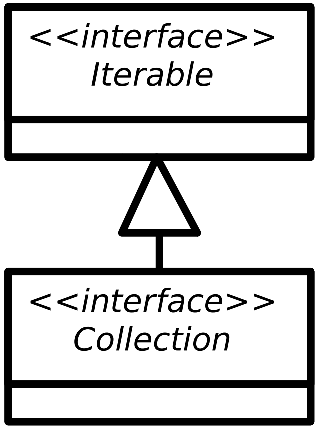
Java Class Library

- Java the platform contains around 4,000 classes/interfaces
  - Data Structures
  - Networking, Files
  - Graphical User Interfaces
  - Security and Encryption
  - Image Processing
  - Multimedia authoring/playback
  - And more…

- All neatly(ish) arranged into packages (see API docs)

Java's Collections Framework

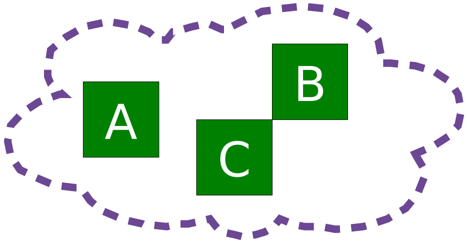


- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

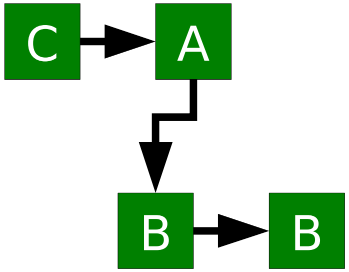
- The Collections framework has two main interfaces: Iterable and Collections. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

Major Collections Interfaces I

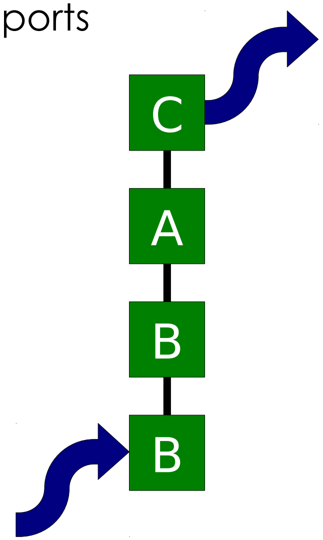
- <<interface>> Set
  - Like a mathematical set in DM 1
  - A collection of elements with no duplicates
  - Various concrete classes like TreeSet (which keeps the set elements sorted)



- <<interface>> List
  - An ordered collection of elements that may contain duplicates
  - ArrayList, Vector, LinkedList, etc.

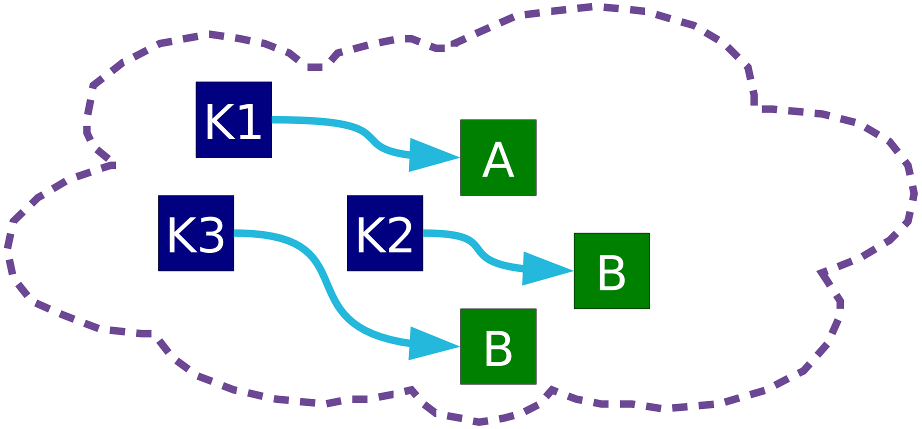


- <<interface>> Queue
  - An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
  - PriorityQueue, LinkedLIst, etc.



Major Collections Interfaces II

- <<interface>> Map
  - Like relations in DM 1, or dictionaries in ML
  - Maps key objects to value objects
  - Keys must be unique
  - Values can be duplicated and (sometimes) null.



Iteration

- for loop

```
LinkedList list = new LinkedList();
...
for (int i=0; i<list.size(); i++) {
    Object next = list.get(i);
}
```

- foreach loop (Java 5.0+)

```
LinkedList list = new LinkedList();
...
for (Object o : list) {

}
```

list = new LinkedList<Integer>

for(Integer i : list)

Iterators

- What if our loop changes the structure?

```
for (int i=0; i<list.size(); i++) {
    If (i==3) list.remove(i);
}
```

- Java introduced the Iterator class

```
Iterator it = list.iterator();

while(it.hasNext()) {Object o = it.next();}

for (; it.hasNext(); ) {Object o = it.next();}
```

- Safe to modify structure

```
while(it.hasNext()) {
    it.remove();
}
```

Collections and Types I

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
     Object o = it.next();
     Integer i = (Integer)o;
}
```

- The original Collections framework just dealt with collections of Objects
  - Everything in Java “is-a” Object so that way our collections framework will apply to any class
  - But this leads to:
    - Constant casting of the result (ugly)
    - The need to know what the return type is
    - Accidental mixing of types in the collection

Collections and Types II

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person(“Bob”));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

Going to fail for the second element! (But it will compile: the error will be at runtime)

Java Generics

- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// Make a TreeSet of Integers
TreeSet<Integer> ts = new TreeSet<Integer>();

// Add integers to it
ts.add(new Integer(3)); ✓
ts.add(new Person(“Bob”));

// Loop through
iterator<Integer> it = ts.iterator();
while(it.hasNext()) {
     Integer i = it.next();
}
```

Won't even compile

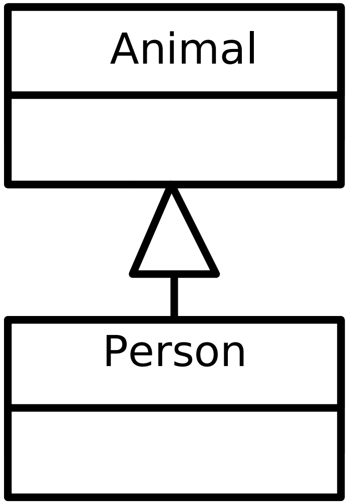
No need to cast :-)

Generics Declaration and Use

```
public class Coordinate <T> {
    private T mX;
    private T mY;

    public Coordinate(T x, T y) {
        mX=x; mY=y;
    }

    public T getX() { return mX; }
    public T getY() { return mY; }
}

Coordinate<Double> c =
    New Coordinate<Double>(1.0,1.0);

Double d = c.getX();
```

Think of this as getting replaced

Generics and SubTyping



Animal

Person

```
// Object casting
Person p = new Person();
Animal o = (Animal) p;
```

✓

```
// List casting
List<Person> plist = new LinkedList<Person>();
List<Animal> alist = (List<Animal>)plist;
```

will not compile

So a list of **Persons** is a list of **Animals**, yes?

alist.add(new Hippo());

Section: Comparing Java Classes

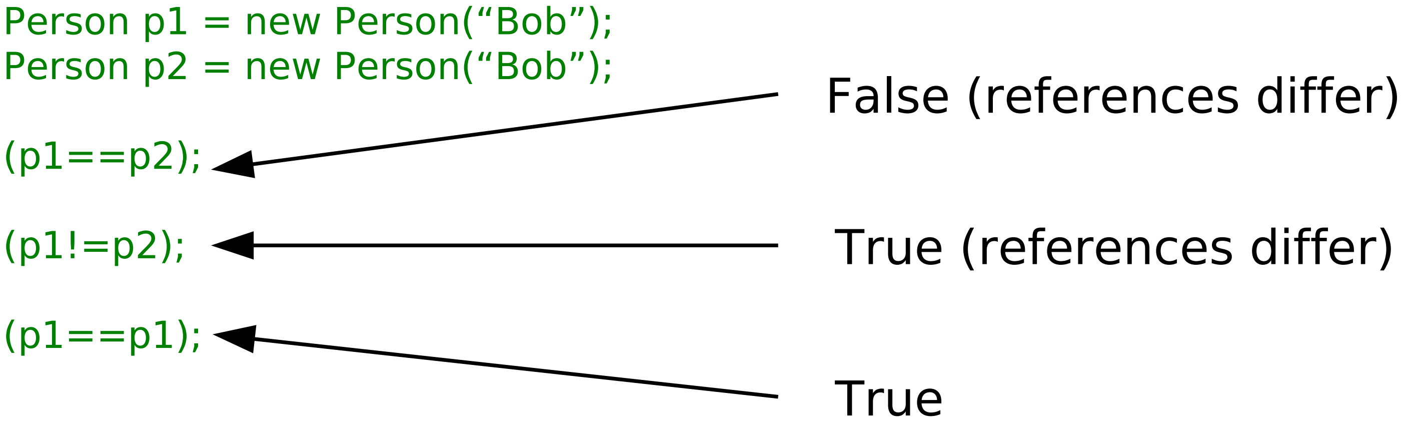
Comparing Primitives

>	Greater Than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to

- Clearly compare the value of a primitive
- But what does (ref1==ref2) do??
  - Test whether they point to the same object?
  - Test whether the objects they point to have the same state?

Option 1: a==b, a!=b

- These compare the *references directly*



```
Person p1 = new Person(“Bob”);
Person p2 = new Person(“Bob”);

(p1==p2);

(p1!=p2);

(p1==p1);
```

False (references differ)

True (references differ)

True

Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
  - Returns boolean, so can only test equality
  - Override it if you want it to do something different
  - Most (all?) of the core Java classes have properly implemented equals() methods

```
public EqualsTest {
    public int x = 8;

    public boolean equals(Object o) {
        EqualsTest e = (EqualsTest)o;
        return (this.x==e.x);
    }

    public static void main(String args[]) {
        EqualsTest t1 = new EqualsTest();
        EqualsTest t2 = new EqualsTest();
        System.out.println(t1==t2);
        System.out.println(t1.equals(t2));
    }
}
```