What are ‘Generic Types’ or ‘Generics’?

**Definition**
- Reference type parameters for use in class and method definitions
- Unlike formal parameters for methods, generic types define ‘macros,’ the class name replaces the type parameter in the source code (‘search and replace’)

**Syntax**
- `<C>` for parameter, use as `C` elsewhere (`C must be a class`)
  - `public class Widget <C> { .... }` // definition
  - `Widget<String> = new Widget<String>();` // instantiation
  - `public <C> void test( C o1, int x ) {  C  temp; .... }` // method

**Purpose: Avoiding ‘Dangerous’ Polymorphism**
Prevent run-time errors (exceptions) due to improper casting (type errors)

Example: Comparable Interface

**Prior to JDK 1.5 (and Generic Types):**
- `public interface Comparable { public int compareTo(Object o) }`
- `Comparable c = new Date();`
- `System.out.println(c.compareTo("red"));`

**JDK 1.5 (Generic Types):**
- `public interface Comparable<T> { public int compareTo(T o) }`
- `Comparable<Date> c = new Date();`
- `System.out.println(c.compareTo("red"));`

“Raw Types” and Associated Compiler Warnings

**Raw Types**
(Provided for backward compatibility)
Generic types (classes) that are used without the type parameter(s) defined
- e.g. Comparator c = Comparator<Object> c

**Compiler Warnings**
- javac will give a warning about possibly unsafe operations (type errors) at run-time for raw types
  - use -Xlint:unchecked flag
- javac will not compile programs whose generic types cannot be properly defined
  - e.g. Max.java, Max1.java (pp. 699-700 in Liang)
**Generic Types and the Inheritance Hierarchy**

See Figure 21.6, page 703

** Generic class is shared by all instances of the class, regardless of concrete types for type parameters (<T,G>, etc.)

** Caution: A<Number> is not a superclass of A<Integer> (etc.)

---

**Storing Data in Java**

**Variables**

Primitive type (int, double, boolean, etc.)

- Variable name refers to a memory location containing a primitive value

Reference type (Object, String, Integer, MyClass, etc.)

- Variable name refers to a memory location containing a reference value for data belonging to an object

**Data Structure**

Formal organization for a set of data (e.g. variables)

- int intArray [] = {1, 2}; int a = intArray[0]; intArray[1] = 5;

- e.g. Objects: data member names representing variables

- player.name, player.hits, player.team ...

- player.hits = 100;

---

**Abstract Data Types (ADTs)**

**Purpose**

Define interfaces to data structures while hiding (abstracting) implementation details

**Examples of Common ADTs**

- List: Sequence of elements. Elements may be inserted or removed from any position in the list

- Stack: List with last-in, first-out (LIFO) behaviour (“most recent,” call stack)

- Queue: List with first-in, first-out (FIFO) (“in-order”, lining up)

- Set: Unordered group of unique items

- Map: Set of entries, each with a unique key and a value

- Tree: Graph with directed edges, each node has one parent (except root), no cycles

---

**Example: Implementing Abstract Data Types**

**List ADT**

Represents series of elements, insertion and deletion of elements

**Some Possible Implementations:**

- An array and operations on it

  - l.add(E) would copy E at the end of the array, l.get(4) returns 5th item in array

- A set of objects with references to one another representing a simple graph (a "linked list") and operations on it

  - l.add(E) would create a link from last node to a new node for E ; l.get(4) traverses the graph and then returns the 5th item

**Choosing an Implementation for an ADT**

Depending on common operations, some better than others

- Finding elements in list faster for array implementation

- Inserting, deleting arbitrary elements faster for linked list implementation
Ordering in 'Unordered' ADTs

"Unordered" on paper vs. in code
In practice, some type of order must be used to implement a set, as memory and files contain ordered lists of bytes

Sets
By definition, a set is an unordered group of unique elements

Maps
By definition, a map is a set of (key,value) pairs

Ordering Sets and Maps
We can order the storage of set elements by:
1. A value computed for each element ("hash code") that determines where an element is stored (e.g. in a "hash table", a sophisticated ADT built on arrays); for maps, usually based on key value
2. The order in which elements are added (e.g. in a list)
3. The element (for map: key) values themselves (e.g. using a binary search tree)

Exercise: Generics and ADTs

Part A
1. In one sentence, what is a generic type?
2. What errors are generic types designed to prevent?
3. Which javac flag will show details for (type) unsafe operations?
4. What do the following represent:
   a) < ? extends MyClass>
   b) < ? super YourClass>
   c) <E extends Comparator<E>>
5. Write a java class GenX which has a generic type parameter T, a public data member identity of type T, and a constructor that takes an initial value for identity. Add a main method that constructs one GenX object using type String, and another using type Integer.

Part B
1. What is an abstract data type?
2. How is a list different from a set?
3. How are elements stored in a binary search tree (BST)?
4. In what ways can we order the elements of a set, or pairs of a map?
5. Are sets and map elements/pairs ordered in their ADT definitions?

The Java Collections Framework

Definition
Set of interfaces, abstract and concrete classes that define common abstract data types in Java
- e.g. list, set, map, queue, stack
Part of the java.util package

Implementation
Extensive use of generic types, hash codes (e.g. hashCode()), and Comparable interface (compareTo(), e.g. for sorting)

Collection Interface
Defines common operations for sets and lists ('unordered' ops.)

Maps
Represented by separate interfaces

Java Collections Interfaces

ADIts in Java: The Java Collections Framework

Note: Some of the material on these slides was taken from the Java Tutorial at http://www.java.sun.com/docs/books/tutorial
Note: All of these classes have a generic type parameter; e.g. Collection<E>: see course text (Ch. 22)
Common List and Set Operations: the Collection Interface

See Figure 22.3 (page 715)
List of operations to add, remove, and search for elements (of a generic type \(E\)).

Operations:
- add elements (add/addAll) to a set/list
- Remove elements (remove/removeAll)
- Take intersection (for sets), keep a set of elements (for lists) using retainAll()
- Search for elements in a collection (contains/containsAll)
- Many operations return a boolean value, to indicate whether an operation was successful.
- Return an iterator, which allows us to visit each element in a set or list one-at-a-time (similar to getting tokens from a Scanner object).

Iterator Interface

Purpose
Provide uniform way to traverse sets and lists

Instance of Iterator given by iterator() method in Collection

Operations
- Check if all elements have been visited (hasNext())
- Get next element in order imposed by the iterator (next())
- remove() the last element returned by next()
- Roughly similar to operations used in Scanner to obtain a sequence of tokens

Implementation Classes

(slides derived from: Carl Reynolds)

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
<th>Historical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet</td>
<td>TreeSet</td>
</tr>
<tr>
<td>List</td>
<td>ArrayList</td>
<td>LinkedList</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td>TreeMap</td>
</tr>
</tbody>
</table>

Note: When writing programs think about interfaces and not implementations. This way the program does not become dependent on any added methods in a given implementation, leaving the programmer with the freedom to change implementations.

Notes on ‘Unordered’ Collections (Set, Map Implementations)

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashMap, HashSet</td>
<td>Hash table implementation of set/map</td>
</tr>
<tr>
<td></td>
<td>Use hash codes (integer values) to determine where set elements or (key,value) pairs are stored in the hash table</td>
</tr>
<tr>
<td>LinkedHashMap, LinkedHashSet</td>
<td>Provide support for ordering set elements or (key,value) pairs by order of insertion by adding a linked list within the hash table</td>
</tr>
<tr>
<td>TreeSet, TreeMap</td>
<td>Use binary search tree implementations to order set elements by value, or (key,value) pairs by key value</td>
</tr>
</tbody>
</table>

HashSet

(Example: TestHashSet.java, p. 717)

Methods
Except for constructors, method interface identical to Collection

Element Storage:
- ‘Unordered,’ according to their hash codes
- **All elements are unique**
- Do not expect to see elements in the order you add them

Hash Codes
- Most classes in Java API override the hashCode() method in the Object class
- Need to be defined to properly disperse set elements in storage (i.e. throughout the hash table)
- For two equivalent objects, hash codes must be the same
LinkedHashSet
(example: TestLinkedHashSet.java, p. 718)

Methods
Again, same as Collection Interface except for constructors

Addition to HashSet
– Elements contain extra field defining order in which elements are added (as a linked list)
– List (quietly) maintained by the class

Hash Codes
Notes from previous slide still apply (e.g. equivalent objects, equivalent hash codes)

Ordered Sets: TreeSet
(example: TestTreeSet.java)

Methods
Add methods from SortedSet interface:
first(), last(), headSet(toElement: E), tailSet(fromElement: E)

Implementation
A binary search tree, such that either:
1. Objects (elements) implement the Comparable interface (compareTo())
   ("natural order" of objects in a class), or
2. TreeSet is constructed using an object implementing the Comparator interface (compare()), which may be used to compare objects of different classes
   One of these will determine the ordering of elements.

Notes
– It is faster to use a hash set to add elements, as TreeSet keeps elements in a sorted order
– Can construct a tree set using an existing collection (e.g. a hash set)

List Interface
(slide: Carl Reynolds)

```java
package List;

// Positional Access
get(int): Object;
set(int, Object): Object; // Optional
add(int, Object): void; // Optional
remove(int index): Object; // Optional
addAll(int, Collection): boolean; // Optional

// Search
int indexOf(Object);
int lastIndexOf(Object);

// Iteration
listIterator(): ListIterator;
listIterator(int): ListIterator;

// Range-view List
subList(int, int): List;
```

List: Example
TestArrayAndLinkedList.java

ListIterator
(slide: Carl Reynolds)

the ListIterator interface extends Iterator
Forward and reverse directions are possible

ListIterator is available for Java Lists, such as the LinkedList implementation

Map Interface
(slide: Carl Reynolds)

```java
package Map;

// Basic Operations
put(key: Object, value: Object);
get(Object): Object;
remove(Object): Object;
containsKey(Object): boolean;
containsValue(Object): boolean;
isEmpty(): boolean;

// Bulk Operations
void putAll(Map): void;
void clear(): void;

// Collection Views
collection(): Collection;
keys(): Set;
values(): Collection;
entrySet(): Set;
```

EntrySet

Map Examples

TestMap.java
CountOccurrenceOfWords.java

The Collections Class

Operations for Manipulating Collections
Includes static operations for sorting, searching, replacing elements, finding max/min element, and to copy and alter collections in various ways. (using this in lab5)

Note! Collection is an interface for an abstract data type, Collections is a separate class for methods operating on collections.

Comparator Interface
(a generic class similar to Comparable)
(comparator slides adapted from Carl Reynolds)

You may define an alternate ordering for objects of a class using objects implementing the Comparator Interface (i.e. rather than using compareTo())

Sort people by age instead of name
Sort cars by year instead of Make and Model
Sort clients by city instead of name
Sort words alphabetically regardless of case

Comparator<T> Interface

One method:
compare( T o1, T o2 )
Returns:
negative if o1 < o2
Zero     if o1 == o2
positive if o1 > o2

Example Comparator: Compare 2 Strings regardless of case

import java.util.*;
public class CaseInsensitiveComparator implements Comparator<String> { 
  public int compare( String stringOne, String stringTwo ) { 
    // Shift both strings to lower case, and then use the 
    // usual String instance method compareTo() 
    return stringOne.toLowerCase().compareTo( stringTwo.toLowerCase() ); 
  } 
}

Using a Comparator...

Collections.sort( myList, myComparator );
Collections.max( myCollection, myComparator );
Set myTree = new TreeSet<String>( myComparator );
Map myMap = new TreeMap<String>( myComparator );