Generics

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 10

A Simple Component

- Client-side view: Pencil

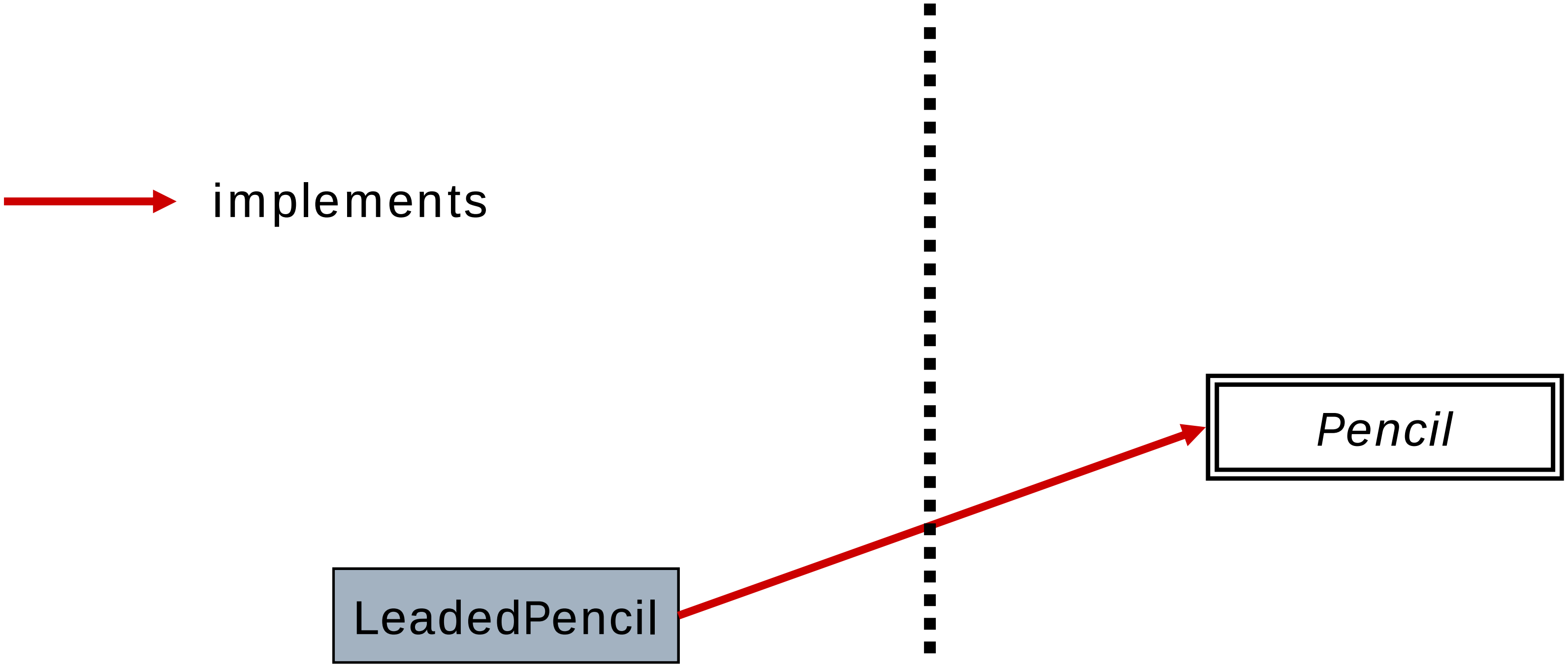
```
interface Pencil {
   String toString();
   void setColor(Colors newColor);
   void sharpen(int remove);
}
```

- Implementer's view: LeadedPencil

```
class LeadedPencil implements Pencil {
   private static final int STD_LENGTH = 10;
   private Colors color;
   private int length;
   . . . etc . . .
}
```

- See code listings for full documentation

Pencils



Background

- Methods are parameterized by the *values* of their formal arguments

```
void enableLaunch (boolean go) { … }
```

  - In a sense, there are 2 enableLaunch()'s:
    - one where go begins with value true
    - one where go begins with value false
  - *Could* define enableLaunchT(), enableLaunchF()

```
boolean isEven (int i) { … }
```

  - In a sense, there are 4,294,967,296 versions of isEven() (half return true, half return false)
  - *Could* define isEven0(), isEven1(), isEven2(), …

```
void println (String s) { … }
```

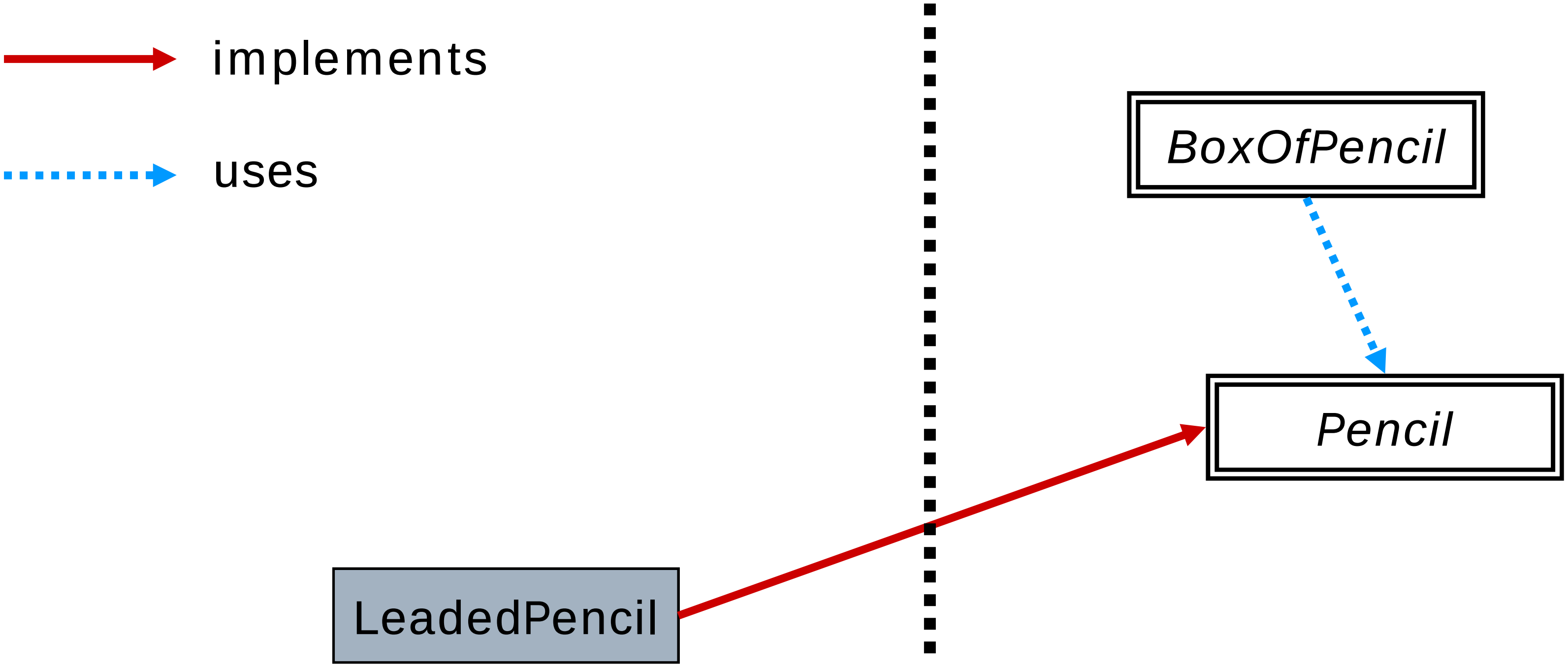
  - In a sense, there are ?? versions of println()

Motivation: Using Components

- Consider a box that holds a pencil
  - See BoxOfPencil.java
  - Box contains at most one pencil
  - Methods: size, contains, insert, removeAny
- Aside: Notice "coding to the interface"
  - Method signatures contain interface types

    ```
    boolean contains(Pencil target)
    void insert(Pencil item)
    Pencil removeAny()
    ```
  - Specifications also contain this type
- Recall: Declared vs Dynamic type
  - The dynamic type of these arguments and return values will be a reference to an instance of a class that *implements* Pencil (eg LeadedPencil)

Box of Pencils



implements

uses

BoxOfPencil

Pencil

LeadedPencil

Using a Different Component

- Now consider a box that holds a string
  - See BoxOfString.java
- (Aside: Is it coded to the interface?)
- These two class definitions differ *only* in:
  - The argument type of contains()
  - The argument type of insert()
  - The return type of removeAny()
  - The types mentioned in specifications
- All the rest is identical!
- BoxOfPencil and BoxOfString are like two instantiations of a generic class definition
  - Parameterized by *type* (not value)

Example: Generic Box Interface

- Declaration

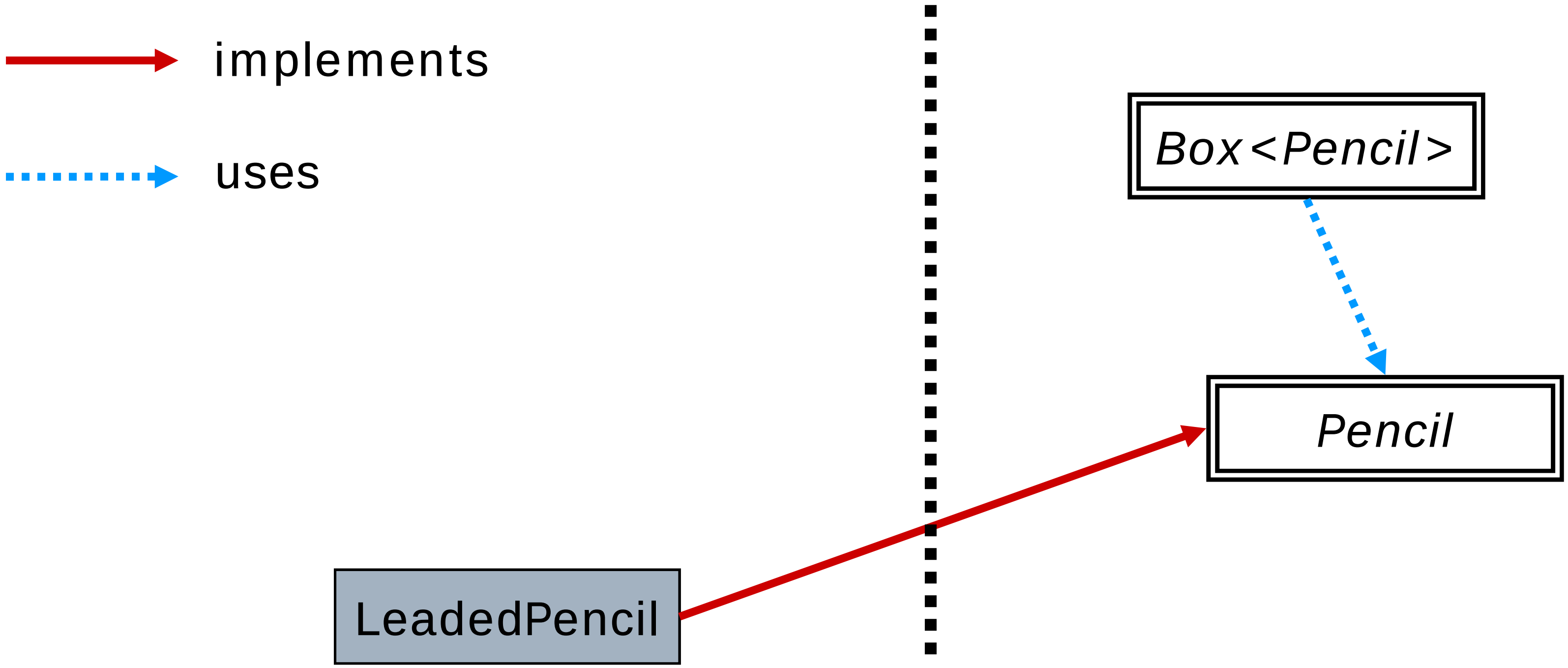
```
interface Box<T> { . . . }
```

- In body of interface declaration, T can now be used as a type

```
boolean contains(T target)
void insert(T item)
T removeAny()
```

- See Box.java
- Vocabulary:
  - T is a *type variable/parameter, or a naked type*
  - Box (ie without < >'s) is called a *raw type*

Box of Pencils



→ implements

⇢ uses

Box<Pencil>

Pencil

LeadedPencil

Example: Generic Implementation

- Declaration

```
class PlasticBox<T> implements Box<T> {
  . . .
  PlasticBox() { . . . }
}
```

- In body of class definition, T can now be used as a type
  - In fields

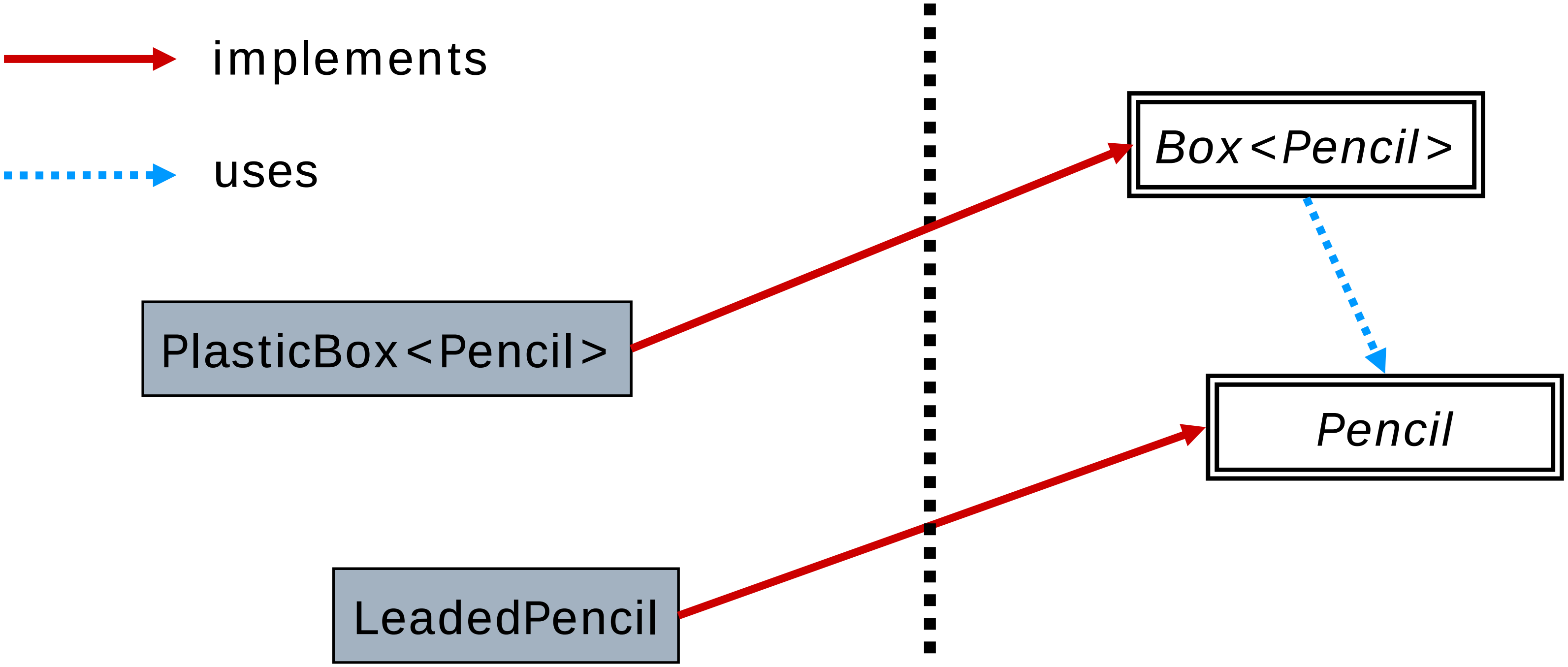
```
private T value
```

  - In methods

```
public void insert (T item)
```

- See PlasticBox.java
  - Note: Name of constructor in class definition is PlasticBox(), not PlasticBox<T>()

Box of Pencils



Example: Client Use of Generic

□ To use generic type: classname<type>

□ Usual rules of coding to the interface apply

```
Box<Pencil> bp = new PlasticBox<Pencil>();
bp.insert(new LeadedPencil());
Pencil p = bp.remove();

// the following are all errors...
String s = bp.remove();
LeadedPencil p2 = bp.remove();
Box<Pencil> bp2 = new PlasticBox<String>();
Box<Pencil> bp3 = new Box<Pencil>();
```

Example: Comparable Interface

- Some classes have natural orderings
  - eg Integer(3) < Integer(14)
- java.lang.Comparable

```
public interface Comparable<T> {
  int compareTo(T o)
}
```

  - Returns -ve, 0, or +ve if this object is <, =, or > argument o
- Typical use

```
if (p1.compareTo(p2) < 0)  // p1 < p2
if (p1.compareTo(p2) == 0) // p1 == p2
if (p1.compareTo(p2) > 0)  // p1 > p2
```

Good Practice: Total Ordering

- compareTo should induce a total ordering on its type parameter
  - Reflexive

    ```
    x.compareTo(x) == 0
    ```
  - Transitive

    ```
    x.compareTo(y) < 0 && y.compareTo(z) < 0
      ==> x.compareTo(z) < 0
    ```
  - Antisymmetric

    ```
    x.compareTo(y) <= 0 && y.compareTo(x)<=0
      ==> x.equals(y)
    ```
  - Total
    - Any two instances of T can be compared

Implementing Comparable

- Simple case for typical use

```
class LeadPencil implements
        Pencil, Comparable<LeadPencil> {
   int compareTo(LeadPencil o) { . . . }
}
```

- Or even better (coding to the interface!)

```
class LeadPencil implements
        Pencil, Comparable<Pencil> {
   int compareTo(Pencil o) { . . . }
}
```

- Or even better (but we'll talk about extends later)

```
interface Pencil extends Comparable<Pencil> { ... }
class LeadPencil implements Pencil {
   int compareTo(Pencil o) { . . . }
}
```

Example: Lists

- Array size fixed by instantiation with new

  ```
  Integer[] A = new Integer[145];
  ```

- What if you need the array to grow?
  - Allocate new (larger) array
  - Copy old values into new
- Better approach: java.util.List<T>
  - Generic interface
  - Holds an ordered list of Ts
  - Can be accessed by index like an array
  - But also has a dynamically changeable size
- Implementations: ArrayList, Vector
  - ArrayList more efficient, need Vector for threads

Using List (and ArrayList)

```
import java.util.List;
import java.util.ArrayList;

List<String> list = new ArrayList<String>();
list.add(“Hello”);
list.add(“there”);
list.add(0,”Sam”);
System.out.println(list.get(1));  //”Hello”

foreach (String str : list) {
   System.out.println(str);
}  //prints “SamHellothere”
```

Methods of List

- Array-like
  - set / get for index-based access
- Adding items
  - add(T) / add(int,T)
  - Causes the List to grow
- Removing items
  - remove(int) / removeRange(int,int)
- Memory management
  - isEmpty / size

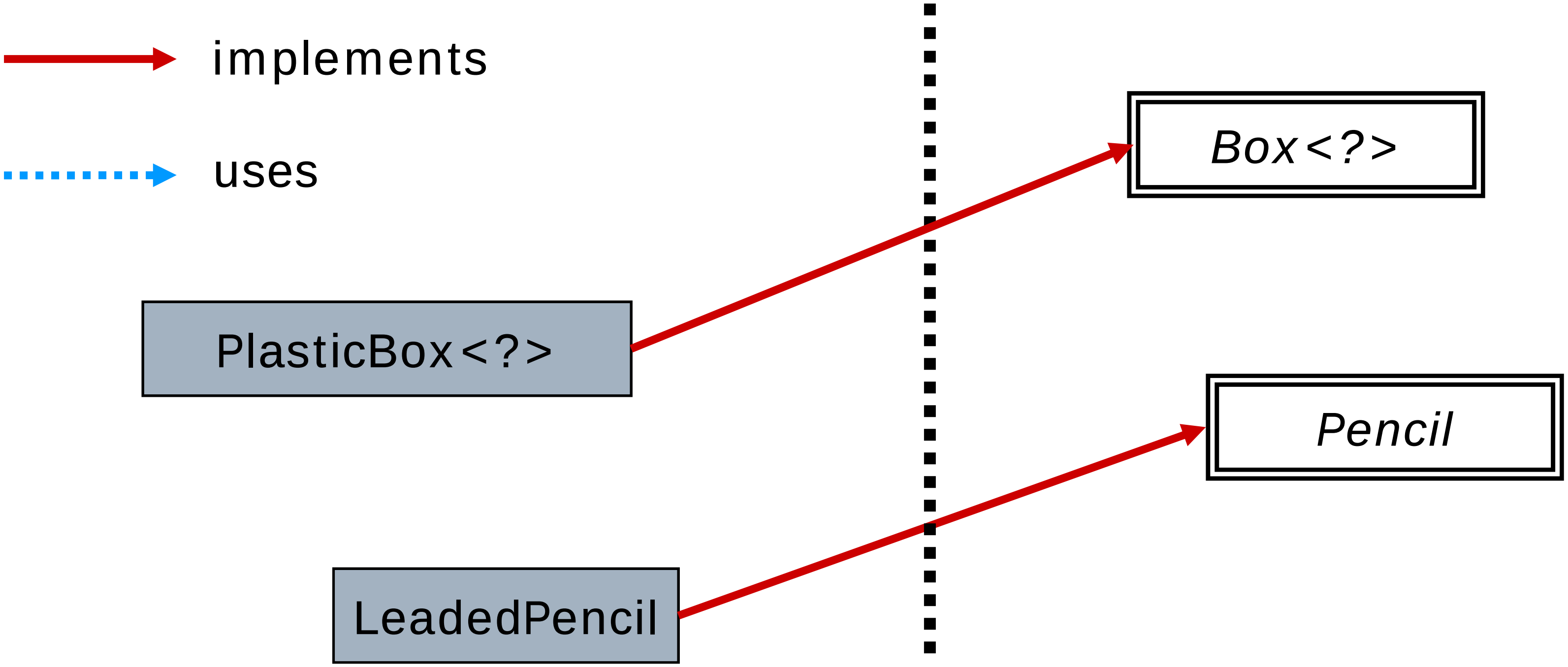
Type Erasure

- Note: PlasticBox<Pencil> and PlasticBox<String> are *not* two separate classes
  - They are two generic type *versions* of *one* class, PlasticBox

```
Box<Pencil> b1 = new PlasticBox<Pencil>();
Box<String> b2 = new PlasticBox<String>();
assert b1.getClass() == b2.getClass(); //passes
```

- Think of <Pencil> as additional information at declarations and at **new** expressions, so the compiler can do appropriate type casting and type checking
- At run-time, no generic type information remains in PlasticBox objects
  - The type parameter, T, has been "*erased*"
  - Left with one class: PlasticBox<?>
- All of this is needed so that the JVM does not have to know about and deal with generic types

Box of Pencils at Run Time



Consequences of Type Erasure

- □ All type-instances share the same static members

```
static int nextID; //shared by all Box<?>
```

- □ Static members can not refer to naked type

```
private static T value; //compile error
```

- □ New instances and arrays of naked type can not be created

```
T value = new T(); //compile error
T[] myArray = new T[50]; //compile error
```

- □ Casts ignore parameter type information

```
Box<String> x = (Box<String>) b; //unchecked
Box<?> y = (Box<?>) b; //ok
```

A Few Other Issues

- A number of other restrictions due to type erasure
  - eg cannot parameterize with primitive types: no List<int>; use List<Integer> instead
  - eg cannot have arrays of generic types: no Box<String>[] ; use ArrayList<Box<String>>
- Can also parameterize a method instead of an entire class/interface – *generic method*
  - public static <T> T getMiddle(T[] a)
- Type bounds
  - eg <T extends Serializable>
- Wildcard types

Summary

- Genericity through type parameters
  - Declaration of generic interfaces/classes
  - Use of generic interfaces/classes
- Comparable interface
  - Total ordering, strongly typed thanks to generics
- List (and ArrayList)
  - Like arrays, but better!