constructors.
- This means the method or constructor can't change the caller's value.
- But be careful with references (see later slides).

```
main( ) {
   int n = 10;
   sub( n );
   out.println( n );
   // what is n?
}
```

```
// a complicated method
sub( int n ) {
   n = n + 1;
}
```

Answer: n = 10. sub can change its *copy* of n but not the caller's value.

Call by Value with References

- How about this case?
- Instead of primitive, we pass an object reference.

```
main( ) {
   Date d = new Date();
   out.println( d );
   sub( d );
   out.println( d );
   // did d change?
}
```

```
// change the date
sub( Date x ) {
     x = new Date();
}
```

Answer: No, for same reason as previous slide. d is a *reference* to an object. It contains the address of the object.

In sub, "x = new Date()" changes the reference x, but sub's x is only a *copy* of the reference d in main.

Call by Value with References (2)

- How about this case?
- sub uses a mutator method to change the Date.

```
main( ) {
   Date d = new Date();
   out.println( d );
   sub( d );
   out.println( d );
   // did d change?
}
```

```
// change the year
sub( Date x ) {
   int year = x.getYear();
   year++;
   x.setYear( year );
}
```

Answer: Yes!

in sub, x refers to (points to) the same date object as in main, so by changing the **contents** of the Date object, the object referred by d (in main) changes, too.

Another Example: swap

- Can we do this to swap two variables of primitive type

```
main( ) {
   int a = 10;
   int b = 20;
   swap( a, b );
   // what is a?
   // what is b?
}
```

```
// swap values
swap( int x, int y ) {
   int tmp = y;
   y = x;
   x = tmp;
}
```

Answer: a = 10, b = 20.

Same as previous example. Java passes a *copy* of the parameter value to swap, so swap can't change the caller's values.

swap using Objects

- Can we do this to swap values of two objects?

```
main( ) {
   String a = "hi";
   String b = "bye";
   swap( a, b );
   // what is a?
   // what is b?
}
```

```
// swap values
swap(String x, String y)
{
   String tmp = y;
   y = x;
   x = tmp;
}
```

Answer: a = "hi", b = "bye".

swap doesn't work for same reason as given in previous slide.

swap using Mutable Objects

- Can we do this to swap values?

```
main( ) {
   Person a =
      new Person("Hi");
   Person b =
      new Person("Bye");
   swap( a, b );
   // who is a?
   // who is b?
}
```

```
// swap attributes
swap(Person x,Person y)
{
 String nx= x.getName();
 String ny= y.getName();
 y.setName( nx );
 x.setName( ny );
}
```

Answer: a and b still refer to the original objects, but now a has name "Bye" and b has name "Hi".

As in previous example, method has a *reference* to same object as in main, so if he changes the object that reference points to, it changes object in main.

Another Example: swap with array

- Can we do this to swap two primitive variables?

```
main( ) {
   int [] a = new int[2];
   a[0] = 10;
   a[1] = 20;
   swap( a );
   // what is a[0]?
}
```

```
// swap array elements
swap( int[] x ) {
   int tmp = x[0];
   x[0] = x[1];
   x[1] = tmp;
}
```

Answer: `a[0] = 20, a[1] = 10.  Yes, it swaps.`

Arrays are reference type. When we elements of the array, its like changing attributes of an object. it affects the caller, too. Its like calling a.setName( ) for an object. Create an array in BlueJ's object workbench and inspect it.

Another Example: create a new array

- Can we do this to change an array?

```
main( ) {
   int [] a = new int[2];
   a[0] = 10;
   a[1] = 20;
   sub( a );
   // what is a[0]?
}
```

```
// create new array
sub( int[] x ) {
   x = new int[2];
   x[0] = 100;
   x[1] = 200;
}
```

Answer: `a[0] = 10, a[1] = 20.`

In main, a is a *reference* to an array. When we call sub( ), Java *copies* the reference into sub's parameter. sub can change his copy of the *reference*, but it doesn't affect the reference in main.

Another Example: change String array

- Can we change elements of a String array like this?

```
main( ) {
   String[] a =
      new String[2];
   a[0] = "java";
   a[1] = "beans";
   sub( a );
   // what is a[0]?
}
```

```
// change array elements
sub( String[] x ) {
   x[0] = "coffee";
   x[1] = "grounds";
}
```

Answer: `a[0] = "coffee", a[1] = "grounds".`

Arrays are reference type, so "a" is just a reference (pointer) to the array data. Changing elements of an array affects ***every reference*** to the same array.

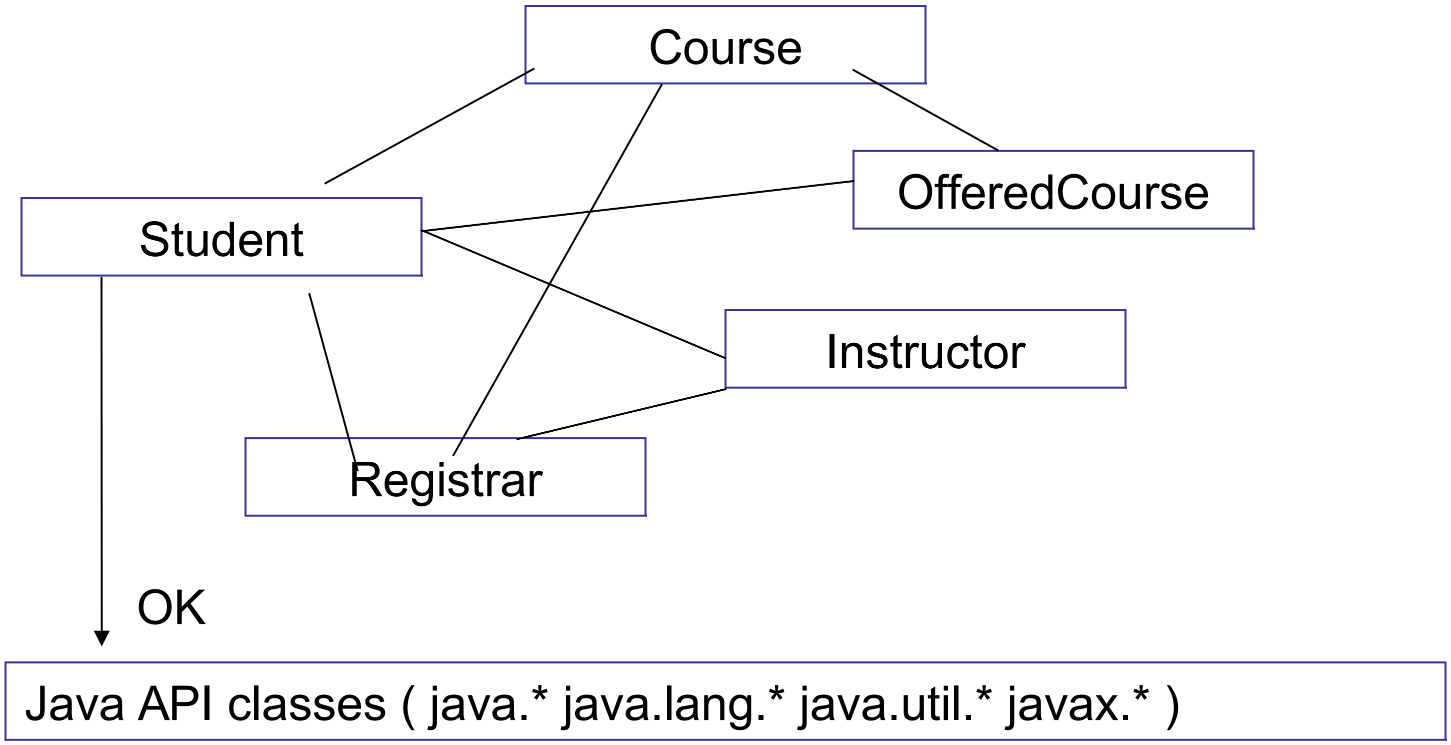
Class Design

Name 5+1 criteria for good class design

- Hint: the names start with the letter "c".

1. Clarity
2. Consistency
3. Completeness - *just* enough methods to do all its responsibilities
4. Convenience - useful for the application it was designed for
5. Cohesion (high) - behavior and state of a class are related to the same purpose
6. Coupling (low) - class does not depend on many other (unstable) classes . Dependency on JavaSE is ok

Class diagram with high coupling



Course

OfferedCourse

Student

Instructor

Registrar

OK

Java API classes (java.* java.lang.* java.util.* javax.*)

Classes depend on many other classes.

3 Properties of Objects

- What are the 3 characteristics of objects?
  - Give example of each characteristic using the code below.

```
class Student {
   private String name;
   private Date birthday;
   private String id;
   private List<Course> courseList;

   public addCourse( Course course ) {
      courseList.add( course );
   }
   public static void main( ... ) {
      Student s1 = new Student("Bill Gates");
      Student s2 = new Student("Bill Gates");
```

Encapsulation of Attributes

- What is encapsulation?
- Give examples from the code below.

```
class Purse {
   private int capacity;
   private List<Coin> coins;
   public Purse( int capacity ) {
      this.capacity = capacity;
      coins = new ArrayList<Coin>(capacity);
   }
   public boolean insert(Coin coin) {
      return list.size() < capacity && list.add(coin);
   }
   public Coin[] withdraw(int amount) { ... }
}
```

Benefit of Encapsulation

What are the benefits of encapsulation?

1. We can change the implementation without any affect on the other parts of program.

   Example: Purse could store Coins in a List or an array.

2. Reduces coupling between classes.

   What other classes can't see they can't couple to.

3. Simplifies the program.

   An object *encapsulates* all the data it needs to do its job. We don't have to store the data elsewhere.

Polymorphism

- *See separate set of slides.*
- We can invoke the same method (by name) using different objects.
- The *actual method* invoked depends on the object.
- It is decided at runtime.

```
Object p = null;
p = new Date( );
System.out.println( p.toString() ); // print the date
p = new Coin( 10 );
System.out.prinln( p.toString() ); // print coin value
```

Polymorphism (2)

- It is enabled by Interfaces and inheritance.
- Example using interface:

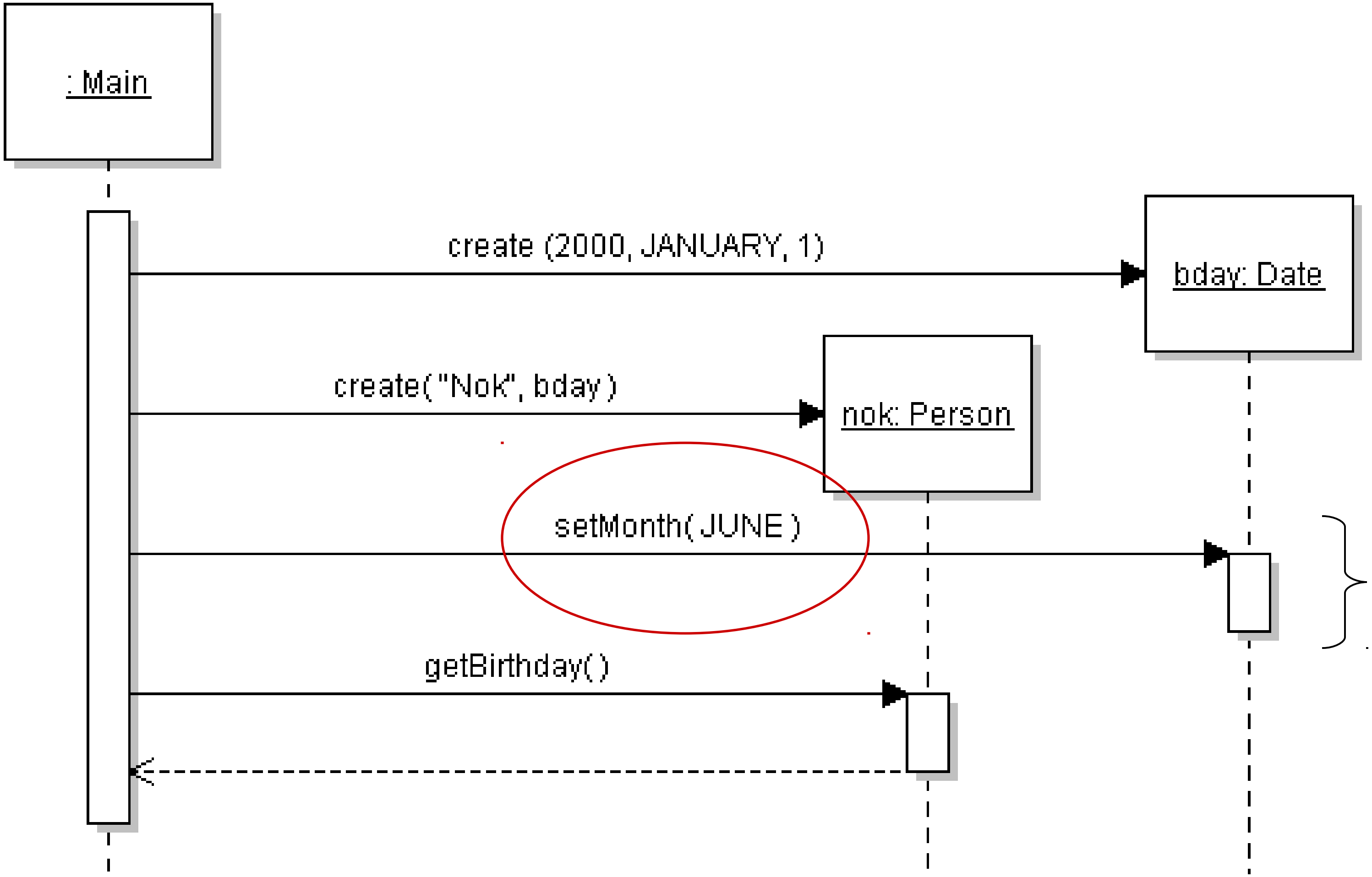
```
/**
 * Return the "maximum" of any two objects.
 * It works even though the class of a and b
 * are not known.
 */
public Object max(Comparable a, Comparable b) {
   if ( a.compareTo(b) > 0 ) return a;
   else return b;
}
```

Breaking Encapsulation

- if an object exposes its internal structure, it can break encapsulation
- *copying a reference to a mutable object can break encapsulation*

```
class Person {
   private String name;
   private Date birthday;  // Date is mutable
   public Person(String name, Date bday) {
      this.name = name;
      this.birthday = bday;
   }
   public String getName() { return name; }
   public Date getBirthday() { return birthday; }
}
```

Can we change a Person's birthday?



Does this change Nok's birthday?

Can we change a Person's birthday?

```
// java.util.Date adds 1900 to the year.
// So year 100 = 100 + 1900 = 2000 AD
Date bday = new Date( 100, Calendar.JANUARY, 1);
Person nok = new Person( "Nok", bday );


// we already created Nok.
// Change the date object.
bday.setMonth( Calendar.JUNE );
System.out.println(
   nok.getBirthday() );
```

Does this change Nok's birthday?

Breaking Encapsulation (Again)

- examine the source code for Person

  `this.birthday = bday;`

  copies the *reference* to the Date object.

- So `birthday` and `bday` refer to the same object.

```
class Person {
   private String name;
   private Date birthday;  // Date is mutable
   public Person(String name, Date bday) {
      this.name = name;
      this.birthday = bday;
   }
```

Can we change a Person's name?

```
// java.util.Date adds 1900 to the year.
// So year 100 = 100 + 1900 = 2000 AD
Date bday = new Date( 100, Calendar.JANUARY, 1);
String name = "Nok";
Person nok = new Person( name, bday );

// we already created Nok.
// Change the name
name = name.toUppercase();
System.out.println(
   nok.getName() );
```

Does this change Nok's name?

Can we change a Person's name?

No.

```
name = name.toUppercase();
```

creates a *new* String. The old String didn't change.

- String objects are *immutable* (cannot be changed).
- Copying a reference to a String is safe.

```
class Person {
   private String name; // String is immutable
   private Date birthday;
   public Person(String name, Date bday) {
      this.name = name;
      this.birthday = bday;
   }
```

Controlling Object Creation

- For some classes, creating an object is expensive.
- We want to have control over creating the objects.
- Use a *private constructor* to prevent object creation.

```
public class ProductCatalog {
  private String DBASE = "jdbc:mysql://ku/catalog";
  private ProductCatalog( ) {
     open the database and initialize attributes
  }
  public static ProductCatalog getProductCatalog( )
   {
     catalog = new ProductCatalog( );
     return catalog;
  }
```

why is this method **`static`** ***?***

Creating Only ONE ProductCatalog

- Create one catalog (the first time).
- Save a reference to this catalog as **static** variable
- *Singleton Pattern* - always return the same catalog.

```
public class ProductCatalog {
  // save a static reference to the catalog
  private static ProductCatalog catalog = null;
  private CourseCatalog( ) {
    open the database and load products ... }
  /** method always returns the same catalog */
  public ProductCatalog getProductCatalog( ) {
    if ( catalog == null ) // create it first time
      catalog = new ProductCatalog( );
    return catalog;
  }
```

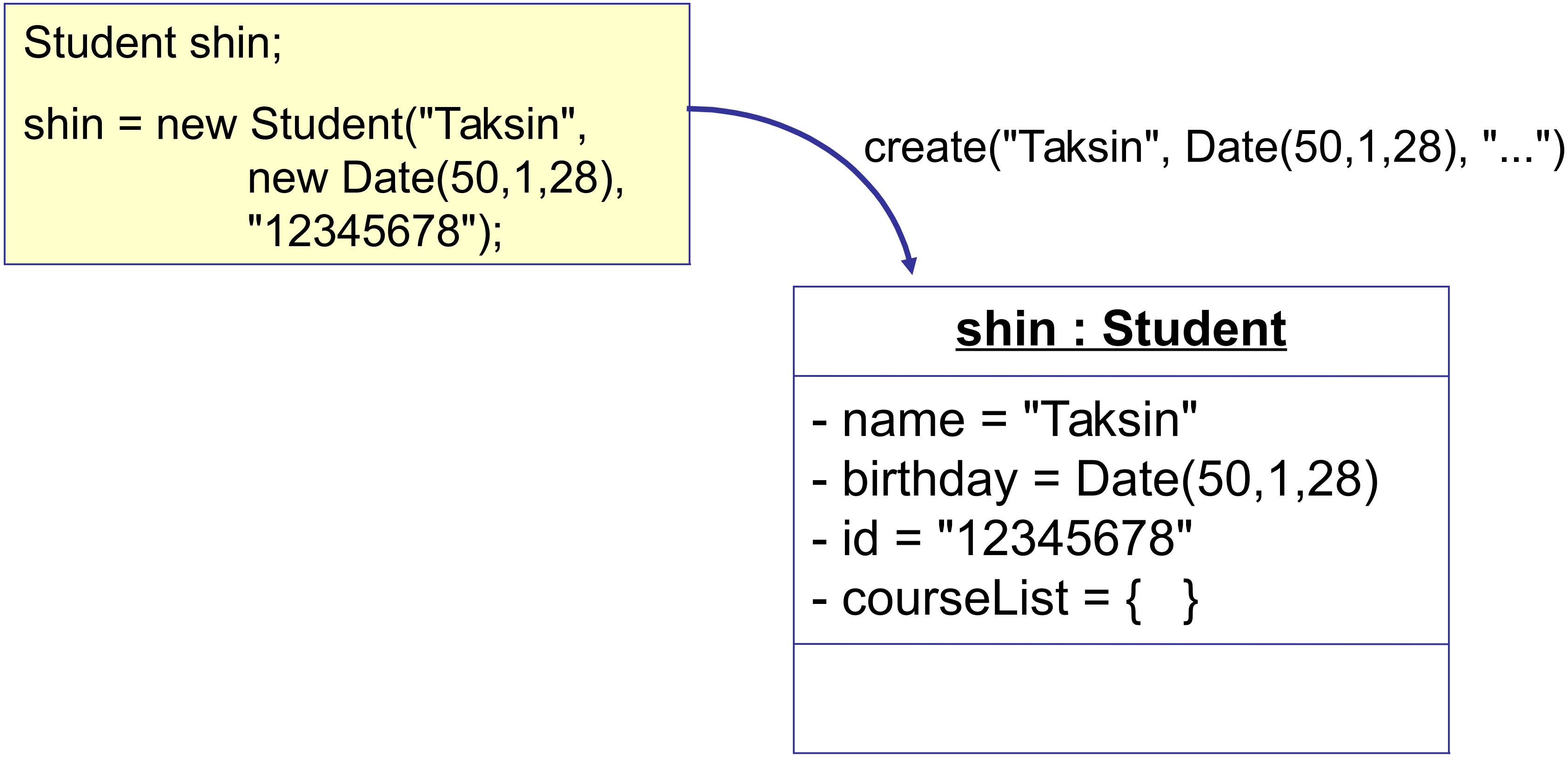
Example: java.util.Calendar

- Creating a Calendar requires knowing a valid Locale.
- Calendar constructors are protected.
- has several static **`getInstance( )`** methods.

```
Calendar cal = Calendar.getInstance( );

// a localized calendar
Locale thai = new Locale( "th" );
Calendar thaiCalendar = Calendar.getInstance( thai );
```

Constructor Parameters (1)

- How can you guarantee that objects have a useful "state" when you create them?



```
Student shin;

shin = new Student("Taksin",
                   new Date(50,1,28),
                   "12345678");
```

create("Taksin", Date(50,1,28), "...")

<u>shin : Student</u>

- name = "Taksin"
- birthday = Date(50,1,28)
- id = "12345678"
- courseList = {   }

Constructor Parameters (2)

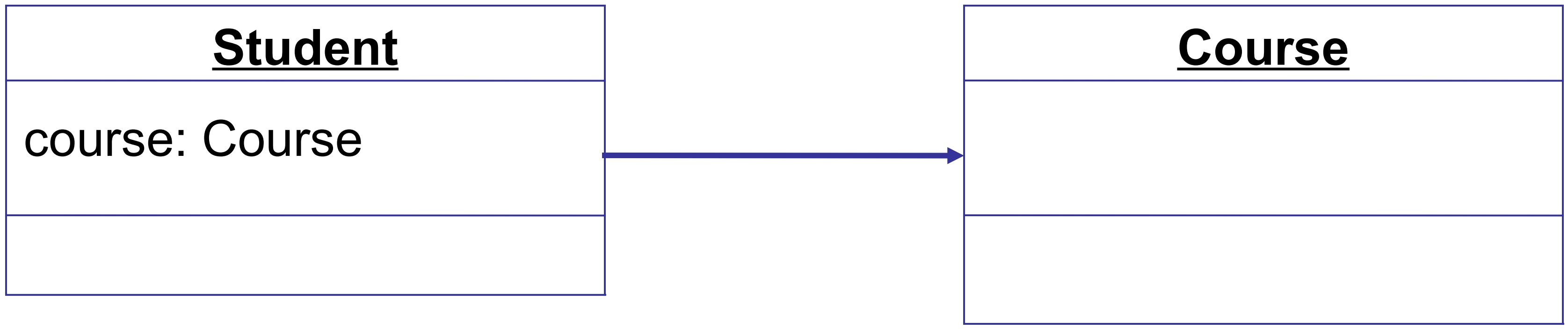
- What is the purpose of providing parameters to a constructor?

```
class Student {
   private String name;
   private Date birthday;
   private String id;
   private List<Course> courseList;

   public Student( String name, Date bday, String id )
   {
      why parameters?  what to do?

      courseList = ??what??
   }
```

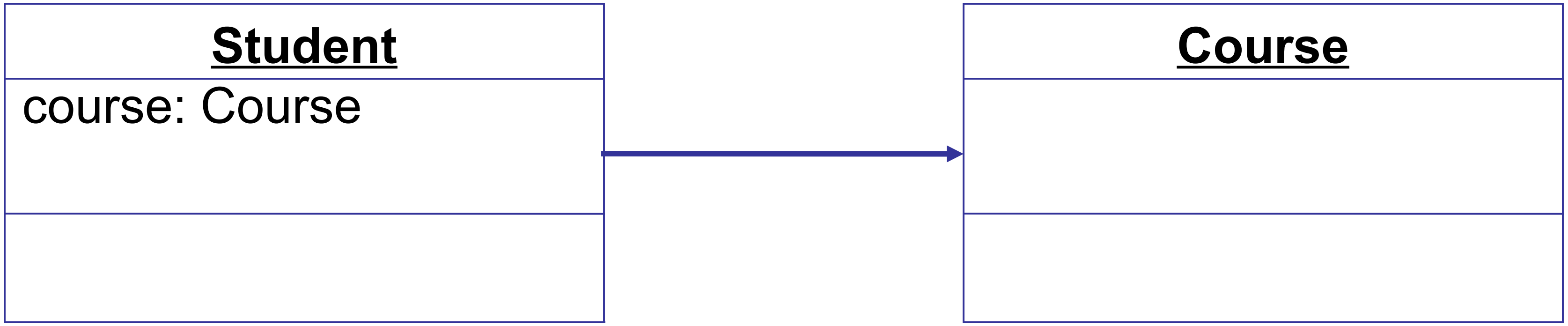
UML Notation

- Explain the meaning of the following UML notation
- Give an example in Java

1. what does this mean?

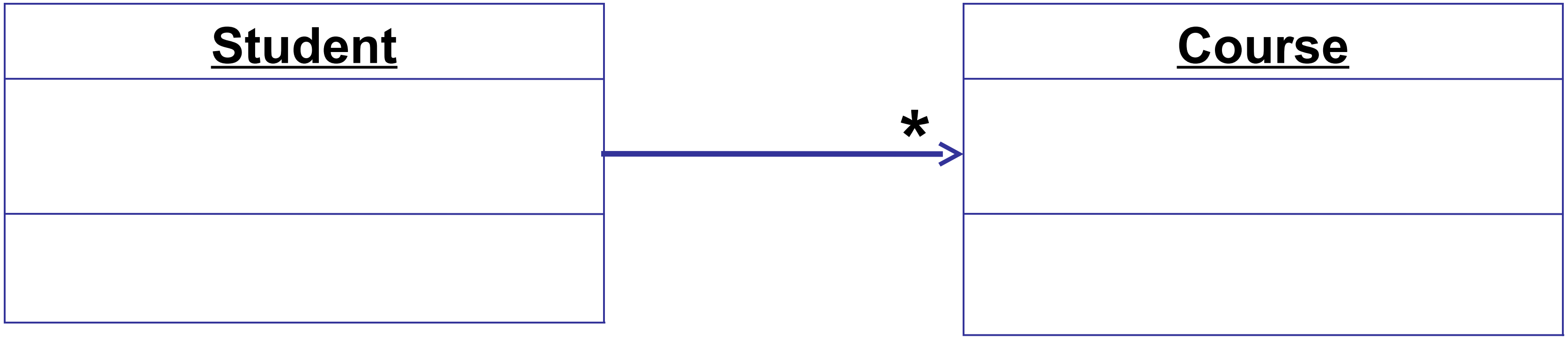


1. (Directed) Association: "has a"



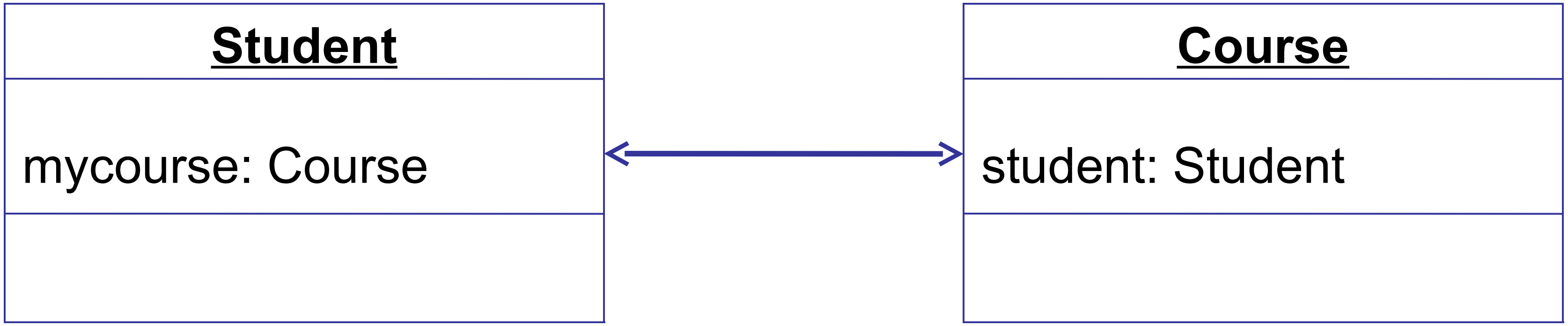
```
public class Student {

   private Course course;
```

1. Association with multiplicity > 1



```
public class Student {

   private Collection<Course> course;
```

3. Bidirectional Association



Association can be "navigated" from other end.

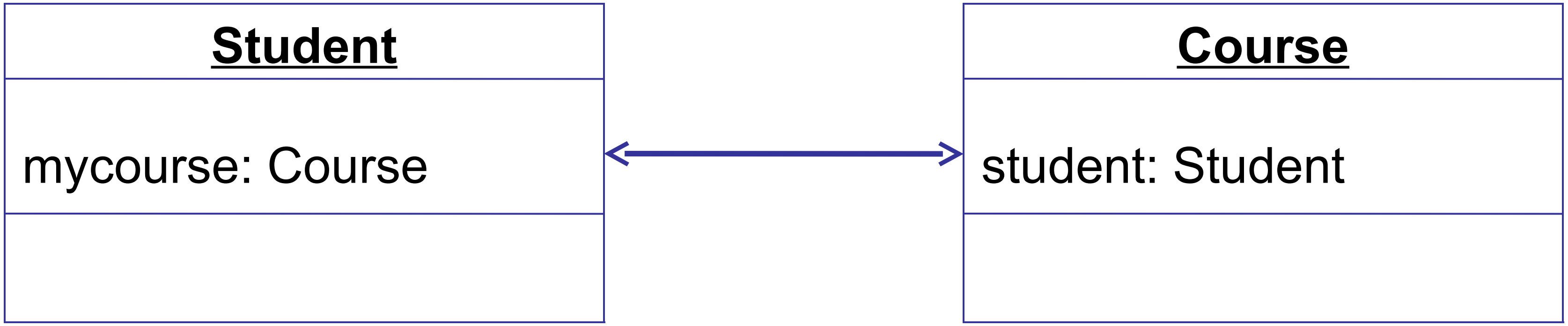
How to ensure that both sides agree? `mycourse.getStudent() == this`

1) use an association class

2) add Java code to enforce 1-to-1

```
public void setCourse( Course course ) {
   this.mycourse = course;
   mycourse.setStudent( this );
}
```

3. Bi-directional Association



Student

mycourse: Course

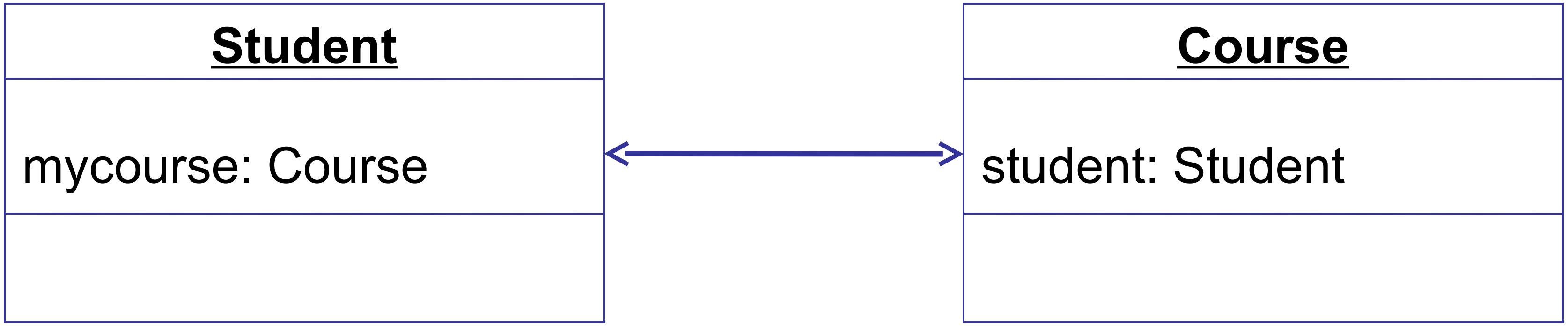
Course

student: Student

```
public class Student {
   Course mycourse;
   public Student( ) {
      mycourse = new Course( this );
   }
```

```
public class Course {
   Student student;
   public Course( Student student ) {
      this.student = student;
   }
```

3. Can you think of another way?



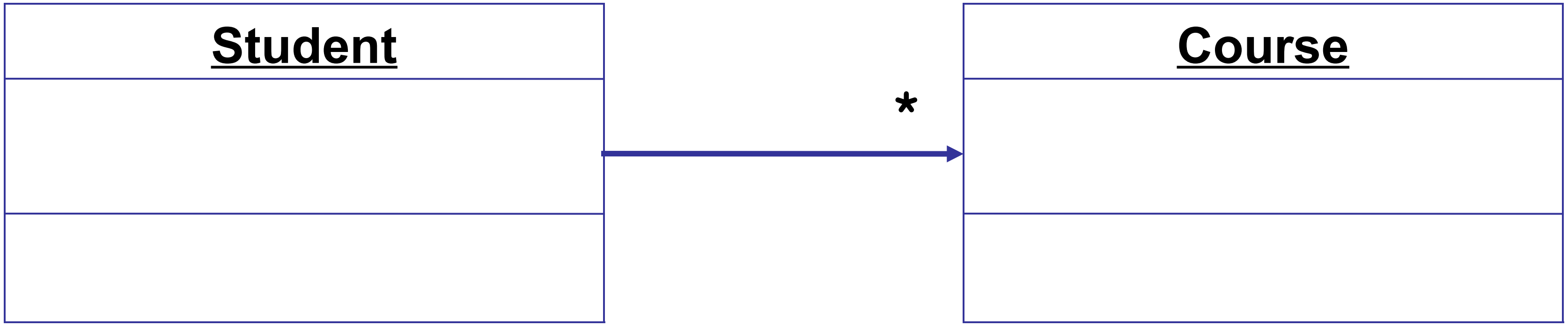
Student

mycourse: Course

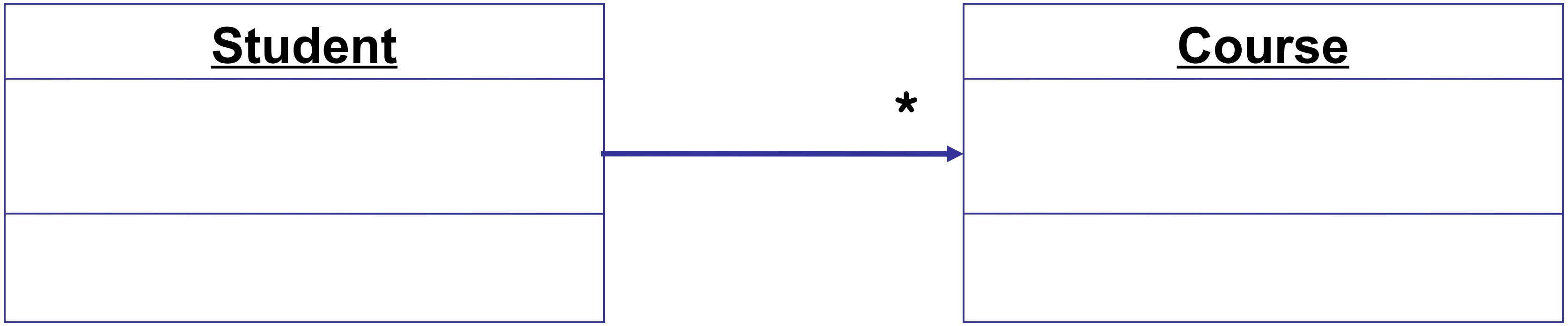
Course

student: Student

4. what does this mean?

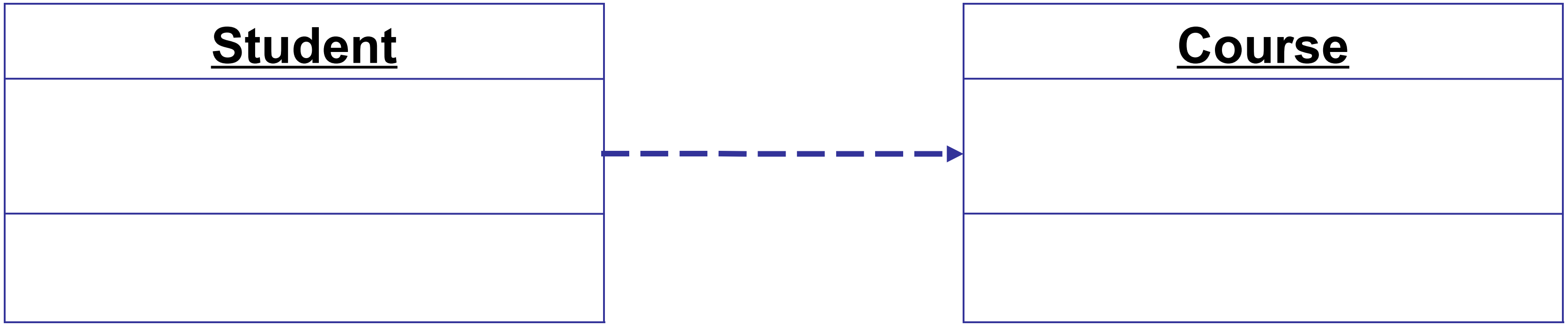


4. One-to-many association

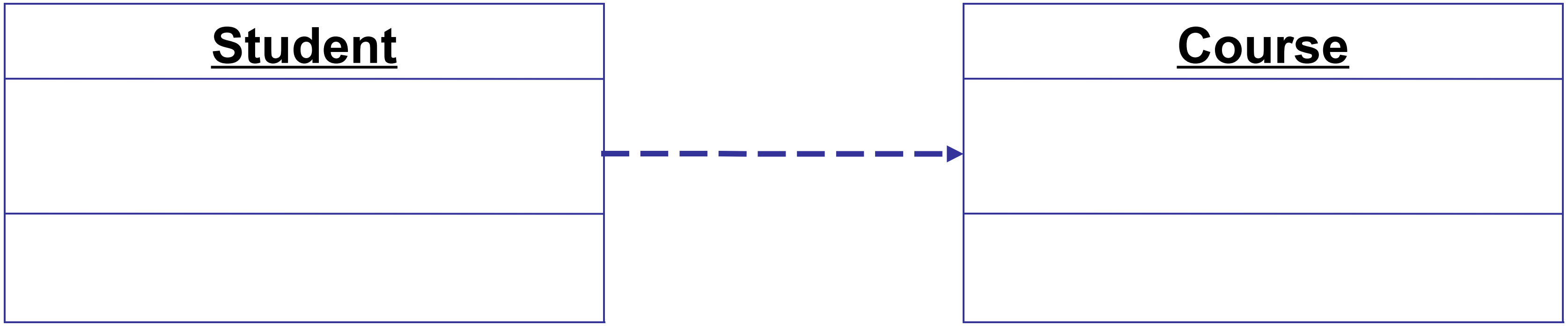


```
public class Student {
   private Collection<Course> mycourses;
   public Student( ) {
      mycourses = new ArrayList<Course>( );
      == or ==
      mycourses = new HashSet<Course>( );
      mycourses = new LinkedList<Course>( );
```

5. what does this mean?

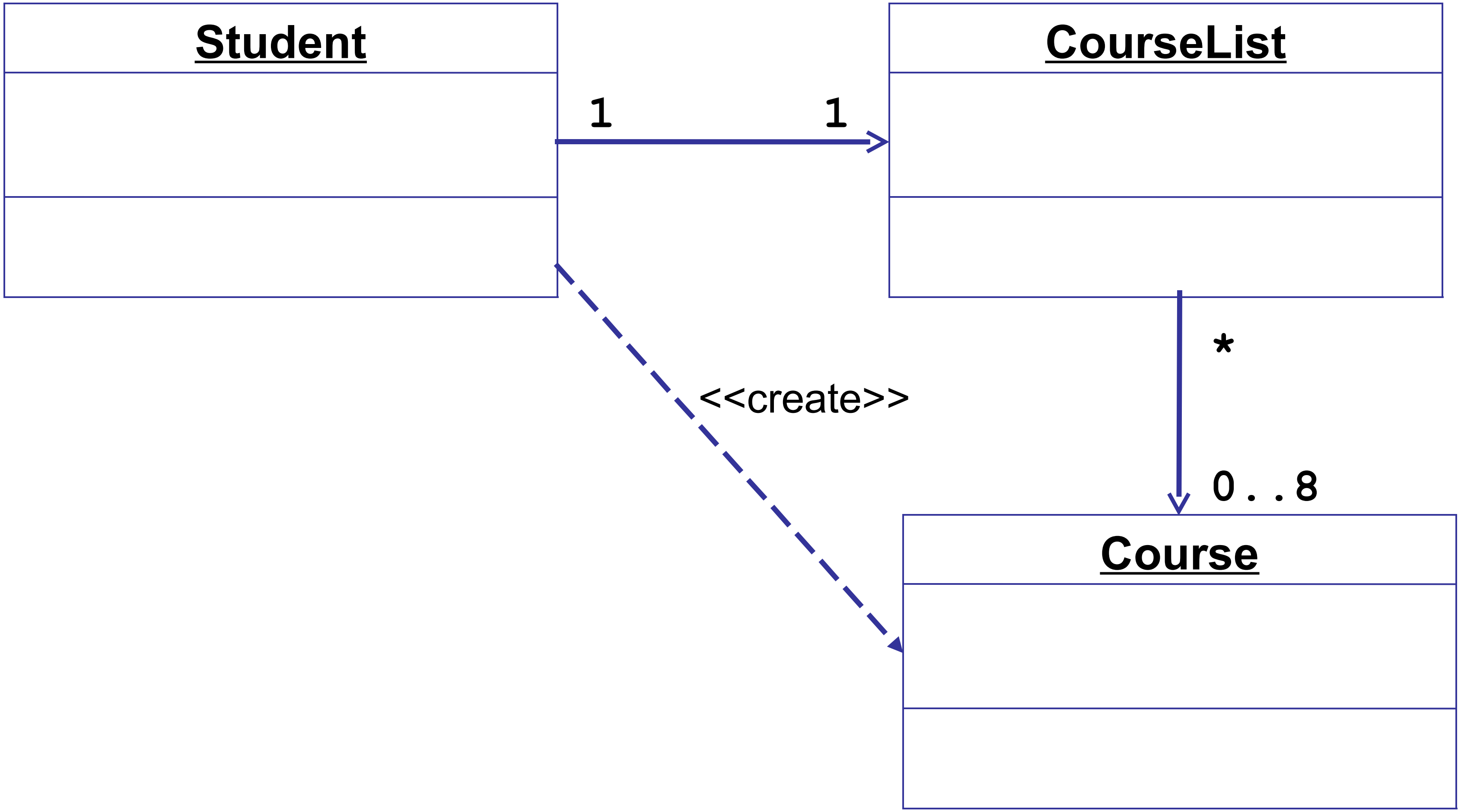


5. Dependence: "uses"

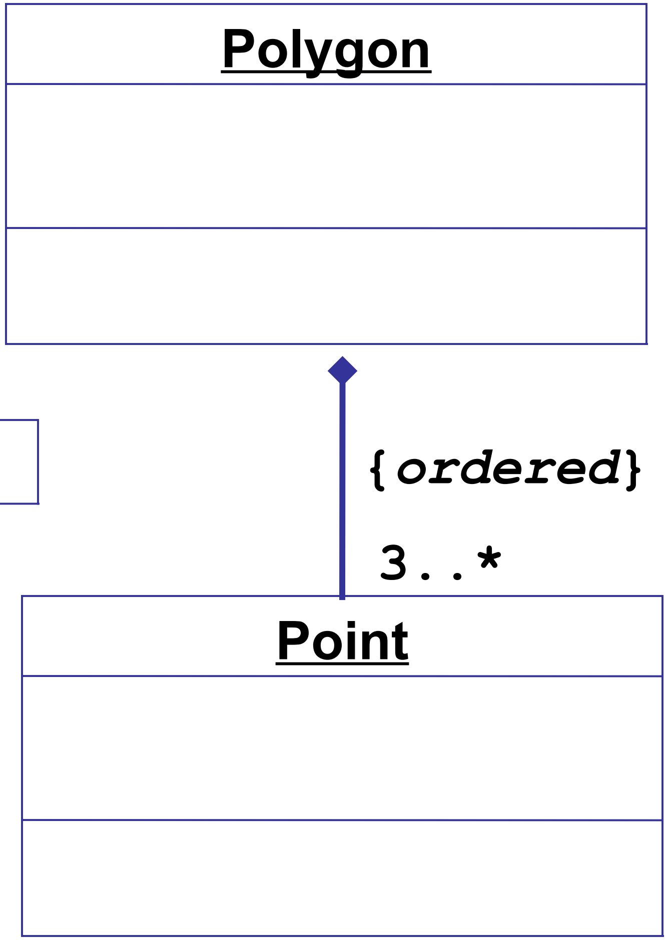


```
public class Student {
   CourseList mylist;
   public Student( ) {
      mylist = new CourseList( );
   }
   // dependency created by a parameter
   public void addCourse( Course c ) {
      mylist.add( c ) ;
   }
```

6a. describe the relationships

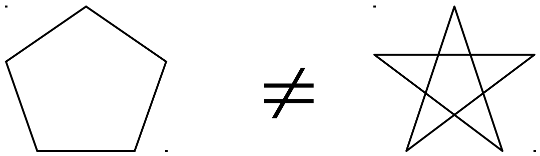


6b.



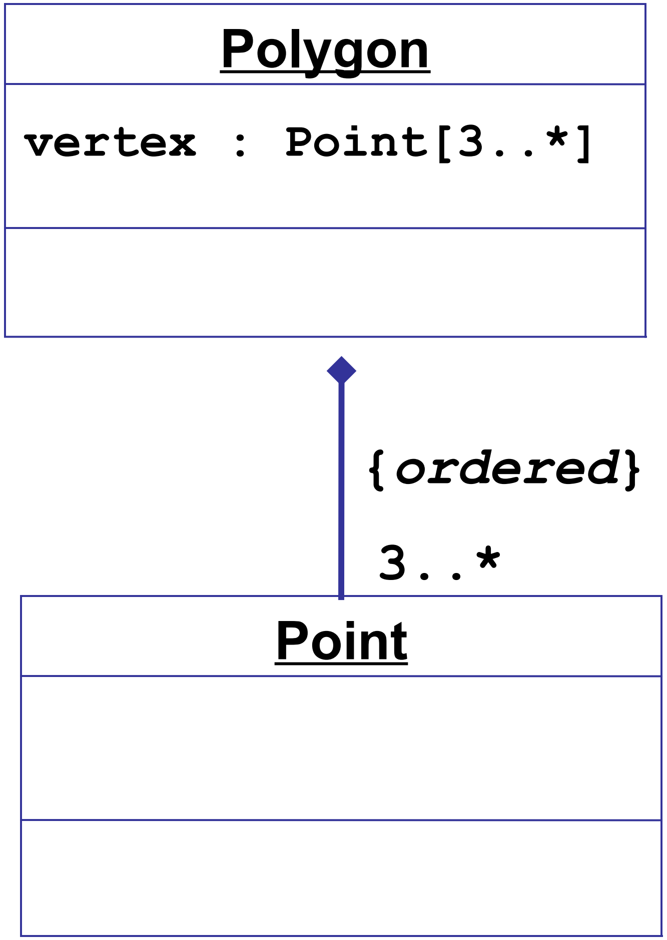
what are the key properties of this relation?

6c.



polygon has an ***ordered*** *coll.* of vertices

polygon *owns* its vertices



Polygon

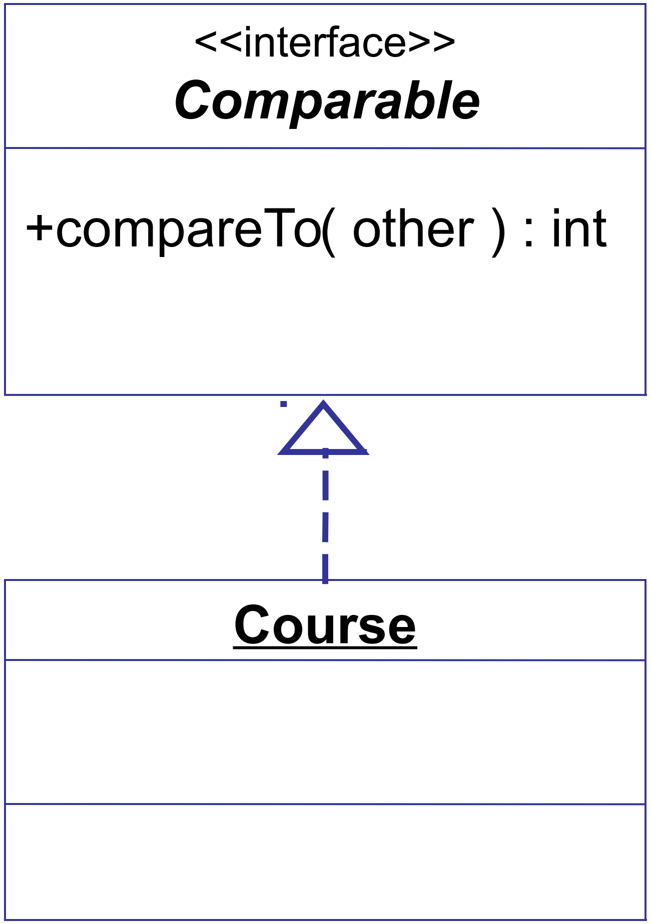
vertex : Point[3..*]

{*ordered*}

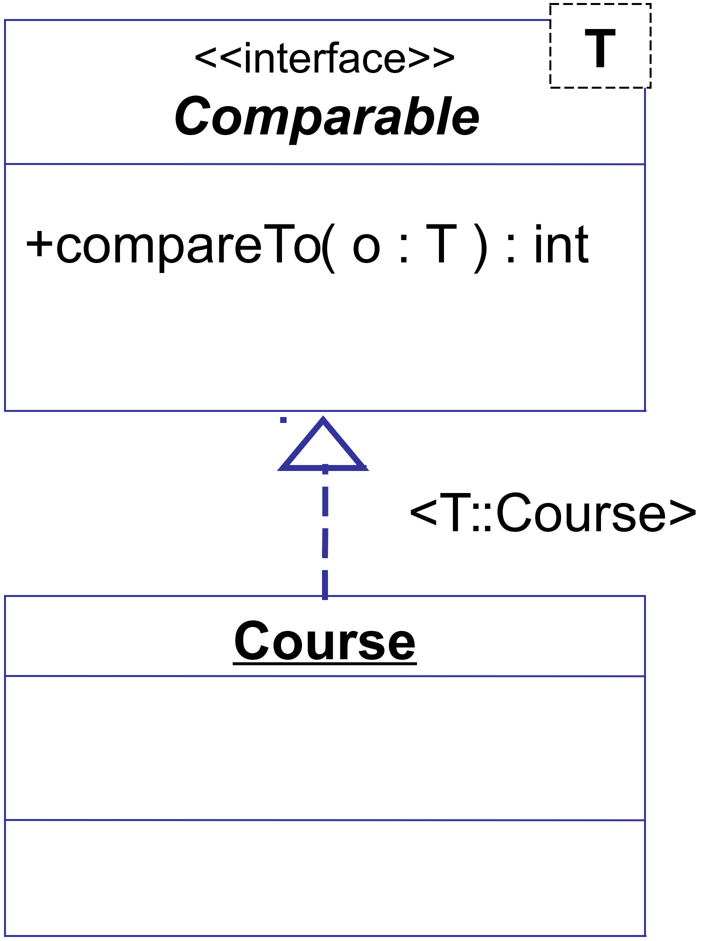
3..*

Point

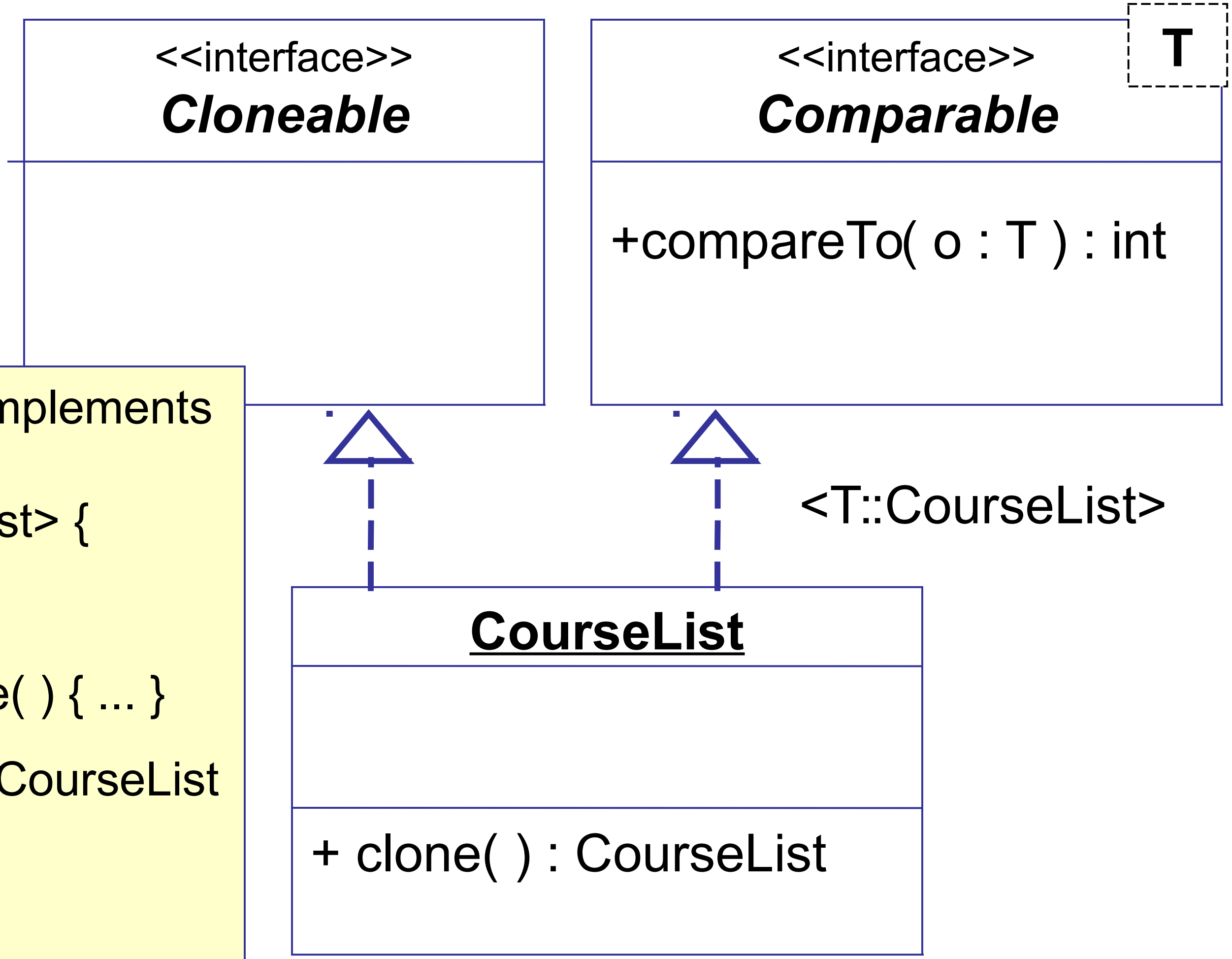
7. Implements



7b. Parameterized interface



7c. Implement multiple interfaces



<<interface>>
Cloneable

<<interface>>
Comparable

T

+compareTo(o : T) : int

<T::CourseList>

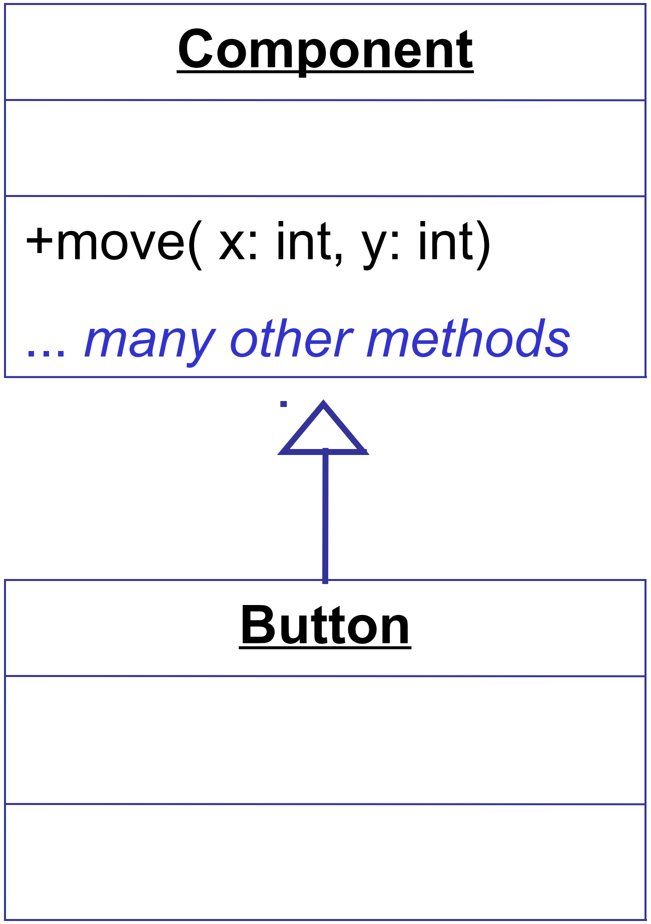
<u>CourseList</u>

+ clone( ) : CourseList

```
public class CourseList implements
   Cloneable,
   Comparable<CourseList> {

   ...

   public CourseList clone( ) { ... }

   public int   compareTo(CourseList
   o) {

            ...

   }
```

8. Inheritance

```
public class Button extends Component {


}
```



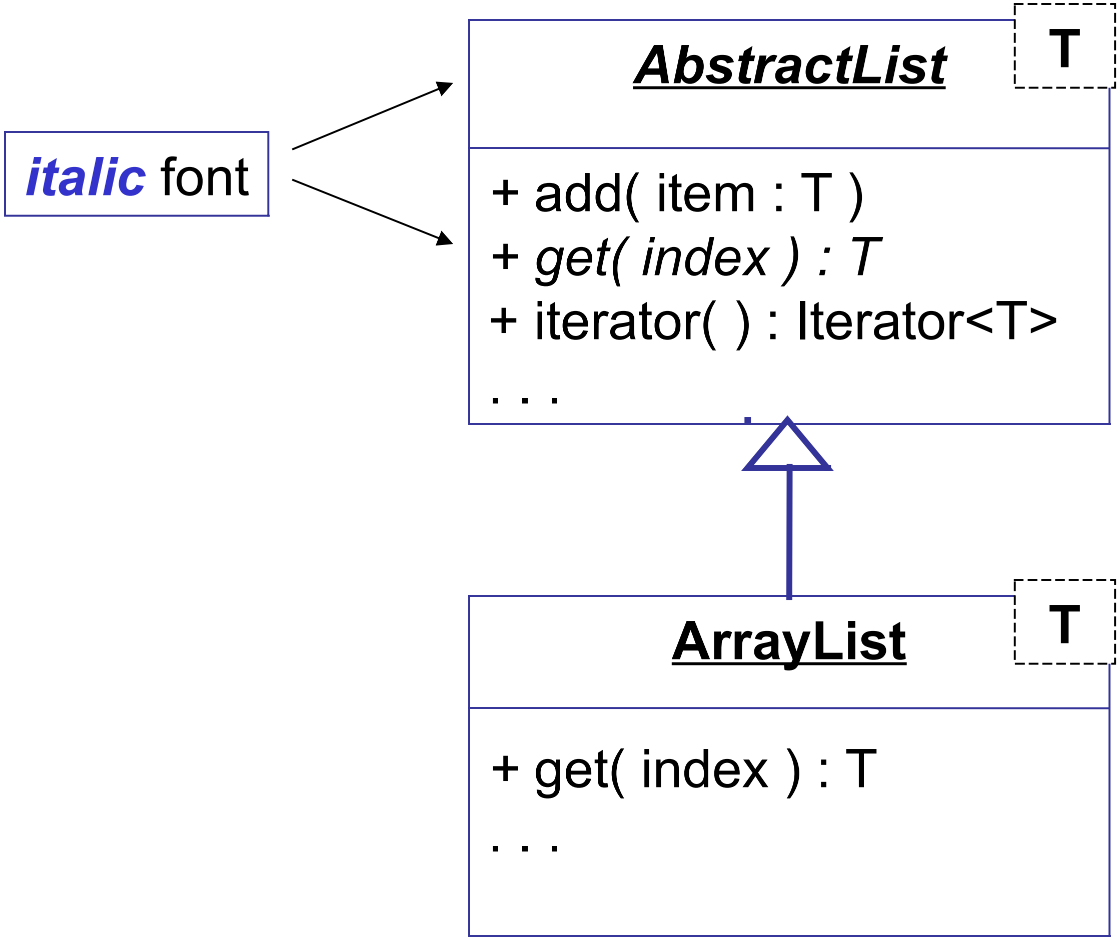
Component

+move(x: int, y: int)

... many other methods

Button

8b. What does *italic font* mean?



italic font

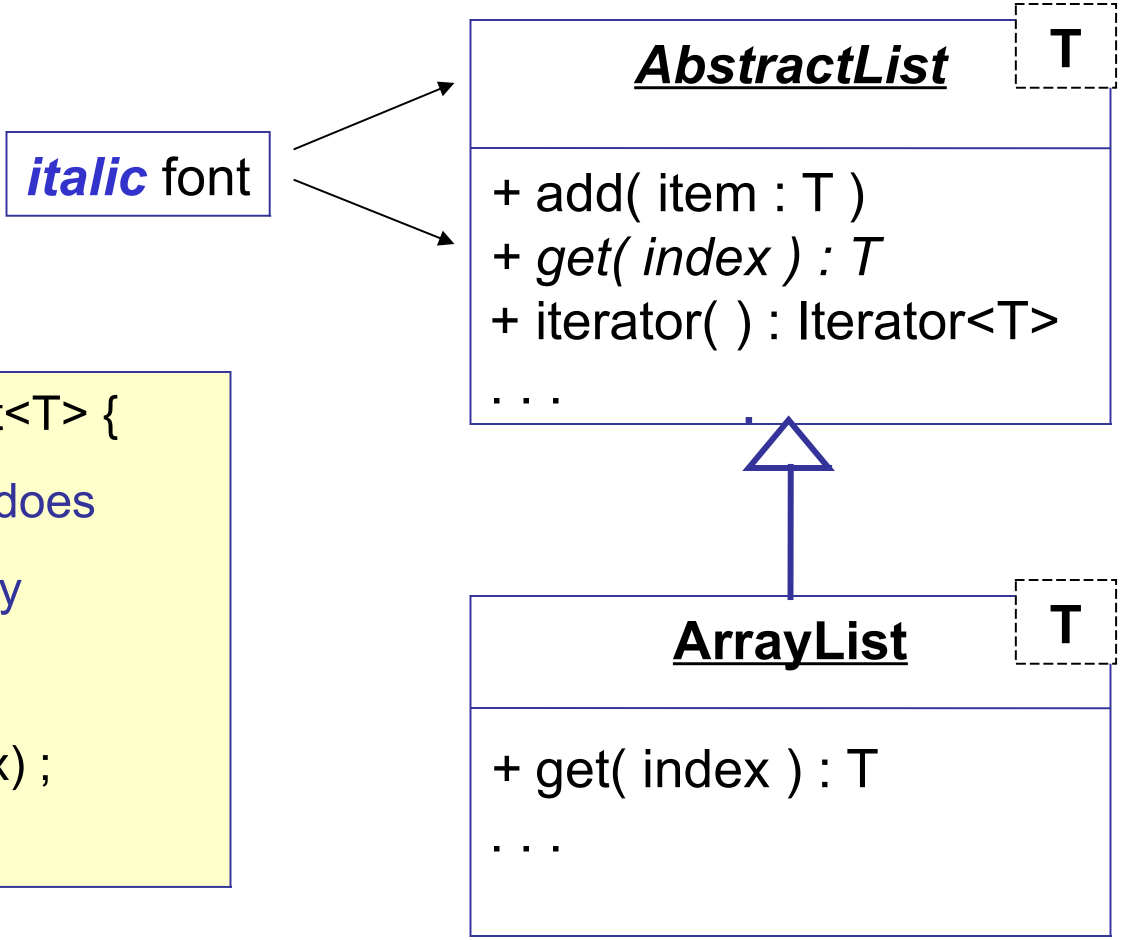
AbstractList **T**

+ add(item : T)
+ *get(index) : T*
+ iterator( ) : Iterator<T>
. . .

ArrayList **T**

+ get(index) : T
. . .

8b. Abstract class, abstract method



```
abstract class AbstractList<T> {
     /* ABSTRACT method does
     not have a method body
      */
     abstract T get(int index) ;
```

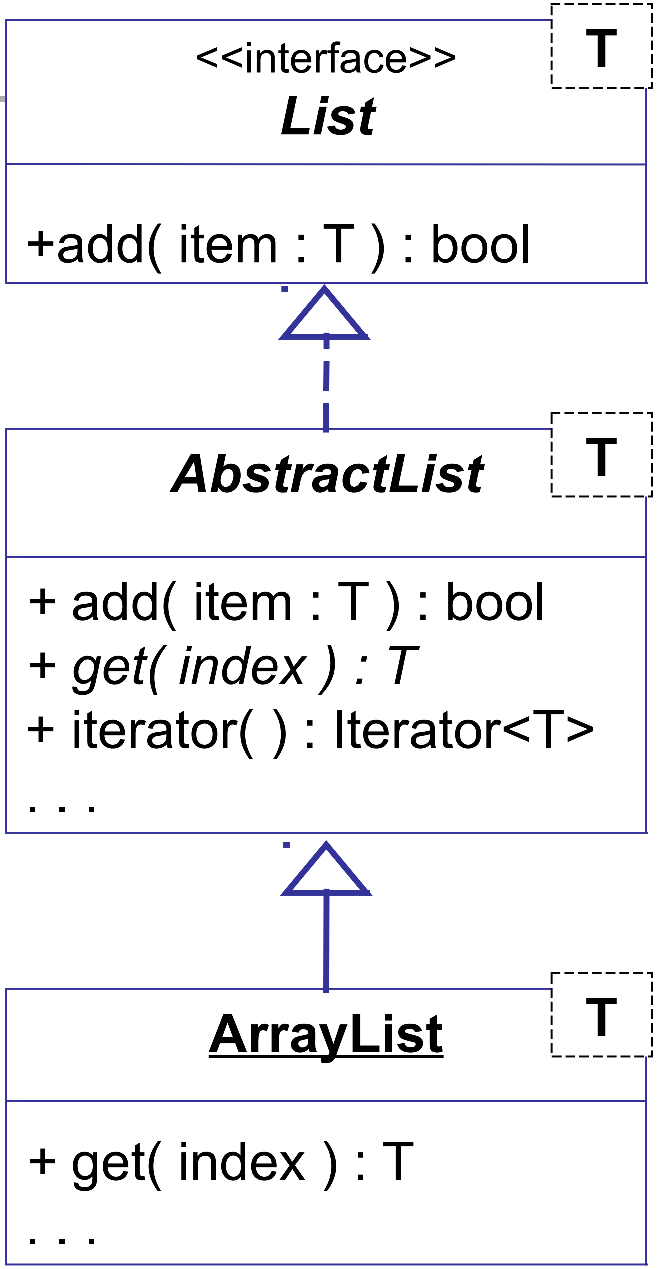
8c.

In Java:

1) *define* a reference (variable) for an ArrayList of **Course** objects

2) *create* an ArrayList of Course objects

```
/* 1. define a reference to a list of Course
   objects */

List<Course> arr;

/*  2. create the ArrayList */

arr = new ArrayList<Course>( );
```



T

<<interface>>

List

+add(item : T) : bool

T

AbstractList

+ add(item : T) : bool

+ *get(index) : T*

+ iterator( ) : Iterator<T>

. . .

T

ArrayList

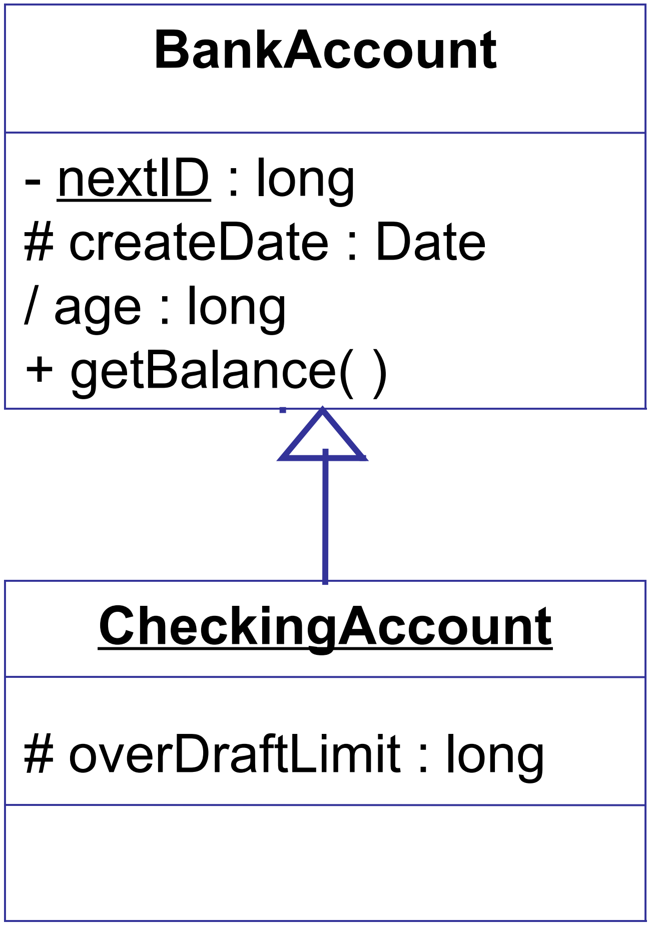
+ get(index) : T

. . .

9. Properties & Visibility

What do +, −, #, / mean?

To show package visibility: ~



BankAccount

- nextID : long
# createDate : Date
/ age : long
+ getBalance( )

CheckingAccount

# overDraftLimit : long

Interfaces

Most general use is:

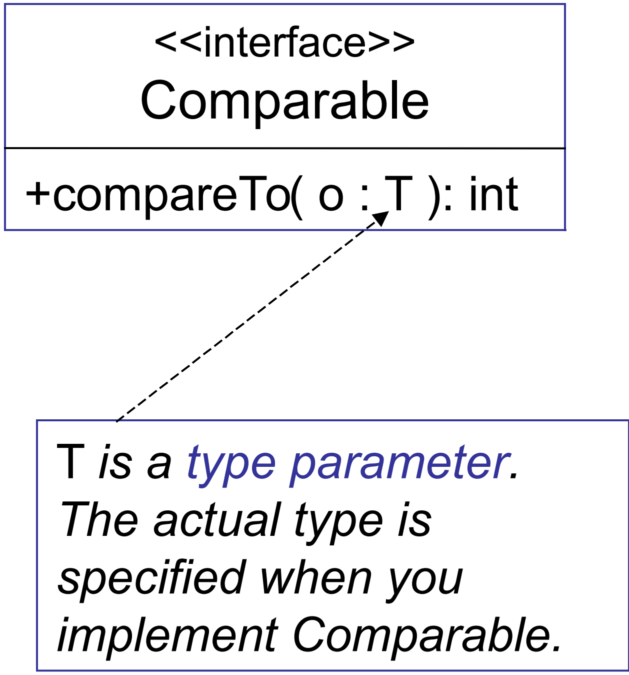
- Specify behavior that a type of object must have.
- *Separate the specification of a behavior from its implementation.*

Example: **`Comparable`**

define an ordering of objects

used by Arrays.sort, Arrays.search

each class can implement the way that it wants. All we need to know is that it specifies a relative ordering.



<<interface>>
Comparable

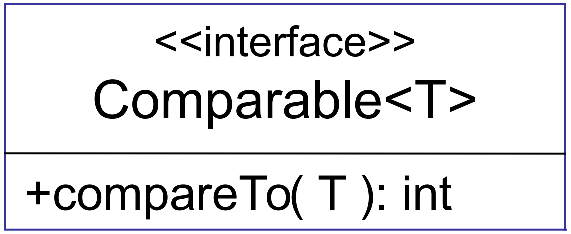
+compareTo(o : T): int

T *is a type parameter. The actual type is specified when you implement Comparable.*

Comparable Interface

Comparable - objects that can be put in some kind of order (compared to each other).

Examples are: numbers, String, Date.


<<interface>>
Comparable<T>
+compareTo(T): int

If you want to be able to sort objects,
then your class should implement Comparable.

```
/* coins can be compared by value */
public class Coin implements Comparable<Coin> {
   private int value;
   public int compareTo(Coin other) {
      if ( other == null ) return -1;
      return this.value - other.value;
   }
```

Meaning of **compareTo**

The return value has this meaning:

a.compareTo(b) < 0   a should come before b

a.compareTo(b) > 0   a should come after b

a.compareTo(b) = 0   a and b have same order

Usually:

a.compareTo(null) < 0   put nulls at end.

```
/* compare students by GPA.  Best student comes first
*/
public class Student implements Comparable<Student> {
  public int compareTo(Student other) {
     if ( other == null ) return -1;
     if ( this.getGPA() > other.getGPA() ) return -1;
     if ( this.getGPA() < other.getGPA() ) return -10;
     return 0;
  }
```

Uses of Comparable

- Sorting: Arrays.sort( ), Collections.sort( )
- Searching: Arrays.search( ), Collections.search( )

  `search` is *much faster* than `contains( )` *if the collection is sorted.*
- making decisions about things. "Which is better?"

Example

String implements Comparable, so we can sort and search an array of Strings.

```
String [] words = { "cat", "bird", "apple", "durian",
             "banana", "fig" };
Arrays.sort( words );
int k = Arrays.search( words, "banana" ); // k = 1
```

You can do this with an array of Coin or anything else that *implements Comparable*.

`Comparator` Interface (1)

There are 2 problems with *Comparable*:

1) a class can only have 1 compareTo method

2) it is part of the class, so we can't change it after the class is compiled.

What if compareTo( ) doesn't do what you want?

Example: sort a list of Strings, *ignoring case of letters*.

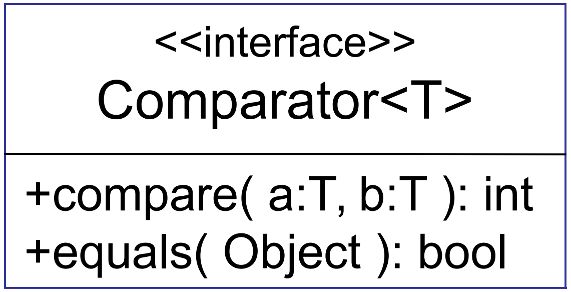
We want: `"apple", "Avacodo", "banana", "Cat"`

String.compareTo won't sort like this.

Comparator Interface (2)

Comparator defines a compare(a,b) method for comparing 2 objects.

Comparator is a separate class, so we can create as many as we want and whenever we want.


<<interface>>
Comparator<T>
+compare(a:T, b:T): int
+equals(Object): bool

compare(a, b) < 0	if a should come before b
compare (a, b) > 0	if a should come after b
compare(a, b) = 0	if a and b have same ordinal value
equals(Object)	used to test if two *Comparator* objects are the same (not used very much)

Comparator Example

Define a Comparator to sort Strings *ignoring the case of letters.*

```
/* compare 2 strings ignoring case */
public class StringComparator
        implements Comparator<String> {
   public int compare(String a, String b) {
     if ( a == null ) {
        if ( b == null ) return 0;
        else return 1;
     }
     if ( b == null ) return -1;
     // String class has method to do this
     return a.compareToIgnoreCase( b );
   }
```

Using Comparator

The sort and search methods of Arrays and Collections have another form that uses a *Comparator:*

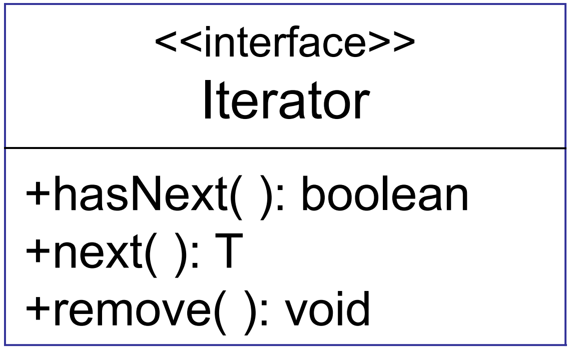
`Arrays.sort( array[ ], `*`comparator`*` )`

`Arrays.search( array[ ], target, comparator )`

```
String [] words = { "apple", "BIRD", "cat", "Durian",
             "banana", "Fig" };
// use the comparator class from previous slide
Comparator<String> comp = new StringComparator( );
// sort the array, ignoring case of letters
Arrays.sort( words, comp );
// find a word, ignoring case
int k = Arrays.search( words, "bird", comp ); // k = 2
```

`Iterator` Interface

`Iterator` - used to *iterate* over a collection of objects.



<<interface>>
Iterator

+hasNext( ): boolean
+next( ): T
+remove( ): void

`remove` *is not used very much. Some* iterators *don't implement it.*

`remove` *may throw an* OperationNotSupported *exception.*

Example:

```
/* Print all the coins in the purse */
List<Coin> coins = purse.getCoins(); // suppose we have
Iterator<Coin> iter = coins.iterator( );
while( iter.hasNext() ) {
   Coin c = iter.next( );
   System.out.println( c );
}
```

Iterator Pattern (1)

Why Iterator?

We can do the same thing like this:

```
/* Print all the coins in the purse */
List<Coin> coins = purse.getCoins(); // suppose we have

for(int k=0; k < coins.size(); k++) {
   Coin c = coins.get(k);
   System.out.println( c );
}
```

Problem:

coins.get(k) works for a List, but not for a Set or Stack...

So, our code depends on the type of collection we want to process!

Iterator Pattern (2)

Motivation:

Provide a way of visiting each element in a collection without knowing the structure or semantics of the collection.

Solution:

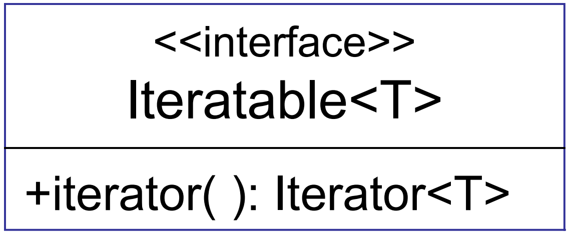
Define an *interface* (Iterator) that each collection can implement any way it wants.

Use the *interface* to visit each element of the collection.

```
// ask a purse to give us an iterator for its coins
Iterator<Coin> iter = purse.iterator( );
for(int k=0; k < coins.size(); k++) {
   Coin c = coins.get(k);
   System.out.println( c );
}
```

Iteratable Interface

Iteratable - defines how to get an iterator.



This solves the problem of "how can we get an iterator without knowing the structure of the collection"?

```
/* Every collection implements Iterable!
 * List, Set, Queue, Stack all implement it. */
Stack<Coin> coins = ... // suppose we have Stack of
Coin
Iterator<Coin> iter = coins.iterator( );
while( iter.hasNext() ) {
   Coin c = iter.next( );
   System.out.println( c );
}
```

`Iteratable` Interface (2)

`Polymorphism and Encapsulation:`

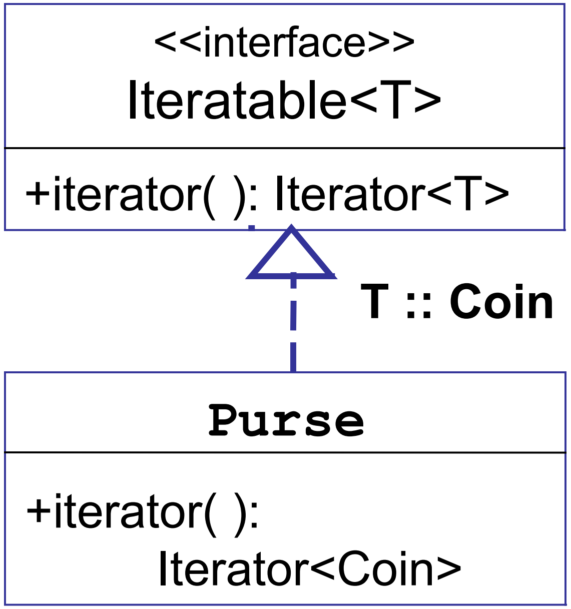
- every class can implement **`Iterable`** the way it wants.
- every class can define its own kind of **`Iterator`**.
- the class can *hide* (encapsulate) details of how its objects are stored. The only thing it shows us is the **`Iterator`**.

This solves the problem of "how can we get an iterator without knowing the structure of the collection"?

Iteratable Interface (3)

Example

- we need a way to view the Coins in a Purse
- the Purse doesn't want us to *change* the Coins while we are viewing them (if it gave us a List, we could change things in the List)
- we don't want to *expose* how the Coins are stored in the Purse.



Solution:

- provide an Iterator for viewing Coins
- provde an iterator() method to get the Iterator

Example Implementation

Implement Iteratable<Coin> in Purse

- since the coins are in a List and **`List`** already implements **`Iterable`**, its pretty easy.

```
/**  A Purse contains some coins, but don't ask how.
 *   Its a secret (encapsulation).
 */
public class Purse implements Iterable<Coin> {
   private List<Coin> coins;
   /**
    * create an iterator for viewing the coins.
    * @return an Iterator of coins in the purse
    */
   public Iterator<Coin> iterator( ) {
      return coins.iterator(); // List does it
   }
}
```

A Mystery...

- `List.iterator()` returns an object.
- we know this object implements the `Iterator` interface .

The Mystery

- **what *class* does this Iterator belong to?**
- **how does it work?**

No one knows.

```
List<Coin> list = new ArrayList<Coin>();
list.add( new Coin(5) ); ...
Iterator<Coin> iter = list.iterator();
what class does iter belong to?  We don't know
```

What are the Key Uses of Inheritance?

Name 3 key uses of inheritance:

1. *Factor out* common behavior (avoid duplicate code)
2. *Specialize* - redefine (override) a behavior
3. *Extend* - add new behavior to a base type

Inheritance versus Interface

Advantage of Inheritance

- base class provides methods that all subclasses can use (code re-use)
- base class defines data members for subclasses

Advantage of Interface

- separate the *specification of a behavior* from the *implementation*
- any class can implement: no fixed hierarchy
- doesn't consume the implementing class's choice of parent class
- a class can implement several interfaces

Draw a Sequence Diagram

Draw a sequence diagram showing what happens when `saveMoney()` is called.
We don't care who the caller is; show caller as a "found" message.

```
class PurseUI {
   private Purse purse;  // a coin purse holds money
   void saveMoney() {
      Money money = new Money( 20, "Baht");
      purse.insert( money );
   }
}
```