• Classes

Each (outer) class must be in its own file, with name of file same as the class name (case sensitive). May contain nested classes.

Naming convention: starts with upper case letter; [A-Z][a-zA-Z]*

Constructor has same name as class, no return type; can be overloaded; if not given, automatic constructor is supplied; constructor cannot be called directly; called using "new".

Fields declared to be "static" have only one instance (across all elements of the class).

Methods are "non-static" by default, and they work on a given class object. A static method cannot call a non-static method, without a class object on which it applies.

• Example: Item class

```
public class Item implements Comparable<Item> {
    private int element;

    Item(int x) { element = x; }

    public int getItem() { return element; }

    public void setItem(int x) { element = x; }

    public int compareTo(Item another) {
        if (this.element < another.element) { return -1; }
        else if (this.element > another.element) { return 1; }
        else return 0; }

    public String toString(){return Integer.toString(element);}
}
```

• I/O

Until you build some expertise, use "Scanner" class for reading input and send output to the console (stdout) using System.out:

```
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.File;
public class IO {
    // Use file name from command line if given, else read from console.
    public static void main(String[] args) throws FileNotFoundException
    {  Scanner in;
        if (args.length > 0) {
            File inputFile = new File(args[0]);
            in = new Scanner(inputFile);
        } else { in = new Scanner(System.in); }

        int s = in.nextInt();   float t = in.nextFloat();
        System.out.println("s: " + s + " t: " + t);
    }
}
```

• Basic types and objects

Java has a set of built-in basic types (`int`, `long`, `boolean`, ...). Everything else is an "object". Basic types are not generally usable in contexts where an object is expected. Java provides corresponding wrapper object classes: `Integer`, `Long`, `Boolean`, ...

This distinction between basic types and objects is a bug, not a feature (poor language design).

Only objects can be used with generics (type polymorphism).

Operators cannot be overloaded (as in C++) for objects.

• main()

Execution begins with the `main()` function of the class with which execution is started. So, if we run "`java Example`" it starts execution of `main()` in the file `Example.java`.

`main()` must be "`public static void`" and takes one array of strings as its parameter. Usual declaration:

```
public static void main(String[] args) { ... }
```

`args.length` is the number of command line arguments. The arguments are: `args[0]`, `args[1]`,…, `args[args.length-1]`.

• All objects are references

All objects must be allocated with "`new`":
```
Item[] e = new Item[n];
```

It allocates an array `e` of `n` references to objects of type "`Item`": `e[0..n-1]`. Space for the actual objects is not allocated.

Later, you could run "`e[i] = new Item(...);`" or "`e[i] = obj;`" (where `obj` is some object of type `Item`, that has already been created), so that `e[i]` references an actual object.

Arrays are also references. Therefore,
```
Item[] A, B; ....  A = B;
```
is legal, and `A` now references the same array as `B`.

• Code hierarchy

Java classes are organized into packages, hierarchically.

Commonly used: `java.io`, `java.util`, `java.lang`, `System`.

Classes within a package get some additional access (like "friend" in C++). Code is organized into packages (name of folder in which the package files reside must be the name of the package).

`CLASSPATH` environment variable is used to find other classes referenced in a program; "." refers to the current directory.

For this class, create folders `cs6301` and `cs6301/`*`g#`*, and use package declaration "`package cs6301.`*`g#`*". Replace "`#`" with your group's number. For example, "`package cs6301.g42;`".

• Generics

Java's technique of allowing polymorphism.

Superficially similar to templates in C++.

Primitive; compiler hack.

Does "type erasure".

Do not attempt to create new generic arrays.

• Java Library

Many data structures are implemented in Java's library:

Lists: `ArrayList`, `LinkedList`, `ArrayDeque`
Sets: `HashSet`, `TreeSet`
Maps: `HashMap`, `TreeMap`
Priority Queues: `PriorityQueue` (binary heaps)
Hashing: `HashMap` (separate chaining)
Balanced binary search trees: `TreeMap` (Red-Black trees)

Most of these data structures are not thread-safe. They fail fast (concurrent modification exception). For example:

```
for (x: list) { if (unwanted(x)) { list.remove(x); } }
```

For multi-threaded applications, use synchronized data structures (or explicit synchronization must be done by the user).

• Interfaces

Similar to function prototypes in C/C++

A set of methods and their signatures

A class can implement any number of interfaces

Some common interfaces: `List`, `Queue`, `Iterator`, `Comparable`

Since array of elements can be sorted with comparisons and swaps, an array of objects can be sorted by a generic sorting algorithm, if the objects are from a class that implements the `Comparable` interface

• Interface Collection<E>

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. This interface is used to pass/manipulate collections where maximum generality is desired.

All collections are iterable. Some operations: `add, contains, equals, isEmpty, iterator, remove, size, toArray`.

Concrete implementations: `ArrayList, HashSet, LinkedList, PriorityQueue, TreeSet`. Not thread-safe. Not all operations may be implemented (`UnsupportedOperationException`).

Usage:
```
Collection<Item> c;       /* ... etc ... */
   for(Item x:c) { /* do something with x */ }
```

• Interface Comparator<T>

A comparison function, imposing a total ordering on a collection.
passed to methods (e.g., Collections.sort or Arrays.sort) to control the sort order,
order of elements in sorted sets and maps, or,
order of collections of objects without a natural ordering.

Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map). If the ordering imposed by c on S is inconsistent with equals, results are unpredictable.
For example,
suppose a, b, with `(a.equals(b) && c.compare(a, b) != 0)`, are added to TreeSet with comparator c. The second add operation will succeed because a and b are not equivalent from the tree set's perspective, contrary to the specification of the Set.add method.

Operations: `compare`, `equals`.

• Comparable interface

Requires the implementation of:

```
    public int compareTo (T x) { ... }
```

"this" object is compared with x and should return
- negative value if this < x,
- 0 if this = x, and,
- positive value if this > x

Algorithms such as Merge Sort and Binary Search can be implemented on arrays of generic classes as long as the underlying classes implement this interface.

• Binary search example

```
/* Implementation of Binary Search algorithm using generics. */
public static<T extends Comparable<? super T>>
  boolean  binarySearch(T[] A, int p, int r, T x) {
    while(p <= r) {
        int q = (p+r) >>> 1;   // same as "q = (p+r)/2;"
        int cmp = A[q].compareTo(x);
        if (cmp < 0) {
            p = q+1;
        } else if (cmp == 0) {
            return true;
        } else {
            r = q-1;
        }
    }
    return false;
}
```

Interface Iterator<E>

Iterate over a collection (replaces Enumeration).

Operations: `hasNext, next, remove`

Implementions: all collections

Usage:

```
void somefunction(Collection<Item> c) {
   Iterator<Item> it = c.iterator();
   while(it.hasNext()) {
      Item x = it.next();
      ...
   }
}
```

Iterator and Iterable interfaces

Iterator: Convenient interface to iterate through collections.

Methods: `hasNext(), next(), remove()`

Iterable: Any type that can be iterated.

Method: `iterator()`

If a collection class implements the Iterable interface, then "foreach" loop can be used to iterate over its objects:

```
for(Item x: col) { ... }
```

This makes the code more readable.

• Interface List<E>

An ordered collection (also known as a sequence). User has precise control over where in the list each element is inserted.

Access to elements is provided through iteration (efficient) or through indexing (inefficient). Avoid accessing lists by indexing (positional access, first element is at index 0) unless you know that its implementation is efficient (e.g., Array Lists).

A special iterator, ListIterator, allows element insertion and replacement, and bidirectional access (next, previous, hasPrevious) in addition to the normal operations of the Iterator interface.

Implementations: `ArrayList, LinkedList`.

Usage: `List<Item> lst = new LinkedList<>();`

• Interface Queue<E>

A collection that implements FIFO

Operations: `add, remove, element` (exception thrown on failure),
`offer, poll, peek` (return special value on failure).

Implementations: `ArrayDeque, LinkedList, PriorityQueue`.

Usage:
```
Queue<Item> q = new LinkedList<>();
...
q.add(x);
...
Item x = q.remove();
...
while(!q.isEmpty()) { ... }
```

• Interface Deque<E>

A linear collection that supports element insertion and removal at both ends (deque is short for "double ended queue").

Methods are provided to insert, remove, and examine the element at both ends of the queue. 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 Deque implementations; in most implementations, insert operations cannot fail.

Queue operations: `add, remove, element, offer, poll, peek`
Stack operations: `push, pop, peekLast`
Implementations: `ArrayDeque, LinkedList`

• Interface Set<E>

A collection that does not allow duplicate elements. Use care if mutable objects are used as set elements. Unpredictable behavior if an object in a set is changed in a manner that affects equals().

Operations: `add`, `contains`, `remove`, `size`, `clear`, `isEmpty`, `toArray`

Implementations: `HashSet`, `LinkedHashSet`, `TreeSet`

All set implementations are iterable.

• Interface Map<K,V>

Maps keys to values; cannot contain duplicate keys; each key can map to at most one value. Replaces Dictionary class.

Provides 3 collection views: set of keys, collection of values, or set of key-value mappings. The order is guaranteed only for some implementations like `TreeMap` (and not for `HashMap`).

Use care if mutable objects are used as map keys. Unpredictable behavior if a key is changed in a manner that affects `equals()`.

Operations: `get`, `put`, `containsKey`, `equals`, `keySet`, `values`, `entrySet`, `isEmpty`.

Avoid inefficient operations like `containsValue`.

Implementations: `HashMap`, `TreeMap`.

• Interface Map.Entry<K,V>

Map entry (key/value pair). The `Map.entrySet` method returns a collection-view of the map, whose elements are of this class. The only way to obtain a reference to a map entry is from the iterator of this collection-view. These `Map.Entry` objects are valid only for the duration of the iteration; unpredictable behavior if a map has been modified after the entry was returned by its iterator.

Operations: `equals`, `getKey`, `getValue`, `setValue`

Implemented by all map implementations