# Reference Sheet for CO120.2 Programming II

Spring 2017

## 1 Java Language Features

### 1.1 Control Flow

- `if else`, `switch` statements. Don't forget to `break` in a `switch` statement.
- `while`, `do while`, `for`, `for each` loops.
- You can label loops, and use `break` and `continue`.

### 1.2 Input and Ouput

#### Processing of Strings

- `Integer.parseInt(String s)` converts `s` to an `int`.
- `.toString()` converts object to `String`.
- `.charAt(int i)` returns character at index `i`.
- `.substring(int i, int j)` returns substring between indices `i` and `j`.
- `.split(String s)` returns array which splits string around matches of `s`.

#### Reading from Input

- `BufferedReader br = new BufferedReader(new InputStreamReader(System.in))` creates buffered reader.
- `String line = br.readLine()` reads line of input.

#### Writing to Output

- `PrintStream ps = new PrintStream(System.out)` creates print stream.
- `.println(String s)` prints new line of output.

### 1.3 Enums

E.g. `public enum Day {MON, TUE, WED, THU, FRI, SAT, SUN;}`.

- Can add fields and constructor to give properties.
- Can add methods within `enum`.

### 1.4 Arrays

E.g. `char[] abc = {'a', 'b', 'c'}`. Cannot be extended once defined. Can define values at given indices. Useful functionality in `util.Arrays`:

- `copyOfRange(T[] a, int i, int j)` copies array between indices `i` and `j`.
- `sort(T[] a, Comparator<T> c)` sorts array `a`.
- `asList(T[] a)` converts `a` into list.
- `toString(Object[] a)` converts array `a` to `String`.

### 1.5 Collections

#### Lists

- `List<E> list = new ArrayList<>()` or `new LinkedList<>()` creates new list.
- `.add(E e)` adds given `e` to list.
- `.contains(E e)` returns whether `e` is in list
- `.get(int i)` returns element at index `i`.
- `.isEmpty()` returns whether list is empty.
- `.remove(int i)` removes element at index `i`.
- `.set(int i, E e)` sets element at index `i` to element `e`.
- `.size()` returns number of items in list.
- `.stream()` creates an ordered stream from list elements.

**Sets**

- `Set<E> set = new HashSet<>()` creates new set.
- `.add(E e)` adds given `e` to set.
- `.contains(E e)` returns whether `e` is in set
- `.isEmpty()` returns whether set is empty.
- `.size()` returns number of items in set.
- `.stream()` creates an unordered stream from set elements.

**Maps**

- `Map<K,V> set = new HashMap<>()` creates new map.
- `.put(K key, V val)` maps `key` to `value`.
- `.get(Object key)` returns value associated with `key`, or null if `key` isn't present.
- `.containsKey(Object key)` returns whether `key` is present.
- `.keySet()` returns set of all keys in map.
- `.values()` returns all vals in map.
- `.isEmpty()` returns whether map is empty.
- `.size()` returns number of items in map.

**Queues**

- `Queue<E> queue = new PriorityQueue<>()` creates new priority queue.
- `.add(E e)` adds e to front of queue.
- `.remove()` removes last element from queue.
- `.poll()` retrieves *and removes* head of queue, returns `null` if empty.
- `.peek()` retrieves head of queue, returns `null` if empty.

**Deques**

- `Deque<E> deque = new ArrayDeque<>()` creates new double-ended queue.
- Can use methods above at first or last element. E.g. `pollFirst()` and `pollLast()`.
- Can use stack methods `push(E e)`, `pop()` and `peek()`.

## 1.6 Iterators

Only use if a `for` loop is not straightforward.

- `.iterator()` converts collection to Iterator.
- `.hasNext()` returns if there is another element.
- `.next()` returns the next element.
- `.remove()` removes the next element.

## 1.7 Streams

- `.stream()` converts `List` or `Set` into `Stream`.
- `.collect(Collectors.toList())` and `.collect(Collectors.toSet())` convert `Stream` into `List` and `Set` respectively.
- `.map(f)` maps `f` (a method or constructor) to a stream.
- `.filter(f)` filters out elements for which `f` applied to the element returns `false`.
- `.reduce(id, f)` reduces the stream starting from `id`, using the function `f`. Note that `f` has the type `T f(T first, T second)`.
- `.toMap(f, g)` creates map where keys and vals comes from `f` and `g` applied to each stream element respectively.
- You can perform a *pipeline* of operations on a stream.

**Lambdas** We can pass anonymous functions of the form `x -> x * 2` or `(x, y) -> x * y`.

**Method References** Static methods: `ContainingClass::method`; Instance methods: `containingObject::method`; Instance method of arbitrary object: `ContainingType::method`; Constructors: `ClassName::new`.

**Optionals** We can also `reduce` without an identity element, returning an `Optional<T>`:

- `.isPresent()` returns whether object is present.
- `.get()` returns value if present, else throws exception.
- `.orElse(T alternative)` returns value if present, else `alternative`.
- `.reduce(f)`. Now if steam is empty, an empty `Optional<T>` is returned.

## 1.8 Random

- `Random generator = new Random()` creates new `Random` object.
- `.nextInt(n)` returns random `int` in range [0..`n`).

# 2 Object-Oriented Programming in Java

## 2.1 Types

| | Primitive Types | Reference Types |
|---|---|---|
| **Definition** | built-in | class definition |
| **Creation** | literals | `new` |
| **Initialisation** | default (e.g. `0`) | `null` or constructor |
| **Usage** | operators (e.g. `+`, `*`) | methods |
| **Content** | value | pointer to an object |

Should try to make objects immutable as much as possible - using the keyword `final`.

## 2.2 Classes

Contain fields, constructor, methods.

**Fields**

1. *Constants*: should be `final` (cannot change) and `static` (one per class, not one per object). Typically `public` (accessible from anywhere).
2. *Non-constants*: should be `private` (accessible only from within class).

**Methods** `public` methods should provide service to class users. `private` methods support methods in the class.

## 2.3 Interfaces

1. Can define method *signatures* (implicity public): describe required capabilities of a class that `implements` the interface.
2. Can define `default` methods in interface (can be overriden in implementing classes).
3. Can define *constant* fields.

**Implementing Interfaces** Use notation: `@Override` when a class method implements an interface method.

**Apparent and Actual Types** Consider `Shape circle = new Circle()`. Has apparent type `Shape` but actual type `Circle`. Only methods and fields from apparent type (`Shape`) are available, but methods are implimented by actual type (`Circle`).

## 2.4 Inheritance

1. A subclass (`extends` a superclass) inherits all fields and methods from it's superclass, can also override / add additional functionality.
2. `protected` visibility in superclass allows access to its fields / methods from a subclass (also accessible anywhere within package).

`super`

1. Can call superclass constructur using `super()` (called implicitly if no other constructor provided).
2. Can call superclass method `method` from subclass using `super.method()`.

`abstract`

1. Used to define class which cannot be instantiated.
2. `abstract` methods have no body, needs to be overriden by subclass.
3. *Style*: usually a class `extends` an abstract class (with constructor and fields) which `implements` an interface (caters for case where abstract class is too specific).

`final`

1. Methods which are `final` cannot be overriden.
2. Classes which are `final` cannot be extended.

## 2.5 Casting

1. *Upcasting*: cast from subclass to superclass. Done automatically, cannot fail.
2. *Downcasting*: cast from superclass to subclass. E.g. for `Shape shape`, define `Circle circle = (Circle)shape`. Narrow apparent class so you can call certain methods / access certain fields.
3. Downcasting can lead to a `ClassCastException`. We can avoid this by using the `instanceOf` keyword to determine the actual type.
4. *Style*: `instanceOf` can indicate poor design. Often better to use subclass methods.

## 2.6 Object Equality

`Object` implements `equals` based on identity. Often we want it to compare field contents:

1. We `@Override` the method `public boolean equals(Object other)`.
2. Start by handling standard object equality cases (`==` and `null`).
3. Check incoming object appropriate type, then downcast.
4. Compare fields.

We *must* also override `public int hashCode()`.

1. Must return same value when `equals` returns true.
2. Should tend to return different values for objects which are not equal.

## 2.7 Generics

1. *Classes*: we can define a class `ClassName<A, B>` where `A` and `B` are *type parameters*. We can then use `A` and `B` as normal type names within that class.
2. *Methods*: example: `public static <S, T> Pair<S, T> makePair(s First, T second)`. Then use `S` and `T` freely inside method.

### Wildcards

1. `Set<Shape>` refers to a set of shapes. We can add any shape to this set, and can retrieve shapes. `Set<Circle>` however is *not* a subtype of `Set<Shape>`.
2. `Set<? extends Shape>` refers to any set whose elements are a subclass of `Shape`. We cannot add to this set, but we can retrieve shapes. A `Set<Circle>` *is* a subtype of `Set<? extends Shape>`.
3. *Note*: We could instead use `Set<T extends Shape>` if we care about the type of `?`.

### Inheritance

1. *Extending a generic class / Implementing a generic interface*: E.g. `public abstract class AbstractSet<E> implements Set<E>` and `public class HashSet<E> extends AbstractSet<E>`.
2. *Extending / Implementing a class to be specific*: E.g. `public interface Comparable<T>` and `public class ClassName implements Comparable<ClassName>`.

## 2.8 Functional Interfaces

1. Annotated by `@FunctionalInterface`.
2. Declares exactly one (abstract) method. E.g. `public interface Comparator<T> {int compare(T o1, T o2);}`.
3. Sort list using void `.sort(Comparator<E> c)`.
4. Means we can write `strings.sort((a, b) -> a.compareTo(b))` since this provides the single method required by `Comparator<String>`.

## 2.9 Singleton Pattern

Only allows one instance to be created:

```
public class OnlyOne {
  private static OnlyOne instance;
  private onlyOne () {}
  public static OnlyOne getInstance () {
    if (instance == null) {
      instance = new OnlyOne();
    }
    return instance;
  }
}
```

## 2.10 Cloning

If you really think it's necessary:

1. Implement `Clonable`.
2. Override `clone`.
3. Increase visibility to `public`.
4. Restrict return type. Call `Object`'s clone to create bitwise copy, using `(myClass)super.clone()`.
5. Deep-clone fields if appropriate.
6. Return the clone.

## 2.11 Exceptions

Exceptions can be thrown by called methods. Can either be caught in `try`, `catch`, `finally` block or propagated. Unchecked exceptions, such as runtime exceptions do not need to be caught / propogated.

# 3 Abstract Data Types in Java

## 3.1 Linear Data Structures

### Lists

1. Arbitrary number of elements ordered by position.
2. Need to be able to create, check if empty, obtain size, get, add and remove.
3. *Array-based*: use dynamic expansion (use copy of old array).
4. *Linked*: use nodes that store current elem and reference to next node, keep a reference to the head node. Note that you need seperate cases for dealing with the first element!
5. *Ordered*: Use `compareTo` from `Comparable` interface. Add using recursion.
6. *Iterators*: Usually implemented as inner classes. Implement `Iterable`. Need to be careful of concurrent modifications.

### Stacks

1. Linear sequence of items with insertions and deletions only allowed at top. Implements LIFO access.
2. Example applications: frame stack for recursive functions, reverting text, validating parentheses.
3. Need to be able to create, check if empty, push, pop and peek.
4. *Array-based*: Use array of specific length, keep the index of top element.
5. *List Based*: Use a linked list. Head of list is the top.

### Queues

1. Allows insertion only at back, deletion only at front. Implements FIFO access.
2. Example applications: scheduling, demerging.
3. Need to be able to create, check if empty, enqueue and dequeue.
4. *Array-based*: Use circular array (using `%`) of specific max length. Keep indexes for first and last elements.
5. *List-based*: Use linked list. Keep references to first and last elements.
6. *Priority*: Extend node class to include priority. Keep an ordered linked list whose elements are orderd according to priority value. Or use a min-heap.

### Maps

1. Collection of items described by (key, value) pairs. No duplicate keys.
2. Example applications: caches, finding duplicates, random access to large data sets.
3. Need to be able to create, check if empty, obtain size, check if contains element, put, get and remove elements.
4. *Array-based*: use an array to store values, index computed by hash function applied to key.
5. Hash function should: minimise collisions, be fast to compute, distriibute elements uniformly through the array, be deterministic.
6. Example hash functions: selecting digits, adding digits together, modulo arithmetic.

### Sets

1. Models mathematical set.
2. Need to be able to create, check if empty, obtain size, add and remove.
3. Can use hash map to implement.

## 3.2 Tree-Based Data Structures

### Binary Trees

1. Consist of data element (root) and two disjoint binary trees.
2. Need to be able to create, check if empty, obtain size, set and get left and right subtrees.
3. Example applications: syntax trees, Huffman coding trees.
4. *Height*: the number of *levels* in a tree.
5. *Perfectly balanced*: height equal to length of shortest path. A perfectly balanced tree has $2^h - 1$ nodes.
6. *Complete*: full down to height $h-1$ and level $h$ filled from left to right.
7. *Linked*: contain current element and references to left and right subtrees.
8. *Array-based*: root at index 0, children at $2i+1$ and $2i+2$, parent at $(i-1)/2$.

**General Trees** Nodes can have any number of children. Linked implementation possible with lists.

**Traversal**

1. *Depth First*: Recursive (or using stack). *Pre-order*: root, left, right. *In-order*: left, root, right. *Post-order*: left, right, root.
2. *Breadth First*: Level by level. Use a queue.

**Binary Search Trees**

1. All values on left subtree smaller than in root, on right are larger.
2. Provide a `Comparator` or implement `Comparable` to provide search key.
3. Search, insert and size all defined recursively.
4. Remove has several cases:
   (a) If leaf, just delete.
   (b) If has one child (L or R), use that child.
   (c) If has two children, swap with leftmost child of the right child.

**Self-Balancing Trees**

1. *Balanced*: Nodes at level $\leq h - 2$ have 2 children.
2. *Rotations*:
   (a) Reparent child on opposite side.
   (b) Set child on rotation side to the old parent.
3. *Double Rotations* (e.g. Left-Right rotation):
   (a) Left side (left rotation).
   (b) Right rotation.
4. *AVL Tree*: Store height values. After each insertion, check node and rebalance if necessary:
   (a) Left bigger than Right and Left side (left bigger than right): left-right rotation.
   (b) Left bigger than Right: right rotation.
   (c) Right bigger than Left and Right side (right bigger than left): right-left rotation.
   (d) Right bigger than Left: left rotation.
5. *Red-Black Tree*: Store a color value (R or B). Root is black. Number of black nodes on every path from the root is the same. No two consecutive nodes are red. Start by setting the node red. Then five cases to handle:
   (a) *Root node*: If no parent, colour current black, finish.
   (b) *Black parent*: If parent is black, finish.
   (c) *Red uncle*: If uncle exists and is red, colour parent and uncle black and grandparent red. Begin case (a) on grandparent.
   (d) *Fix zig-zags*:
      i. If parent left of grandparent and current right of parent: rotate parent left. Begin case (e) on parent.
      ii. If parent right of grandparent and current left of parent: rotate parent right. Begin case (e) on parent.
   (e) *Fix colouring and rotate*:
      i. If is left child, colour parent black and grandparent red. Rotate grandparent right, finish.
      ii. If is right child, colour parent black and grandparent red. Rotate grandparent left, finish.

**Heaps**

1. Complete binary tree with weak ordering.
2. Need to be able to create, check if empty, obtain size, add elements, peek and poll max.
3. *Max Heap*: root contains largest element. Each subtree is a max heap.
4. *Array-based* is efficient implementation (see bin trees).
5. *Retrieving max element*: leaves two subheaps to be merged:
   (a) Replace root with last element in array to create semi-heap.
   (b) *Sift* by swapping root with largest child recursively.
6. *Adding element*: add element to end of array and sift upwards.
7. *Heap sort*: convert array to heap first, then sort array heap.

# 4 Concurrency in Java

## 4.1 Threads

1. Specify a thread.
   (a) Extend `Thread` and override `run()`.
   (b) Implement `Runnable` and override `run()`.
   (c) `Thread t = new Thread(() -> {code goes here})`.
   (d) `Runnable r = () -> {code goes here}; Thread t = new Thread(r)`.
2. Start the thread using `t.start()`.
3. Wait for termination using `t.join()`.
4. Sleep `x` seconds using `TimeUnit.SECONDS.sleep(x)` (goes inside `Thread`).

## 4.2 Avoiding Race Conditions

1. Use the `synchronized` keyword (do this for all methods that can read or write shared fields - note `final` fields do not need it!).
2. Use locks:
   (a) Always use a `try-finally` block to ensure lock is released!
   (b) `ReentrantLock`
      i. Use `Lock lock = new ReentrantLock(true)` to set a fair lock
      ii. `lock.lock()` and `lock.unlock()` to lock and unlock.
   (c) `ReadWriteLock`
      i. `ReadWriteLock lock = new ReentrantReadWriteLock(true)` for new lock.
      ii. `lock.readLock().lock()` and `lock.readLock().unlock()` for read lock and unlock.
      iii. `write.readLock().lock()` and `write.readLock().unlock()` for write lock and unlock.
   (d) `StampedLock`
      i. `StampedLock sl = new StampedLock()`.
      ii. `long stamp = sl.readLock()` and `sl.unlockRead(stamp)`.
      iii. `long stamp = sl.tryOptimisticRead()` followed by `if (!sl.validate(stamp)){stamp = sl.readLock()}`.
3. Use semaphores:
   (a) Create new using `Sephamore(int permits, bool fair)`.
   (b) `aquire()` to take a permit, waiting if necessary.
4. Use atomic classes. E.g. `new AtomicBoolean(x)`, `new AtomicInteger(x)` and new `AtomicReference(x)`, use `.get()` and `.set(x)`.
5. Use adders and accumulators. E.g. `new DoubleAccumulator((x, y) -> x * y, 1.00)` followed by `.accumulate(x)` and `.get()`.
6. Use the `volatile` keyword. Guarantees visibility of changes (ignores cache). Can avoid locks and synchronisation if operations are atomic.
7. Use the `final` keyword. Enforces visibility (as above).
8. Immutable objects are always thread safe. Objects are immutable if all fields are final and immutable (if a field is not immutable, return copies of it using `clone()`).
9. Use a `ConcurrentHashMap`. Also includes functionality such as:
   (a) `.forEach((k, v) -> ...)`
   (b) `.putIfAbsent(k, v)`
   (c) `.getOrDefault(k, defaultV)`
   (d) `.replaceAll ((k, v) -> ...)`
   (e) `.search((k, v) -> {...})`
   (f) `.reduce((k, v) -> {...}, (p1, p2) -> {...})`
10. Use a `parallelStream`.
11. *Data parallelism* approaches (e.g. for linear algebra and image processing - can avoid locks entirely).

# 5 Tips for Lexis Tests

- Read the questions and guidance very carefully.
- Think carefully anywhere you have to change mutable objects. Don't forget to store the old information in a temp variable if you need it later.
- Avoid repetition of code. Move it to a private helper method or abstract class if you can.
- Handle unexpected cases. Make use of the exceptions provided.
- Be careful. Provided ADTs are often 1-indexed.