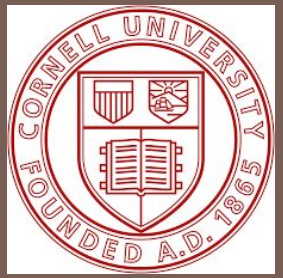
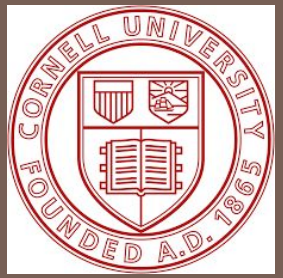
Object-oriented programming and data-structures



CS/ENGRD 2110 SUMMER 2018



Lecture 5: Exceptions and advanced typing

http://courses.cs.cornell.edu/cs2110/2018su

Lecture 4 Recap

- Finished introducing main OOP principles and how they are instantiated in Java
  - Modularity
  - Inheritance
  - Abstraction
  - Polymorphism

Lecture 5

- Focus on specific Java features that we feel you should know
  - Parametrised Types & Generics
  - Enumerations
  - Exceptions & Testing
  - Java Collections
  - Cloning

- Last topic before we declare you a Java Expert, and move on to datastructures

Limiting input options

- Recall our Date class
  - field month should really only contain valid months, not arbitrary integers or strings
  - how do we limit the type of inputs that can be passed in?

```
class Date {
        String month;
        int day;
        int year;
}
```

Enumeration

- Most OOP languages address this problem with **enums**
  - An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.
- Enums are most often used in **switch statements**, where different actions are taken for each value of the enum

```
class Date {
        Month month;
        int day;
        int year;
}


 enum Month {
        JANUARY,
        FEBRUARY,
        MARCH, ...
 }

 Month month = …;
 switch(month):
        case JANUARY:
        case FEBUARY:
```

Generics (very briefly)

- It occasionally makes sense for a class to store groups of fields of different types
  - ex: a 2D vector stores x and y coordinates. x/y could be integers, doubles, floats, etc.
  - But operations on them would remain the same

```
class 2DIntVector {
        int x;
        int y;
}
```

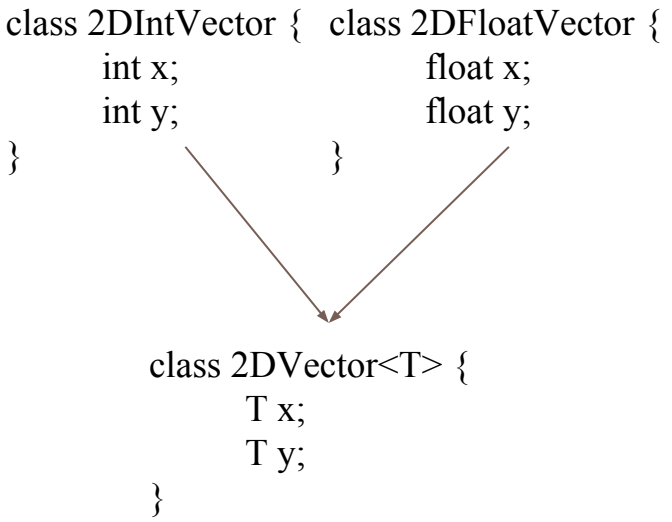
```
class 2DFloatVector {
        float x;
        float y;
}
```

- Generics allow us to **parametrise** a class by a specific type. Avoid having to rewrite multiple classes

- More in Assignment 3….

Generics (very briefly)

- It occasionally makes sense for a class to store groups of fields of different types
  - ex: a 2D vector stores x and y coordinates. x/y could be integers, doubles, floats, etc.
  - But operations on them would remain the same



- Generics allow us to **parametrise** a class by a specific type. Avoid having to rewrite multiple classes

- More in Assignment 3….

What if things go wrong

- So far, we've assumed that every input was correct, and that we wrote perfect code (ahem)
  - this is rarely true!

- Three types of errors
  - **syntactic error** -> code doesn't compile, etc. Usually fairly easy to spot
  - **logical errors** (ie: bugs) -> your function doesn't do what you thought it does
  - **external errors** -> external inputs that we can't control

Minimising Bugs - Unit Testing

- Modular and systematic testing of every method/constructor
  - Yes, it's boring and may sometimes seem stupid, but necessary
  - OOP encourages you to make each class independent of others, so should also be possible to test independently of others.

- Java provides two main tools to do that
  - assert statement -> program will crash if assertion is false
    - `assert(x==5);`
  - JUnit, a framework for writing repeatable test
    - Tutorial in the Java Hypertext

Dealing with errors

- **Definition** Errors are expected exceptional behaviour (ex: the file is corrupted or does not exist)

- Three main ways
  - via return codes
  - via deferred error-handling
  - via exceptions

- Different languages prefer different styles
  - Ex: C prefers return codes, Java Exceptions

Error Codes

- Traditional way of handling errors is to return value that indicates outcome of function
  - 0 for success, 1 for failure for reason X, 2 for failure for reason Y,, etc.

```
int setValueArray(int index, int[] array, int value) {
        if (index >= array.length) return -1
        else {
                array[index] = value;
                return 0 ;
        }
}
```

Error Codes

- Problems
  - Have to keep checking what the return values are meant to signify
  - The actual result can't actually be returned in the return type
  - Can ignore the return value

- How would you implement a `getValueAtIndex` function with error codes?

Deferred Error Handling

- Set some state in the system that needs to be checked explicitly for errors
  - C has a field “errno” that is used by certain functions to store the error code of the function
- Allows the function to return a value
- Still requires checking **errno** everytime

```
int main () {
        FILE * fp;
        fp = fopen ("filedoesnotexist.txt", "rb");
        if (errno == 0) {
                fprintf(fp, …):
        } else {
                fprintf(stderr, "Value of errno: %d\n", errno);
        }
        ...
        }
```

Exceptions

- Java privileges **exceptions**

- **Definition**: an exception is an object that can be **thrown** by a method when an error occurs. The exception is **caught** by a handler.

Exceptions

- Java privileges **exceptions**
- **Definition**: an exception is an **object** that can be **thrown** by a method when an error occurs. The exception is **caught** by a handler.
- Java exceptions have thrown with the following syntax:
  - Methods that can throw an exception must be marked as such
    - void setDay(int day) throws Exception {

```
if (day < 0 || day > 31) throw new Exception("Illegal Day");
else this.day = day;
System.out.println("Day set");
}
```

Why not use an assert here?

Throwing an exception is an exit point of the function. Last line not printed if day >31

Exceptions - Try/Catch

- Exceptions are **handled** in a **try/catch** block.
- Exception is **raised** in a **try** block and **caught** in a catch block.
  - Catch block is referred to as a **handler**
- Can be multiple **handlers** for a given **try** block, for different exception types.

```
try {
        setDay(day);
}
catch (IllegalDayException e) {
        // NEVER LEAVE THIS BLANK
}
catch (IllegalYearException e) {
        ...
}
```

Exceptions - Finally

- Exceptions break the control flow of the program as prevent code that follows the exception to be executed.

- If need to write cleanup code regardless of whether the exception is thrown, wrap it in a **finally** block.

```
try {
        FileReader file = new FileReader(fileName);
        file.read();
        file.close();
}
catch (IOException e) {
        ...
}
finally {
        if (file!=null) file.close();
}
```

Checked exceptions

- Java supports two types of exceptions: **checked** and **unchecked**

- Checked exceptions must be handled
  - Must be specified in the method signature (**throws**)
  - Code won't compile unless provide a **handler** for each thrown exception
  - Usually used for "expected" errors (ex: file does not exist, IO bug, etc.)

- Unchecked exceptions do not need to be handled. Arise at runtime. Will crash program. Arise because of programming bugs usually
  - Ex: NullPointerException.

Exceptions are classes too!

- Exceptions are defined as regular classes in Java
  - Possible to define specialised exceptions
  - All exceptions must extend the **Exception** class
    - Q: why is Exception not an interface?

```
class MyNewException extends Exception {
        int illegalValue;
        MyNewException(int value, String msg) {
                super(msg);
                this.illegalValue = value;
        }
}
```

Java Class Library

- Java isn't just a language, it's a platform with thousands of classes/interface with
  - Data Structures
  - Networking/Files
  - GUI/Multimedia/playback (!)
  - Security
  - Image Processing
  - Concurrency

- You should get into the habit of searching the javadoc to find the appropriate package

Collections

- Will take brief look at collections
  - Very useful to use in most programs
  - Demonstrate the use of **interfaces**

- **Definition**: grouping of objects that can be iterated over

- Collections implement two main interfaces**: Iterable**, and **Collection**
  - (look it up!)
  - All collections therefore support a common set of operations

Collections

- If all support the same operations, why have more than one collection?
  - Collections implement multiple algorithm, that have different performance characteristics (we'll see later in class)
  - STL containers in C++ is closest equivalent
- 4 main ones:
  - sets/lists/queues/maps
- Lookup the API!
  - https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

Sets (implements Set<E>)

- A collection of elements without duplicates
  - Implementing classes:
    - TreeSet
      - Objects are sorted in order
      - Fast to retrieve contiguously sorted items
    - HashSet:
      - Objects are unordered
      - Supports fast addition/retrieval of single elements

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Interface Set<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:

AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

---

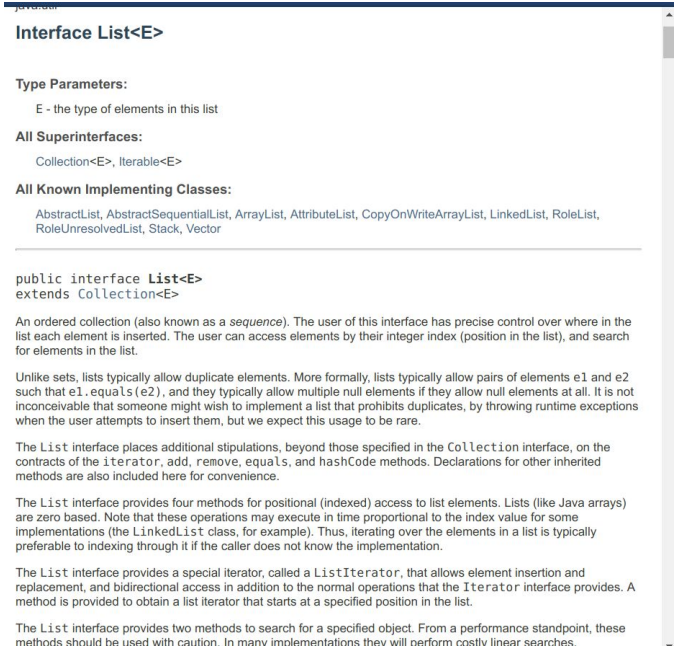
```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The Set interface places additional stipulations, beyond those inherited from the

Lists (implements List<E>)

- An ordered collection of elements (may contain duplicates)
  - Linked List: linked lists of elements, store pointer to the head and the tail of the list. Can grow dynamically
  - ArrayList: array of elements. Efficient to access, but costly to grow. Costly to insert elements in the middle of the list.



Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

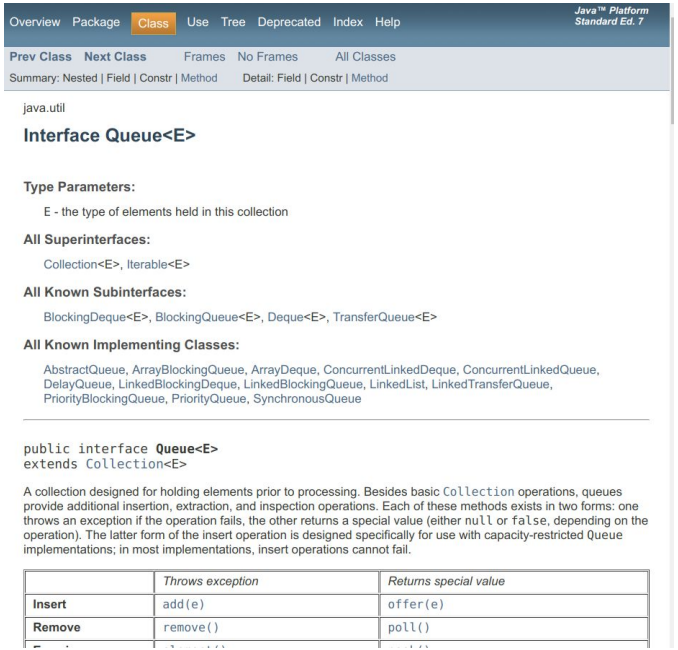
The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

```
LinkedList<Integer> l = new LinkedList<Integer>();
l.add(1);
l.get(); // return 1
...
```

Queue (implements Queue<E>)

- Ordered collection of elements (may contain duplicates) that supports removal from head only
  - LinkedList: yes, it's both a list and a queue!
  - PriorityQueue: sort elements according to priority so that higher priority elements are at head of queue



Overview Package Class Use Tree Deprecated Index Help — Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface Queue<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

```
public interface Queue<E>
extends Collection<E>
```

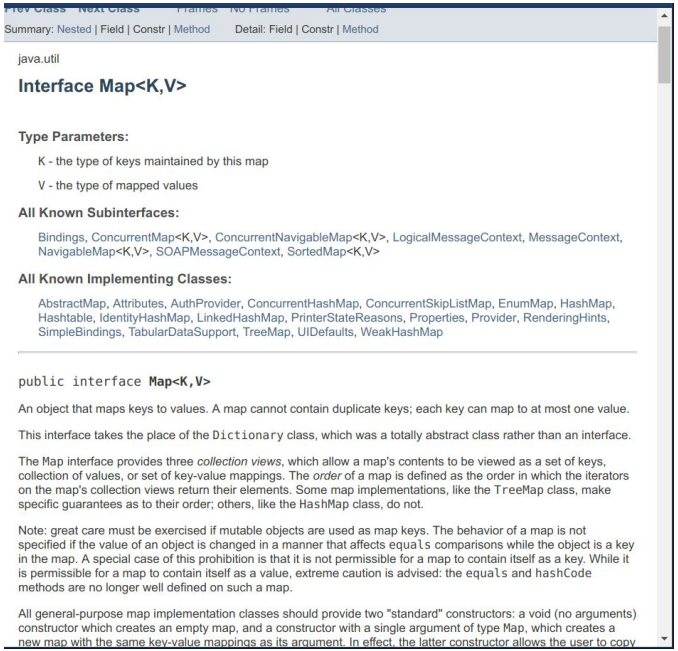
A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations; in most implementations, insert operations cannot fail.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`

```
LinkedList<Integer> l = new LinkedList<Integer>();
l.offer(1);
l.poll(); // return 1
...
```

Maps (implements Map<K,V>)

- Maps keys to values
- Keys must be unique but values can be duplicated or null
- Think of dictionaries in Python (for ex) or Matlab
  - TreeMap: keys kept in order
    - Fast to lookup contiguously sorted items
  - HashMap: keys not sorted in order
    - Fast lookup/insertion of single item



Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy

Iterating over a collection

- Through a for loop

```
Set<Integer> mySet = new TreeSet<Integer>();
for (Integer i: mySet) {
        System.out.println(i);
}
```

- Through an iterator
  - Key benefit of an iterator: safe to remove elements to the collection

```
Set<Integer> mySet = new TreeSet<Integer>();
Iterator<Integer> it = mySet.iterator();
while (it.hasNext()) {
        Integer i = it.next();
}
while (it.hasNext()) {
        it.remove();
}
```

Manipulating objects - Equality

- Collections requires testing whether objects are equal, or sorting objects
- It is straightforward to compare primitive types
  - >, <=, ==, !=., <, <=
- Objects require more care
  - == on objects tests **reference equality:** checks whether point to same object
- We would like a way to compare objects whose **state is identical**

Equals() method

- Recall that every class extends the **Object** class
- Object class introduces an **equals()** method
  - default implementation just does reference equality
- To test for value equality, need to **override** the equals method

```
     * <p>
     * The equals method for class <code>Object</code> implements the most
     * discriminating possible equivalence relation on objects; that is,
     * for any reference values <code>x</code> and <code>y</code>, this
     * method returns <code>true</code> if and only if <code>x</code> and
     * <code>y</code> refer to the same object (<code>x==y</code> has the
     * value <code>true</code>).
     *
     * @param   obj   the reference object with which to compare.
     * @return  <code>true</code> if this object is the same as the obj
     *          argument; <code>false</code> otherwise.
     * @see     java.lang.Boolean#hashCode()
     * @see     java.util.Hashtable
     * @since   JDK1.0
     */
    public boolean equals(Object obj) {
        return (this == obj);
    }

    /**
     * Creates a new object of the same class as this object. It then
     * initializes each of the new object's fields by assigning it the
     * same value as the corresponding field in this object. No
     * constructor is called.
     * <p>
     * The <code>clone</code> method of class <code>Object</code> will
     * only clone an object whose class indicates that it is willing for
     * its instances to be cloned. A class indicates that its instances
     * can be cloned by declaring that it implements the
     * <code>Cloneable</code> interface.
     *
     * @return     a clone of this instance.
     * @exception  CloneNotSupportedException  if the object's class does not
```

Equals() method

- Recall that every class extends the **Object** class
- Object class introduces an **equals()** method
  - default implementation just does reference equality
- To test for value equality, need to **override** the equals method

```
class Person {
        private String name;
        private Date dob;
        private String netId;

        @Override
        public boolean equals(Object o) {
                If (o instanceof Person) {
                        Person p = (Person) o;
                        return p.netId.equals(o.netId);
                }
                else return false;
        }
}
```

Manipulating objects - Comparable

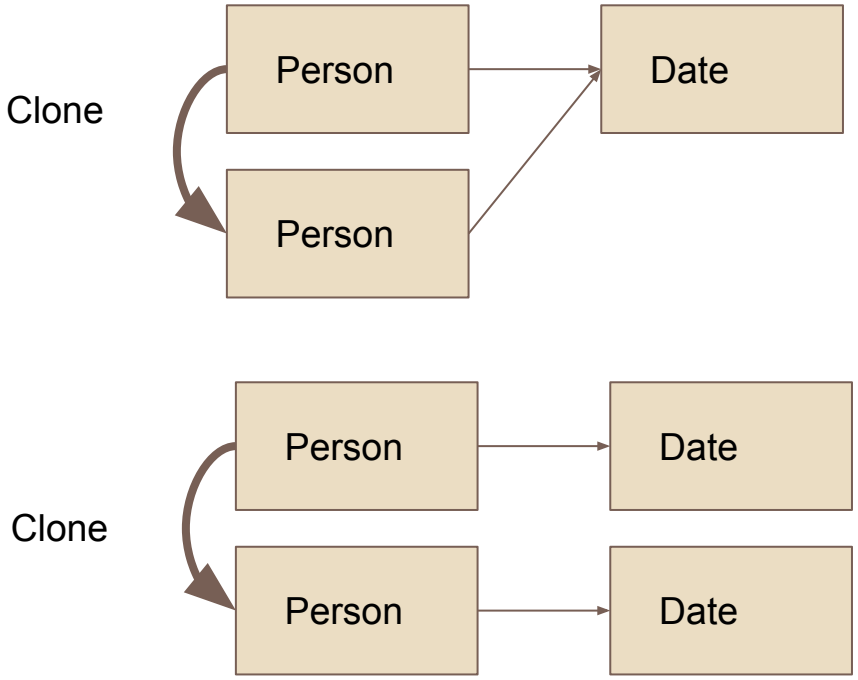
- Sometimes equality is not enough: many collections require sorting
- Objects that are **comparable** implement the interface **Comparable<T>**
  - Pay attention to the use of an interface! Many objects that have nothing to do with each other can all have the **sortable** functionality. It would not make sense to use an abstract class in this case
- Comparable Interfaces allow you to define **greater than/smaller than/equal** functionality
  - Must implement method **int compareTo(T obj)**
    - Returns <0 if smaller, then obj, >0 if greater, 0 if equals

Manipulating objects - Cloning

- Recall how immutable classes? To make a class immutable, need to **copy** mutable objects before assigning them to a field
- Java provides a mechanism to do that: the **clone()** method in the **Object** class.
- Other languages provide what is called a **copy constructor**

Manipulating objects - Cloning

- Distinguish between **deep** and **shallow** cloning
  - **Shallow cloning:** makes a copy of the object, does not change its fields.
  - **Deep cloning:** makes a copy of the object, including all objects that it has as fields (recursively)



Clone() method

- Like equals(), must **override** the **clone** method
- Must also implement the **cloneable** interface
  - But cloneable interface is empty!
  - It is a **marker interface**
    - marker interfaces are empty interfaces used to **label** classes for the compiler
- Clone is quite ugly, unfortunately, copy constructors are problematic too (for inheritance)

```
class Person {
        private String name;
        private Date dob;
        private String netId;

        @Override
        public Person clone() throws … {
                Person person = super.clone();
                Person.dob = dob.clone();
                return person;
        }
}
```

You are now Java experts!

- This is almost all the Java that we will teach you in this course
- Will see a few last things in the remainder of class

- Now will begin focusing on datastructures

References in JavaHyperText

enumeration

exception

assert

error handling

collections

comparable

cloning

equals