CSE 1325 Exam 3 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** – 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 constants, method signatures, default methods, static methods, 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** – In program that runs in managed memory systems to free unreferenced memory

Class Members Et. Al.

- **Attribute** – A class member variable (also called a "field")
- **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
- **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 derived class inheriting class members from two or more base classes
- **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

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 (in Java e.g., `Foo f = new Foo();`)
- **Global** – Memory for declarations outside of any class or function scope
- **Code** – Read-only memory for machine instructions

Algorithms

- **Algorithm** – A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute
- **Generic** – A Java construct representing a method or class in terms of generic types
- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
- **Iterator** – A standard library abstraction for objects referring to elements of a container

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 be paused while executing, and then safely 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 attributes 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
- **Branch** – A second distinct development path within the same organization and often within the same version control system
- **Fork** – A second distinct and independent development path undertaken (often by a different organization) to create a unique product
- **Baseline** – A reference point in a version control system, using 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
- **Agile Process** – Prioritizing individuals and interactions, frequent delivery of working software, customer collaboration, and flexible response to change
- **Waterfall Process** – Separating software development into long, discrete phases such as Requirements, Design, Implementation, Verification, and Validation
- **Class Hierarchy** – Defines the inheritance relationships between a set of classes
- **Class Library** – A collection of classes designed to be used together efficiently
- **Generalization** – The process of extracting shared characteristics from two or more classes, and combining them into a generalized base class.
- **Specialization** – The process of identifying and specifying subclasses from a base class.
- **Principle of Least Astonishment** – A user interface component should behave as the users expect it to behave
- **Second System Effect** – The common mistake of attempting to include far too many features in (typically) version 2.0, causing catastrophic schedule slips

Object-Oriented Programming

Encapsulation is simply bundling data (attributes) 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 attribute 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 inherit (that is, they are directly accessible from the subclass); **private members and constructors do NOT inherit** but can be reached 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 and implementation in UML 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.

Polymorphism is the provision of a single interface (superclass methods) to multiple subclasses, enabling the same (overridden) 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 derived object's method is invoked. `ArrayList<Base> al = new ArrayList<>(); v.add(new Derived()); for( var d : v) d.foo();` is an example of polymorphism, assuming Derived.foo() overrides Base.foo().

Custom Types

Java supports 2 custom type mechanisms:

- **Class** – Encapsulates attributes (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 = (Complex)o;
      return (a == f.a) && (b == f.b);
  }
  private int a;
  private int b;
}
```

- The class is `Foo` and is public (visible in all packages)

- The attributes are `int a;` and `int b;` and are private (visible only within the Foo class)
- The 2 public constructors are `Foo() {this(0, 0);` 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 *attributes* (do these objects have *equal attributes*)?

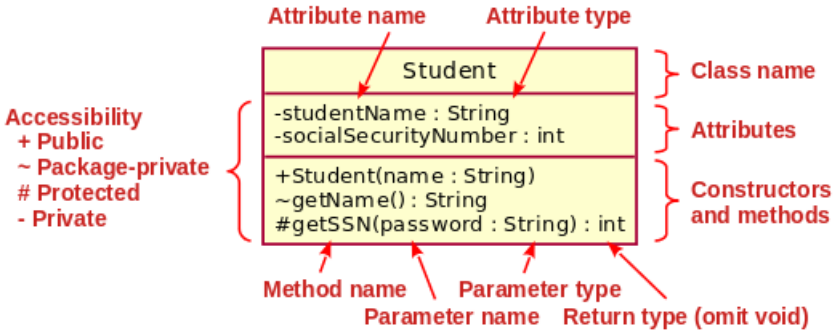
A **default constructor** (with no parameters, like `Foo()` above) is provided by default, but **only** if no non-default constructor is defined. Any number of constructors may be defined ("overloading"), as long as the types of each set of parameters is unique (called the “parametric signature” of the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

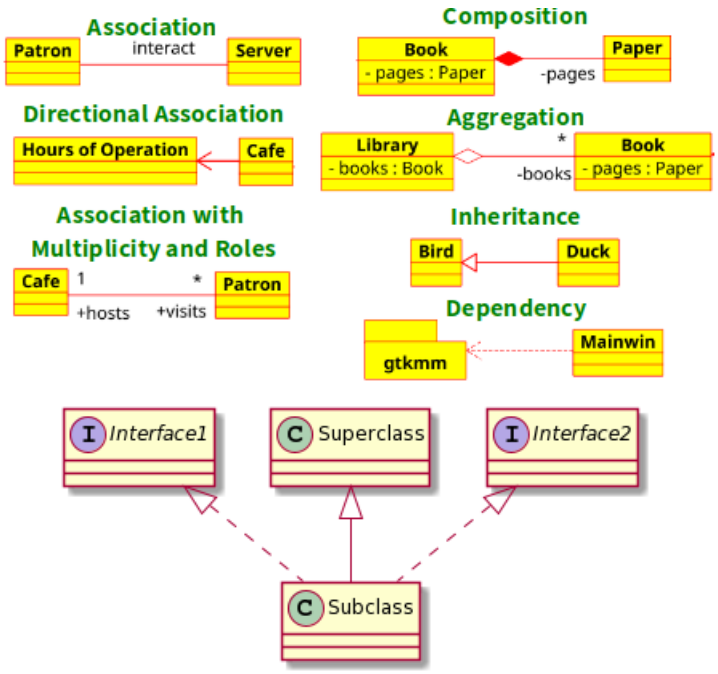
"Overloading" and "parametric signature" also apply to methods. All of the other elements of a method's declaration, such as modifiers, return type, parameter names, exception list, and body are NOT part of its parametric signature.

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, it does not need nor support destructors.

UML Class Diagram

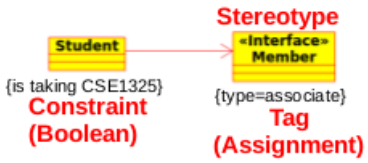
Understand and be able to draw the basic forms of the UML class diagram to the right, the various relationships between classes on lower right, and user-defined extensions below.





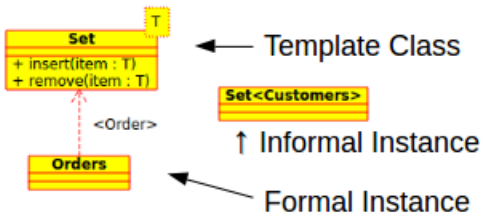
Extensions

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.



Generics

UML calls these "Parameterized Types".



General Java Knowledge

Java syntax: (from CSE 1320) assignments, operators, relationals, naming rules, the 4 most common primitives (boolean, char, int, double) and common classes (String, 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*.

Import: This adds the specified base name 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.

Packages: Purpose, how to use them, and how to build them.

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 other classes within the same package only
- **Public** – Accessible anywhere

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*.

Class member options

- **Non-static attribute** – A unique value for every object
- **Static attribute** – 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 attributes
- **Static method** – Called via object (`foo.bar()`) or class (`Foo.bar()`), and can access only static attributes
- **Non-final attribute** – Value can be changed by any code with visibility
- **Final attribute** – 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) {/*...*/}`

Streams

Streams generalize I/O as a sequence of characters to or from any source - the console, keyboard, file, a string, a device, whatever.

- Console input is `System.in`
- Console output is `System.out`
- Console errors go to `System.err`, e.g.,

```
java.util.Scanner in = new java.util.Scanner(System.in);
System.out.print("Enter a positive int: ");
int i = in.nextInt();
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.printf and String.printf are available for formatting output and strings using the familiar printf codes from C. Alternately, String.format uses a more pictoral formatting system. You needn't know the details of these (I'll provide a handout if needed), but know how to use them.

File I/O

While Java has a wide range of classes for managing files, the ones of primary interest for text are BufferedWriter / FileWriter with the write method and BufferedReader / FileReader with the readLine method. For binary file I/O, it's BufferedOutputStream / FileOutputStream with the write method and BufferedInputStream / FileInputStream with the read method.

The general pattern relies on try-with-resources. Resources (classes that implement the Autocloseable interface) are defined in parentheses immediately after try and before the opening brace. If an exception occurs within the try-with-resources, the close() method of each resource is called automatically.

BufferedWriter implements the Autocloseable interface (your classes can, too!). In the following example, we know our br stream will be properly closed on any exception within the try-with-resources.

```
try (BufferedWriter br = new BufferedWriter(new FileWriter(args[0]));) {
    br.write("Hello, world!\n");
} catch (Exception e) {
   System.err.println("Failed to write: " + e);
}
```

Error Handling

Know why exceptions are usually superior to returning an error code, and how to instance, throw, catch, and handle an exception. 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 (std::runtime_error e) {  // Catch the runtime_error
            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 Exception's 4 constructors, though you may also add your own.

```
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). The use of the assert keyword to check them, e.g., `assert errorCode == 0;` If error_code is 0, nothing happens, but if not zero, the program aborts with an “assertion failed: error_code == 0” message on System.err. Alternately, the message may be provided in the assert, e.g., `assert errorCode == 0 : "Non-zero error code";`

Miscellaneous

Be able to use these additional common Java library members without referencing the documentation.

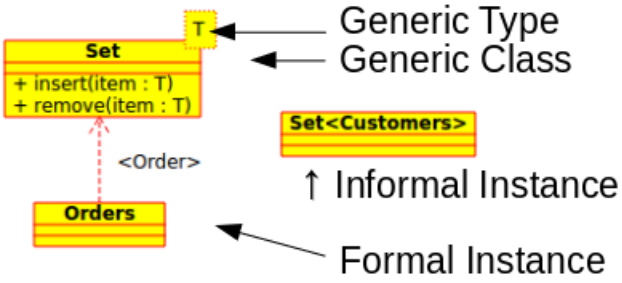
Functions

- **Math.random()** – returns a random double between 0 and 1. Adjust , e.g., `(int) (Math.random()*20-10)` gives an int between -10 and 10.
- **Collections.sort** – given an ArrayList al, sort to default order with `Collections.sort(al)`.

Advanced Topics

Generics

A generic is a Java construct representing a class or method in terms of generic types. Generics enable us to write algorithms independent of the types to which the algorithms can apply – called "generic programming". Generics are represented in the UML via a dashed “T” box to upper right of class, and instanced via references (or informally in the name).



Be able to code simple examples, for example:

```
class LIFO<E> {
    private ArrayList<E> lifo;
    public void push(E value) {lifo.add(value);}
    public E pop() {return lifo.remove(lifo.size()-1);

    public void main(String[] args) {
        LIFO<String> lifo; // Create a LIFO that manages strings
    }
 }
```

Also be able to constrain the generic type for both methods and classes, and explain what the constraint means. For example:

- **Upper bounded:** `void process(List<? extends Number> list)` - list must be of class Number or one of its subclasses (via inheritance or interface)
- **Unbounded:** `void printAll(List<?> list)` - list may be of any type
- **Lower bounded:** `void addNumbers(List<? super Integer> list)` - list must be of class Integer or one of its superclasses (via inheritance or interface)

Standard 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. Of course, javax.swing with java.awt supports our GUI programs.

Collections / Maps

The java.util module defines the following, which you should know how to use give basic documentation (except basic ArrayList and HashMap methods).

- **Algorithms** like sort, search, and copy
- **Iterators** which allow algorithms to generically manipulate collections
- **Collection interface (typically implemented as generics) with subinterfaces**
    - **List** implemented as ArrayList and LinkedList
    - **Set** and **SortedSet** implementd as the unsorted HashSet and sorted TreeSet (Remember, you must define your class' equals() and hashCode() to store its instances in a for TreeSet)
    - **Queue** and **Deque** (double-ended queue) implemented as PriorityQueue (a TreeSet that allows duplicate elements) and ArrayDeque (which makes a great stack!)
- **Map** and **SortedMap** interfaces typically implemented as generics, such as the unsorted HashMap and sorted by keys TreeMap (same caveat on the key type as for TreeSet)

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

- **Store / Retrieve elements in a Collection**
    - [Collection](Collection<? extends E> c) – constructs new collection (e.g., Set) from any other collection (e.g., List)
    - add(E e) – append a new element
    - remove(E e) – removes an existing element
    - addAll(Collection<? extends E> c) – append the elements of the collection object
    - removeAll(Collection<?> c) – removes all elements found in the collection parameter
    - retainAll(Collection<?> c) – removes all elements NOT found in the collection parameter
    - iterator() – returns an iterator over the container
- **Manipulate Elements in a Collection**
    - clear() – removes all of the elements
    - size() – number of elements
    - isEmpty() – true if size() == 0
    - contains(Object o) – true of the object is an element
    - containsAll(Collection<?> c) – true if all objects are elements
    - toArray() – returns an array of E with the same elements
    - toArray(T[] a) – returns an array of the type of parameter a with the same elements

- **Manage Pairs in a Map**
    - get(O k) – returns value at k, or null if missing
    - remove(O k) – removes pair at k if present
    - clear() – removes all of the pairs
    - size() – number of pairs
    - isEmpty() – true if size() == 0
    - keySet() – returns a Set of all keys in the Map
    - values() – returns a Collection of all values in the Map
    - entrySet() – returns a Set of Entry objects which support getKey() and getValue() methods
    - containsKey(O k) – true if k is a key in the Map
    - containsValue(O v) – true if v is a value in the Map

Iterators

Containers 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 sort method without documentation, and be able to use the other following algorithms with brief documentation.

- 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 (unsorted) collection
- Collections.swap(Collection c, int index1, int index2) swaps the two elements
- 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 value in (sorted!) c
- E Collections.min(Collection c) returns the smallest valued element in c
- E Collections.max(Collection c) returns the largest valued element in c
- int Collections.frequency(Collection c, E value) returns the number of times value is in c
- boolean Collections.disjoint(Collection c1, Collectionc2) returns true if c1 and c2 have no common elements

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.

```
JButton button = new JButton("Hello, Java!");
button.addActionListener(
    new ActionListener() {  // anonymous class implements ActionListener interface
        public void actionPerformed(ActionEvent event) {
            String str = event.getActionCommand(); // Button text
            System.out.println(str);
        }
    }
);
```

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.

- `(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.

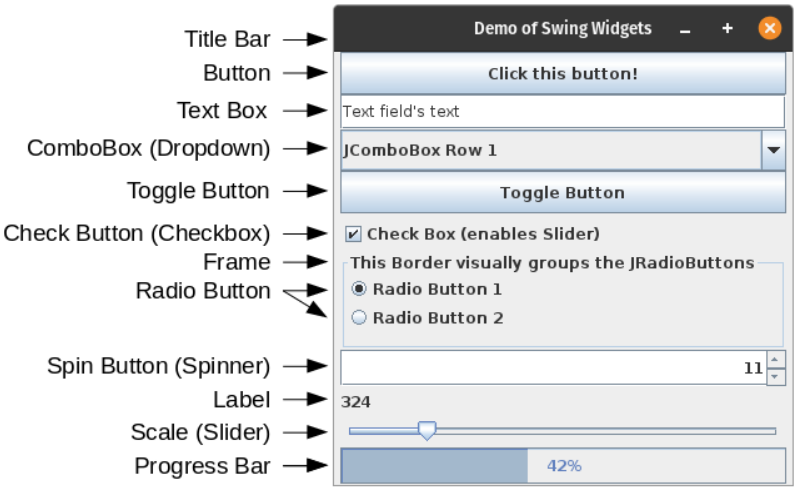
Graphical User Interfaces in Swing

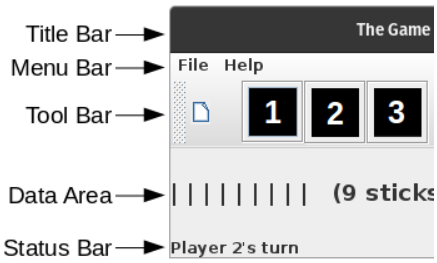
In a Command Line Interface (CLI), the program is in control and queries for information. In a Graphical User Interface (GUI), the user is in control and the program reacts to their actions via *action listeners* (also called observers or callbacks).

Common portable Java GUI toolkits include Swing, JavaFX, Java-Gnome, and QtJambi. CSE1325 lightly covers Swing, which is included in the Java Development Kit.

Coding: Be familiar with the following widgets by name (given its appearance), by appearance (given its name or code), and how to code. If asked to write code using them you will find a subset of the relevant docs from the reference manual (https://docs.oracle.com/en/java/javase/16/) for them at the end of the exam:

- **JFrame, JDialog, JOptionPane:** How to derive from and extend JFrame to create your main window. How to instance, configure, run, and get data from a JDialog, including pre-defined classes derived from Dialog such as JFileChooser and JColorChooser. How to create quick dialogs using JOptionPane's static methods showMessageDialog, showInputDialog, showConfirmDialog, and showOptionDialog.
- **BorderLayout, BoxLayout, FlowLayout, GridLayout, and GridBagLayout:** How to use them to control widget layouts in a container such as **JPanel**.
- **JMenuBar, JMenu, and JMenuItem:** How to construct a main menu with action listeners.
- **JToolBar, JButton, ImageIcon:** How to construct a toolbar with action listeners.
- **JLabel, JButton, JTextField, JComboBox<E>, JSpinner, JSlider, JProgressBar, JToggleButton, JCheckBox, and JRadioButton:** Some of the other widgets that we used in class. How to use these in a JOptionPane static method, and how to create a custom dialog with them.
- **JPanel:** Extended to create a canvas for drawing. How to override paintComponent to implement your drawing when called by Swing, and how to use its Graphics2D parameter to create, translate, setColor, setStroke, drawString, drawLine, drawRect, and drawImage with it. The OS calls JPanel.paintComponent(Graphics) as needed, such as when the window is resized.





Callbacks are registered with widgets via the addActionListener method, passing the callback function or specifying a lambda (know how to do both).

The general pattern for a GUI main window class is

1. Extend (inherit from) JFrame,
2. **In the constructor**
    a. instance and add widgets,
    b. set widget properties,
    c. register widget action listeners (“observers”),
3. Set the window visible, which initiates Swing's GUI main loop.

For the exam:

- Be able to code simple windows, dialogs, and panels from scratch.
- Be able to fill in the blanks in a basic Nim-like program with the correct Swing-related method calls, data structures, and supporting concepts.
- Be able to draw the window or dialog that would result from code provided on the exam.

Related Topics

General Software Development Knowledge

- **Basic command line concepts** such as command line flags, redirection (< and >) and pipes (|)
- **Simple build.xml files**, including rule names, dependencies, and commands that will bring a rule current

Anti-Patterns

Anti-patterns are errors committed during software engineering projects that are common enough to have been given well-known names. Know the six examples covered in class and how to avoid them.

- **Comment Inversion / Documentation Inversion** – Writing obvious and thus unhelpful comments and documents. Consider what the reader actually wants to know instead.
- **Error Hiding** – Catching and ignoring / obscuring an error. Only catch exceptions to which you can helpfully respond.
- **The "God Class"** - Too much functionality generalizes to the root of the class hierarchy. Use UML to identify a helpful hierarchy, or else use a single class instead.
- **The Big Ball of Mud** – A legacy system that “happened”, lacking any discernable architecture. Avoid the temptation to rewrite from scratch; the code contains a lot of subtle aggregated domain knowledge that would be lost. Instead, (1) write regression tests (2) define a target architecture (3) migrate small sections in baby steps to the new architecture, using the regression tests to verify no loss of functionality (4) continue until it’s “maintainable enough”
- **Not Invented Here (NIH) Syndrome** – Reimplementing an existing solution because “we can write it better”. Instead, adapt the existing solution to the need (think patterns like façade and facade), addressing any real issues such as licensing, support, etc.
- **Cargo Cult Programming** – Inclusion of code, language features, data structures, or patterns without understanding them thoroughly. Instead, thoroughly understand everything that you include in your designs and implementations, learning as you go