CSE 1325 Exam 1 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.*

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** – (not on this exam)
- **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

- **Algorithm** – A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute

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

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

Intellectual Property

- **Intellectual Property** – Exclusive right to authors and inventors to their writing and discoveries
- **Trademark** – Symbol or name established by use as representing a company or product
- **Patent** – Exclusive right to make, use, or sell an invention, and authorize others to do the same
- **Copyright** – Exclusive right to print, publish, perform, execute, or record a creative work or its derivatives, and to authorize others to do the same

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.

Import: This adds the name from the specified package to the local namespace. You can either `import java.util.Arraylist; Arraylist<int> al = new Arraylist<>();` OR you can `java.util.Arraylist<int> al = new java.util.Arraylist<>();` - they are equivalent.

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*.

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). The advantage of encapsulation is that we can modify data structures and associated algorithms without affecting code that use them, as long as the interface (public) doesn't change, and classes are easy to reuse without modification. **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*. Know how to model inheritance as shown in "UML Class Diagrams" below, same as `class Subclass extends Superclass implements Interface1, Interface2` in Java. **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.

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 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;            // An object is equal to itself
      if(!(o instanceof Foo)) return false;  // A different type is not equal
      Foo f = (Foo)o;
      return (a == f.a) && (b == f.b);       // Compare relevant fields
  }
  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;`
- An instance is `Foo f = new Foo(3,4);`
- The equals method is `public boolean equals(Object o) { /* 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*)?

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.

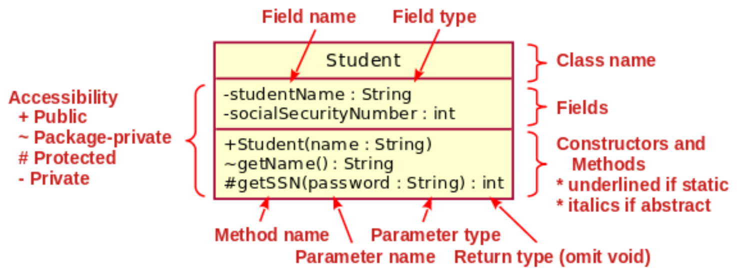
A **destructor** is the opposite of a constructor - it "tears down" the object, freeing any resources necessary. Since Java manages memory via a garbage collector, **Java does not need nor support destructors**.

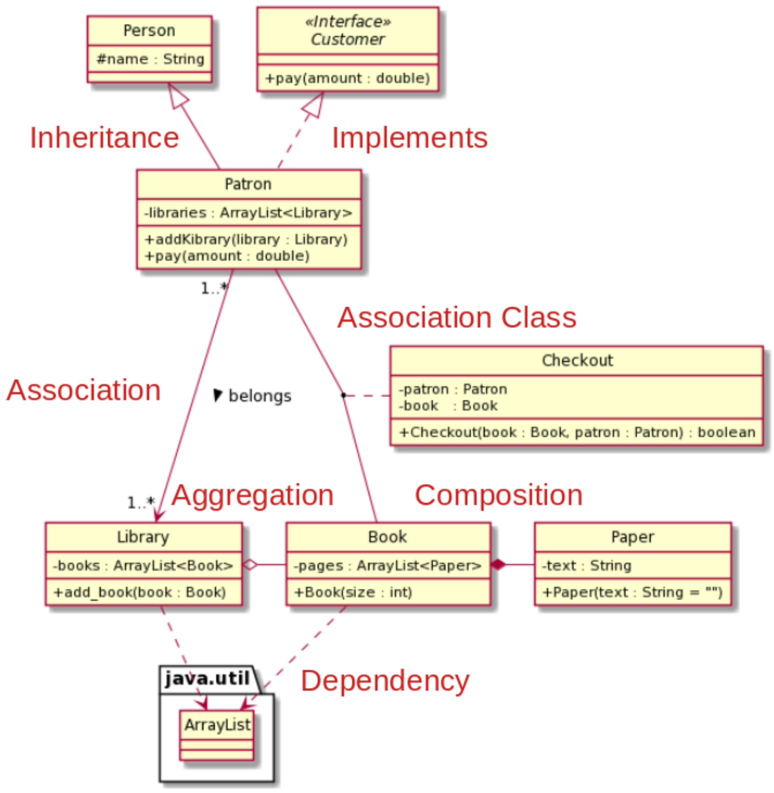
Class Member Options

- **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

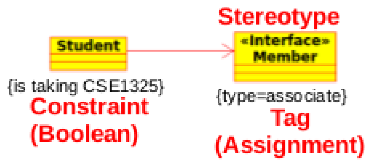
Understand and be able to draw the basic forms of the UML class diagram, the various relationships between, and user-defined extensions below.





Extensions

1. Stereotype – guillemets-enclosed specialization.
2. Tag – curly-brace-enclosed assignment. Always has an =.
3. Constraint – curly-brace-enclosed Boolean condition, restriction, or assertion. May have a comparator, e.g., == or <, or the textual equivalent.



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)

```
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!");
```

- Optionally, the Console class may be used, e.g.,

```
String s = System.console().readLine("Enter your name: ");
System.console().printf("Hello, %s!\n", s);
```

- System.out.printf and String.format are available for formatting output and strings using the familiar printf codes from C. You needn't know the details of these printf codes (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.

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 code should only catch `Exception` objects, NOT `Error` objects. The following prints "Method failed with bad data".

```
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());
        }
    }
}
```

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

Be able to use these additional common Java library members and idioms (common coding techniques) without referencing the documentation.

Functions

- **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.
- **Arrays.sort** - given an array a of any type, sort to default order with `Arrays.sort(a)`.
- **Collections.sort** – given an ArrayList al, sort to default order with `Collections.sort(al)`.

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 *either* of these.

- Using Scanner: `while(scanner.hasNextLine()) text.add(scanner.nextLine());`
- Using Console: `while((line = console.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."""; ``

- Concatenate: `s += ", my name is " + name + '\n' + paragraph;`
- Size of (number of characters): `s.length()`
- Tests: `if(name.startsWith("Ri") && name.endsWith("ce") && name.equals("Rice"))`
- Case-insensitive: `if(name.startsWithIgnoreCase("ri"))` or `name.equalsCaseInsensitive("Rice"))`
- 3-way compare: `int c = name.compareTo("George"); if(c < 0) // less, also == and >`
- 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("/"))`

Related Topics

General Software Development Knowledge

- **Basic command line concepts** such as command line flags, redirection (< and > and 2>) and pipes (|)
- **Simple build.xml files**, including rule names, dependencies, and commands that will bring a rule current (but you need not *write* a build.xml file on the exam!)
- **Basic debugger concepts** such as breakpoints, viewing variables, step into, and step over
- **Basic version control concepts with command line git**, including how to clone, add, commit, and push files; obtain a log and diff; and how to checkout to recover an earlier version.

Intellectual Property

This topic is good for bonus questions. Know the three primary types of intellectual property in the USA of primary interest to computer science and engineering professionals:

- **Trademark** is assigned on first use of a brand (if unique) but may be registered, and is retained until abandoned. Intended to avoid confusion of brands in the marketplace.
- **Patent** is assigned for an invention (including new computer hardware and software technologies) only after application and review, and is usually valid for 14 years. Gives the inventor the right to control the use of their invention.
- **Copyright** is assigned for every work of authorship (software source code as well as literary, artistic, dramatic, etc.), but may be registered, and is retained essentially for a lifetime. Gives the author exclusive control over copying, performing, and modifying their work.

Also know the **primary types of software licenses** necessary for you to reuse other people’s work and to make your work reusable by them, generally from least to most restrictive (examples in parentheses):

- **Public Domain -** all ownership is disclaimed (SQLite)
- **Permissive** (MIT, BSD, Apache) permits use, copying, distribution, and (usually with attribution) derivatives, even proprietary derivatives (BSD Unixis the kernel for Mac OS X)
- **Protective** (GPL 2, 3, Lesser GPL, EPL) permits use, copying, distribution, and derivatives with share-alike rules (Linux, Gimp). GPL 3 also includes important patent clauses.
- **Shareware** permits (sometimes limited) use, copying, and (usually) distribution (Irfanview, WinZip)
- **Proprietary** permits (often restricted) use (Windows, Photoshop)
- **Trade Secret** typically restricts use to the copyright holder