CSE 1325 Exam 2 Study Sheet

This study sheet is provided AS IS, in the hope that the student will find it of value in preparing for the indicated exam. As always, it is the student's responsibility to prepare for this exam, including verification of the accuracy of information on this sheet, and to use their best judgment in defining and executing their exam preparation strategy. *Any grade appeals that rely on this sheet will be rejected.*

Summary of changes for Exam #2:

- Object-Oriented Programming: Added Polymorphism
- New Sections: Menu-Driven Interfaces, File I/O, Generics, Standard Class Library, Collections / Maps, Iterators, Algorithms, Concurrency (Threads), Anonymous Classes, and Lambdas
- Vocabulary: Added Memory Spaces, Concurrency, Terms added to Class Members

Vocabulary

- **Object-Oriented Programming (OOP)** – A style of programming focused on the use of classes and class hierarchies

The "PIE" Conceptual Model of OOP

- **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results
- **Inheritance** – Reuse and extension of fields and method implementations from another class
- **Encapsulation** – Bundling data and code into a restricted container
- **Abstraction** – Specifying a general interface while hiding implementation details (sometimes listed as a 4th fundamental concept of OOP, though I believe it's common to most paradigms)

Types and Instances of Types

- **Primitive type** – A data type that can typically be handled directly by the underlying hardware
- **Enumerated type** – A data type that includes a fixed set of constant values called enumerators
- **Class** – A template encapsulating data and code that manipulates it
- **Interface** – a reference type containing only method signatures, default methods, static methods, constants, and nested types
- **Instance** – An encapsulated bundle of data and code (e.g., an instance of a program is a process; an instance of a class is an object)
- **Object** – An instance of a class containing a set of encapsulated data and associated methods
- **Variable** – A block of memory associated with a symbolic name that contains a primitive data value or the address of an object instance
- **Operator** – A short string representing a mathematical, logical, or machine control action
- **Reference Counter** – A managed memory technique that tracks the number of references to allocated memory, so that the memory can be freed when the count reaches zero
- **Garbage Collector** – A program that runs in managed memory systems to free unreferenced memory

Class Members Et. Al.

- **Field** – A class member variable (also called an "attribute" or "class variable")
- **Constructor** – A special class member that creates and initializes an object from the class
- **Destructor** – A special class member that cleans up when an object is deleted (not supported by Java)
- **Method** – A function that manipulates data in a class (also called a "class function")
- **Getter** – A method that returns the value of a private variable
- **Setter** – A method that changes the value of a private variable

Inheritance

- **Multiple Inheritance** – A subclass inheriting class members from two or more superclasses
- **Superclass** – The class from which members are inherited
- **Subclass** – The class inheriting members
- **Abstract Class** – A class that cannot be instantiated
- **Abstract Method** – A method declared with no implementation
- **Override** – A subclass replacing its superclass' implementation of a method

Scope

- **Namespace** – A named scope
- **Package** – A grouping of related types providing access protection and namespace management
- **Declaration** – A statement that introduces a name with an associated type into a scope
- **Definition** – A declaration that also fully specifies the entity declared
- **Shadowing** – A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope

Algorithms

- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
- **Generic** – A Java construct representing a method or class in terms of generic types
- **Java Class Library** – A library of algorithms focused on organizing code and data as Java generics
- **Collection** – An object that stores and manages other objects
- **Iterator** – A pointer-like standard library abstraction for objects referring to elements of a collection
- **Algorithm** – A procedure for solving a specific problem, expressed as an ordered set of actions

Memory Spaces

- **Stack** – Scratch memory for a thread of execution (in Java, e.g., `int i=5;`)
- **Heap** – Memory shared by all threads of execution for dynamic allocation (`Foo f = new Foo();`)
- **Global** – Memory for static fields (and in C++, non-scoped variables)
- **Code** – Read-only memory for machine instructions

Concurrency (Multi-Threading)

- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space
- **Reentrant** – An algorithm that can pause while executing, then safely be executed by a different thread
- **Mutex** – (contraction of "mutual exclusion") An object that prevents two properly written threads from concurrently accessing a shared resource
- **Synchronized** – The ability to control the access of multiple threads to any shared resource

Error Handling

- **Exception** – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code
- **Assertion** – An expression that, if false, indicates a program error (in Java, via the `assert` keyword)
- **Invariant** – Code for which specified assertions are guaranteed to be true (often, a class in which fields cannot change after instantiation)
- **Data Validation** – Ensuring that a program operates on clean, correct and useful data
- **Validation Rules** – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data

Version Control

- **Version Control** – The task of keeping a system consisting of many versions well organized
- **Baseline** – A reference point in a version control system, usually indicating completion and approval of a product release and sometimes used to support a fork

Process

- **Unified Modeling Language (UML)** – The standard visual modeling language used to describe, specify, design, and document the structure and behavior of object-oriented systems
- **Class Hierarchy** – Defines the inheritance relationships between a set of classes
- **Class Library** – A collection of classes designed to be used together efficiently

Object-Oriented Programming

Encapsulation is simply bundling data (fields) and code (methods) into a container (class or enum) with restricted scope (package-private, protected, or private). **Know how to write a class declaration from a UML class diagram.**

Inheritance is reuse and extension of field and method implementations from another class. Know how to derive a subclass from a superclass, as in `class Sub extends Super`. Public, package-private, and protected members of the superclass are directly accessible from the subclass, while private members are not. **Constructors do NOT inherit** but can be invoked via the `super` keyword. Multiple inheritance of *interfaces*, e.g., `class TA implements Student, Faculty` is perfectly fine, but Java does NOT support multiple inheritance of *classes*. **Use the @Override annotation** on a method declaration when overriding a superclass method, so javac will report an error if a superclass declares no matching method signature.

Polymorphism is the provision of a single interface (superclass methods) to multiple subclasses, enabling the same method call to invoke different overridden methods to generate different results. When a method is called on a variable of a superclass type that holds an object of a subclass type, the subclass object's method is invoked. `ArrayList<Base> al = new ArrayList<>(); v.add(new Derived()); f or(var d : v) d.foo();` is an example of polymorphism, assuming Derived.foo() overrides Base.foo().

General Java Knowledge

Java syntax: (much of this is from CSE 1320) assignments, operators, relationals, naming rules, the 5 most common primitives (boolean, byte (8 bits), char (16 bits!), int, double) and common classes (String, StringBuilder, ArrayList), instancing (invoking the constructor), for and for each, while, if / else if / else, the ? (ternary) operator aka `(x = (a > b) ? a : b;)`, switch statements *and expressions*.

Packages: Purpose, how to use them, and how to build them. Their public classes, constructors, methods, and fields are documented in special `/**` to `*/` comments containing tags such as @author, @version, @since, @param, @return, @exception, @throws, @link, and @deprecated. The JavaDoc documentation is built using the `javadoc` tool, typically via `ant` with a clause in your build.xml file.

Import: This adds the name from the specified package to the local namespace.

Java always passes by value, but the value of a non-primitive variable is the *address* of the object. Thus an object passed as a parameter *may be modified* but *cannot be replaced*.

Visibility levels

- **Private** – Accessible within this class only
- **Protected** – Accessible within this class and its subclasses only
- **Package-private** – Accessible in this class, its subclasses, and classes within the same package only
- **Public** – Accessible anywhere

Custom Java Types

Java supports 3 custom type mechanisms:

- **Interface** – Defines method signatures, default methods, static methods, constants, and nested types and supports multiple inheritance
- **Class** – Encapsulates fields, constructors, and methods and supports inheritance
- **Enum** – A class that also includes a list of enumerators and does NOT support inheritance

Class Members

```
public class Foo {
  public Foo() {this(0, 0);}
  public Foo(int a, int b) {this.a = a; this.b = b;}
  public Foo add(Foo rhs) {return new Foo(rhs.a + a, rhs.b + b);}
  @Override
  public String toString() {return "(" + a + "," + b + ")";)
  @Override
  public boolean equals(Object o) {
      if(o == this) return true;             // 1. An object is equal to itself
      if(!(o instanceof Foo)) return false;  // 2. A different type is not equal
      Foo f = (Foo)o;                        // 3. Downcast to this type
      return (a == f.a) && (b == f.b);       // 4. Compare relevant fields
  }
  @Override
  public int hashCode() {
      Objects.hash(a,b); // parameters MUST match step 4 of equals!
  }
  private int a;
  private int b;
}
```

- The class is `Foo` and is public (visible in all packages)
- The fields are `int a;` and `int b;` and are private (visible only within the Foo class)
- The 2 public constructors are `Foo() {this(0, 0);` (which chains) and `Foo(int a, int b) {this.a = a; this.b = b;}`
- Constructor `Foo()` chains (delegates) to constructor `Foo(int a, int b)`
- The add operator method is `public Foo add(Foo rhs) {return new Foo(rhs.a + a, rhs.b + b);}`
- Calls to the add operator method may be chained, e.g., `Foo f = f1.add(f2).add(f3);`
- The String conversion method is `String toString() {return "(" + a + "," + b + ")";)`
- String conversion is automatic in a "string context", e.g., `String s = "" + foo;`
- The equals method is `public boolean equals(Object o) { /* omitted */ }`
- The hashCode method is `public int hashCode() { /* omitted */ }`
- Given two Foo *instances* f1 and f2, `if (f1 == f2)` compares *memory addresses* (is this the *same object*?), while `if (f1.equals(f2))` compares *fields* (do these objects have *equal fields*)? For *primitives*, `x == y` compares *values* — their addresses can't be compared (may be in registers).
- An instance is `Foo f = new Foo(3,4);` Special cases include `String s = "Hello";` and `enum Color {RED, BLUE}; Color c = Color.RED;`

A **default constructor** (with no parameters, like `Foo()` above) is provided by default, but **only** if no non-default constructor is defined. Any number of constructors may be defined ("overloading"), as long as the types of each set of parameters is unique (called the “parametric signature” of the constructor).

"Overloading" and "parametric signature" also apply to methods. All of the other elements of a method's declaration, such as modifiers, return type, parameter names, exception list, and body are NOT part of its parametric signature.

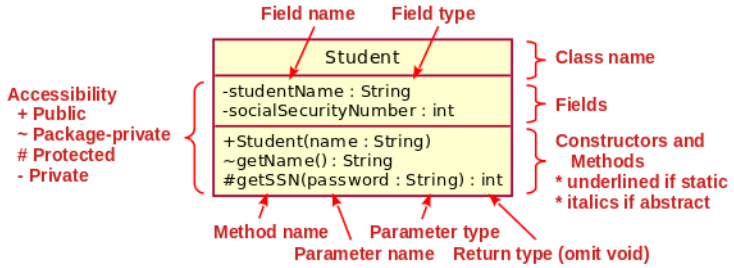
A **destructor** is the opposite of a constructor - it "tears down" the object, freeing any resources necessary. Since Java manages memory via a garbage collector, **Java does not need nor support destructors**.

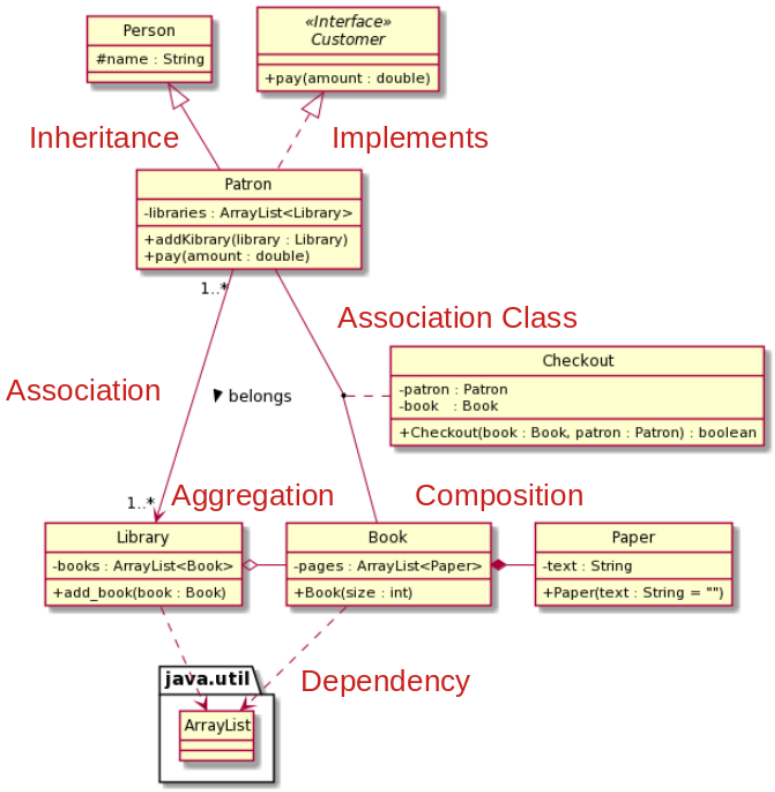
Class Member Options

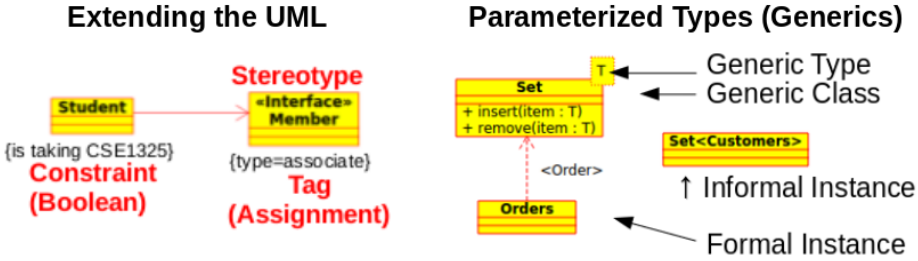
- **Non-static field** – A unique value for every object
- **Static field** – The same value (and memory location) shared among all objects of this class, e.g., `static int x;`
- **Non-static method** – Called only via the object (`foo.bar()`), and can access both static and non-static fields
- **Static method** – Called via object (`foo.bar()`) or class (`Foo.bar()`), and can access only static fields
- **Non-final field** – Value can be changed by any code with visibility
- **Final field** – Value can only be assigned once, e.g., `final int x = 3;` OR `final int x; if (y==0) x=0; else x=255;`
- **Non-final method** – May be overridden by a subclass
- **Final method** – May not be overridden by a subclass
- **Non-final class** – May be subclassed (extended)
- **Final class** – May not be subclassed
- **Overridden method** – Matches superclass method's name, parameter types and order, and return type, e.g., `@Override void int size()`
- **Overloaded method** – Matches method name in *same* class, but with different parameter types or order, e.g., `void int size() {/*...*/}` and `void int size(boolean bytes) {/*...*/}`

UML Class Diagram

Understand and be able to draw the basic forms of the UML class diagram, the various relationships between, and user-defined extensions below.







3 types of extensions: Stereotype (such as «`port`»), tags (assignments such as `{language=Java}`), and constraints (booleans such as `{interfaces}` or `{sloc>100}`). Parameterized (generic) types are supplied when the class is instanced or the method called.

Input / Output (I/O)

Streams

Streams generalize I/O as a sequence of bytes to or from the console, keyboard, file, device, whatever.

- Console input is `System.in` (also known as STDIN)
- Console output is `System.out` (also known as STDOUT)
- Console errors go to `System.err` (also known as STDERR)
- Use `System.out.printf` and `String.format` for for formatting via the familiar C `printf` codes. You needn't know the details of these codes, as I'll provide a handout if needed, but know how to use them. When manipulating text, the `StringBuilder` class is *much* more efficient than `String`.

```
java.util.Scanner in = new java.util.Scanner(System.in);
System.out.print("Enter a positive int: ");
int i = in.nextInt(); // next() for word, nextLine() for \n-terminated String
if(i<=0) System.err.println("That's not positive!");
```

Menu-Driven Interfaces

Class `MenuItem` encapsulates the menu text (accessed via `toString`) and the method (as a `Runnable` implementation invoked via `run()`) to call when selected.

Class `Menu` aggregates many `MenuItem` objects (via the `addMenuItem` method), provides the menu itself (via `toString`), and dispatches to the application code (via `menu.run`).

```
this.menu = new Menu();                                  // Instance the menu object

// Specify the menu           Menu text       Method to call when selected
//                            ==============  ============================
menu.addMenuItem(new MenuItem("Exit",          () -> endApp());          // lambda
menu.addMenuItem(new MenuItem("Do Something", () -> doSomething());     // lambda

System.out.println(menu);                                // Print the menu
int selection = Integer.parseInt(in.nextLine());         // Read the user's selection
menu.run(selection);                                     // Dispatch
```

File I/O

To use a file, you need its name and data format. Use a magic cookie and file version number.

For **text files** (preferred) use BufferedWriter / FileWriter with the write methods and BufferedReader / FileReader with the readLine method. You can also use a `Scanner(new File("text.txt"))` object just as with System.in.

Use all of these with try-with-resources, which relies on interface `Autocloseable`. This code copies the file specified by args[0] to file args[1], appending only if any third command line argument is provided.

```
String line;
// BufferedWriter overwrites on one parameter, appends if a second boolean parameter is true
try (BufferedReader br = new BufferedReader(new FileReader(args[0]));
     BufferedWriter bw = new BufferedWriter(new FileWriter(args[1], args.length>2))) {
    while((line = br.readLine()) != null) bw.write(line); // copy text file 1 line at a time
} catch (IOException e) { // IOException is checked, must "catch" or "throws"
    System.err.println("Failed to copy " + args[0] + " to " + args[1] + ": " + e);
}
```

Error Handling

Know why exceptions are usually superior to returning an error code, and how to instance, throw, catch, and handle an exception. Know that Java has both `Exception` and `Error` classes (with *many* subclasses) that can be instanced and thrown, but only catch `Exception` objects, NOT `Error` objects.

```
public class ErrorHandling {
    public int foo(int data) {
        if (data > 10) throw new Exception("bad data");  // Throw an exception
    }
    public static void main(String[] args) {
        try {  // Create scope in which exceptions can be caught
            int i = foo(42);
        } catch (Exception e) {  // Catch the Exception
            System.err.println("Method failed with " + e.getMessage());
        }
    }
}
```

Some exceptions are "checked" (I'll tell you which on the exam). A checked exception must either be caught OR the method must declare that it may be thrown using `throws`, like this.

```
public int BufferedReader open(String filename) throws IOException {
    return new BufferedReader(new FileReader(filename));
}
```

You may also define custom exceptions by inheriting from class Exception or one of its subclasses. It's a good practice to delegate to each of the superclass' constructors (for the example below, Exception has 4 constructors), though you may also add your own. I'll give you the superclass constructor documentation on the exam so you know to which you should delegate.

```
class BadChar extends Exception {
    public BadChar(String s, char c) {
        super("Bad character '" + c "' in " + s);
    }

    // Delegates to Exceptions 4 constructors
    public BadChar()                                { super();           }
    public BadChar(String message)                  { super(message);    }
    public BadChar(Throwable err)                   { super(err);        }
    public BadChar(String message, Throwable err) { super(message,err);}

}
```

Use `System.out` for data only and `System.err` for error messages only. Java returns 0 from main() by default, but you may use `System.exit(-1)` for a non-zero return. The `ant` and `make` tools will abort on a non-zero return code, and other tools (including bash scripts) can access this integer (in bash, $?) to behave differently when a program fails.

Understand the concepts of pre-conditions (at the start of the method, usually verifying parameters) and post-conditions (at the end of a method, usually verifying results). You may use the `assert` keyword to check them, but only if the -ea flag is given on the command line (`java -ea Main`). Usage is `assert errorCode == 0 : "An error occurred";` If errorCode is 0, nothing happens, but if not zero, the program throws an `AssertionError` exception with the message after the colon (the colon and message are optional), resulting in an abort with the message

```
Exception in thread "main" java.lang.AssertionError: An error occurred
```

Miscellaneous

Idioms

Read lines until end of file (from the keyboard that's Control-d for Linux and Mac, Control-z for Windows) given `String line` and `ArrayList<String> text`. Know these.

- Using Scanner: `while(scanner.hasNextLine()) text.add(scanner.nextLine());`
- Using BufferedReader: `while((line = br.readLine()) != null) text.add(line);`

Strings

- Declare: `String s = "Hi"; String name = new String(char[]{'R','i','c','e'});`
- Declare with multi-line Text Block:

```
String paragraph =
    """
    This is a "multi-line"
    text block.""" // Still interprets \n, though
```

- Concatenate: `s += ", my name is " + name + '\n' + paragraph;`
- Size of (number of characters): `s.length()`
- Tests: `if(s.startsWith("Ri") && s.endsWith("ce") && s.equals("Rice"))`
- Case-insensitive: `if(name.startsWithIgnoreCase("ri"))` or `name.equalsCaseInsensitive("Rice"))`
- 3-way compare: `int c = name.compareTo("George"); <0, ==0, >0`
- 3-way case-insensitive: `int c = name.compareToCaseInsensitive("George");`
- sprintf: `String f = String.format("%4d", 123);`
- Char access: `for(char c : s.toCharArray())` and `s.charAt(3)`
- Substrings: `s.substring(1,4) // access s.charAt(1) to s.charAt(3)`
- Remove surrounding whitespace: `s.trim()`
- Convert case of all characters: `s.toUpperCase()` and `s.toLowerCase()`
- Find index of substring: `int index = s.indexOf("name");` and `s.lastIndexOf("e");`
- Replace substring: `s.replace("name", "username");`
- Split: `String p = "/home/ricegf/resume.odt"; for(String dir : p.split("/"))`

StringBuilder

- Construct: `Stringbuilder sb = new StringBuilder();`
- Append: `sb.append("Me, too!");`
- Insert string: `sb.insert(5, " and me"); // at [5]`
- Delete range: `sb.delete(3, 7); // from [3] through [6]`
- Reverse string: `sb.reverse();`

Also includes these `String` methods that are list above - `charAt`, `compareTo`, `indexOf`, `length`, and `substring` - but NOT the other methods listed for `String` above.

Generics

A generic is a Java construct representing a class or method in terms of generic types. Generics enable algorithms independent of the types to which the algorithms can apply – called "generic programming". The type(s) are replaced with generic types E, T, U, V, ... and the code is written in those terms.

The generic type is replaced with any object type (*not* primitive) when the class is instanced or when the method is called.

Be able to code simple examples like this. See "UML" for the UML representation.

```
class Test<T> { // Generic Class
    T t;
    Test(T t) { this.t = t; }
    public T getField() { return this.t; }
}
public class TestTest { // VVV Generic Method
    public static <E> void printIt(E e) {System.out.println(e.toString());}
    public static void main(String[] args) {
        Test<String> test = new Test<>("I'm generic!");
        printIt(test);
    }
}
```

If the generic type is constrainted (for example, must implement a specific interface), this must be specified like this. ALWAYS use extends in this case, even for interfaces.

```
public class MaxGeneric {
   public static <T extends Comparable<T>> T max(T lhs, T rhs) {
       if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
   }
}
```

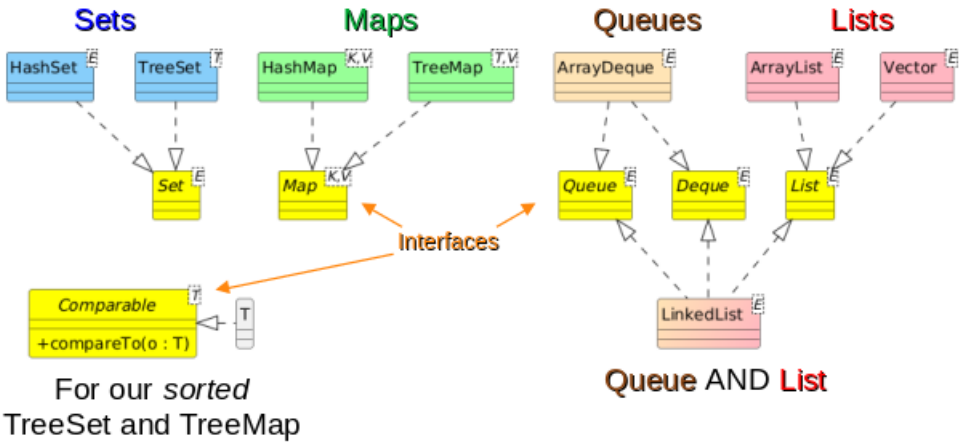
Java Class Library

The Java Class Library (JCL) provides a variety of classes from many modules. The java.lang module is imported automatically into every program. The java.io module provides input and output, while java.time supports date and time calculations.

Collections / Maps

The java.util module defines the following, which you should know how to use given basic documentation (except basic ArrayList and HashMap methods). NOTE: Deque is a "Double-Ended QUEue". `LinkedList` is both a list AND a queue. See the simple class diagram that follows.

- **Algorithms** like sort, search, and copy
- **Iterators** which allow algorithms to generically manipulate collections
- **Collections are generic classes implementing generic interfaces**
    - **List** is implemented as `Vector`, `ArrayList`, and `LinkedList`
    - **Queue** and **Deque** are implemented as `ArrayDeque` or `LinkedList`
    - **Set** is implemented as `HashSet` (unsorted) and `TreeSet` (sorted)
- **Map** is implemented as `HashMap` (unsorted) and `TreeMap` (sorted by keys)



Here's a brief summary of the methods to help you find method documentation.

- **Add an Element**
  - `add(E e)` – List, Deque, Set
  - `push(E e)` – Deque
  - `put(K key, E value)` – Map
- **Get an Element**
  - `E get(int index)` – List
  - `V get(Object key)` – Map
- **Remove an Element**
  - `remove(int index)` – List
  - `E removeLast()` – List, Deque
  - `E pop()` – Deque
  - `boolean remove(Object key)` – List, Deque, Set, Map
  - `clear()` – List, Deque, Set, Map
- **Check the Number of Elements**
  - `int size()` – List, Deque, Set, Map
  - `boolean isEmpty()` – List, Deque, Set, Map
- **Copy to Array**
  - `Object[] toArray()` – List, Deque, Set
  - `T[] toArray(T[] a)` – List, Deque, Set
- **Search**
  - `boolean contains(Object o)` – List, Deque, Set
  - `boolean containsKey(Object o)`, `boolean containsValue(Object o)` – Map

- **Iterate**
    - `for(var v : vs)` – List, Set
    - `for(var key : map.keySet())` **– Map**
      `var value = map.get(key);`
    - Deque isn't compatible with for-each loops
    - Iterators work with all Collections (but not Maps)

Iterators

Collections that implement the Iterable interface (which requires the iterator() method, among others) may be used with a for-next loop.

Obtain an iterator using a Collection's iterator() method. It will be referencing the first element of that collection. Then use the following methods.

- `hasNext()` returns true if another element is available
- `next()` returns the next element
- `remove()` removes the last element returned by next()

More capable is the bi-directional ListIterator, obtained with the listIterator() method where available. A list iterator has additional methods beyond the iterator methods above.

- `hasPrevious()` returns true if the previous element is available
- `previous()` method returns the next element
- `nextIndex()` and previousIndex() returns the index of the element to which the ListIterator points
- `add(E e)` inserts the element into the collection at the index to which the ListIterator points
- `set(E e)` overwrites the element to which ListIterator points (but only if neither add nor remove have been called yet)

Algorithms

You should know the basic use of the following methods without documentation.

- `Math.random()` – returns a random double between 0 and 1. Adjust, for example, `(int) (Math.random()*20-10)` gives an int between -10 and 9.
- `Collections.sort(Collection c)` sorts in place
- `Collections.shuffle(Collection c)` places the collection's elements in random order
- `Collections.reverse(Collection c)` reverses the order of all elements in the collection
- `Collections.fill(Collection c, E value)` populates the collection with value
- `Collections.copy(Collection c1, Collection c2)` copies teh elements of c1 to c2
- `int Collections.binarySearch(Collection c, E value)` returns the index of the value in c or a negative int if not found: IMPORTANT: The Collection MUST be sorted!

Concurrency (Threads)

Concurrency makes better use of multiple processing units (cores) on a modern computer. Each program is an OS process with its own private memory, which can each host many threads of execution sharing that private memory.

Know how to create and join threads, get their ID, and sleep the thread. Know the idiom of creating an array of threads to manage a “thread pool”. Know that the order in which threads execute is unpredictable.

Here’s two threads, one using a lambda and the other the `implements Runnable` approach.

```
class Bobbin implements Runnable {
    String msg;
    public Bobbin(String msg) {this.msg = msg;}

    @Override // Runs when Bobbin instance provided to Thread constructor
    public void run() {
        try {
            for(int i=5; i>0; --i) {
                System.out.print(i + "... ");
                Thread.sleep(1000); // pause 1 second
            }
            System.out.println("Bobbin says: " + msg);
        } catch(InterruptedException e) { // sleep() may throw InterruptedException
        }
    }

    public static void main(String[] args) {
        Bobbin bobbin = new Bobbin("Godspeed on the final!");
        Thread t1 = new Thread(bobbin); // runs bobbin.run() as a thread
        t1.start();

        Thread t2 = new Thread(()->System.out.println("I'm a lambda!"));
        t2.start();

        try { // join() may throw InterruptedException
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
        }
    }
}
```

Anonymous Classes

An anonymous class both declares and instances a class simultaneously, often to implement an interface in-place. Given interface Runnable (which requires a `void run()` method):

```
public class MenuItem {
    public MenuItem(String menuText, Runnable menuResponse) { /* ... */ }
    public static void main(String[] args) {
        MenuItem mi = new MenuItem("Pick me!",
            new Runnable() {                    // <== Anonymous class implements Runnable
                @Override
                public void run() {
                    System.out.println("You picked me!"));
                }
            }
        );
    }
}
```

Lambdas

A lambda is an anonymous method, usually defined where invoked or passed as an argument. It often is used in lieu of an anonymous class when the interface requires only one method to be implemented. The Anonymous Class example above can use a much shorter lambda for the same result:

```
MenuItem mi = new MenuItem("Pick me!", () -> System.out.println("You picked me!"));
```

Other lambda examples include:

- `Thread t = new Thread(() -> threadBody(42));`
- `(int a, int b) -> {int i; while(a>0) {a /= b; ++i;} return i;}`
- `(a, b) -> a + b;`
- `() -> System.out.println("I'm a lambda!")`
- `event -> JOptionPane.showMessageDialog(this, "Clicked!")`

A lambda requires

- A parameter list. Types may be inferred and thus omitted, and if only one parameter is required, the parentheses are also usually omitted as in the last example above.
- `->` signals the lambda, pointing from the parameter list to the method body.
- The method body. If more than one statement is needed, as in the first example, enclose in { } as usual with a return statement (if needed for the lambda's value). If only an expression is needed, omit the { } and the return statement - the value of the expression is automatically returned as in the second example. An expression may also be void, as in the last two examples.

A lambda is prefered to anonymously implement interfaces with a single method to override. If the interface has 2 or more methods to override, you must fall back to an anonymous class.