Rishabh Indoria

Programming Concepts in Java

December 23, 2023

1 Week 1

1. Memory Management
   - Scope: When the variable is available for use
   - Lifetime: How long the storage remains allocated
   - Memory Stack: Create activation record when function is called. Activation records are stacked, Popped when functions exit, Control links points to start of previous record, Return value link tells where to store result. Parameters are part of the activation record of the function. Two ways to initialize a parameter, call by value (Copy the value) and Call by reference (Parameter points to the same location as argument)
   - Heap: Stores dynamically allocated variables, Storage outlives activation record, accesses via some variable in stack. Need to free storage inside heap, else we get dead storage (unusable). Manual Memory Management or Automatic Garbage Collection.
2. Abstraction and Modularity
   - Use Refinement to divide solution into components
   - Build a prototype of each component to validate design
   - Interfaces: What is visible to other components
     Specification: Behaviour of the component as visible through interface
   - Improve each component independently, preserving interface and specification.
   - Control Abstraction: Functions and Procedures, Encapsulate a block of code (Reuse in different contexts)
   - Data Abstraction: Abstract data types, Set of values along with operation permitted on them, Internal representation should not be accessible, Interaction restricted to public interface.
   - Implicit reuse of implementations: subtyping, inheritance.
3. Object-Oriented Programming
   - Object: Similar to an abstract data type, hidden data with set of public operations, All interaction through operations.
   - Uniform way of encapsulation different combinations of data and functionality.
   - Abstraction: Public interface, Private implementation, changing the implementation should not affect interactions with the object.
   - Subtyping: If A is a subtype of B, whenever an object of type B is needed, an object of type A can be used. If f() is a method in B and A is a subtype of B, every object of A also supports f() (Implementation of f() can be different)
   - Dynamic Lookup: How the method acts is a dynamic property of how the object is implemented. A variable v of type B can refer to an object of subtype A. Static type of v is B, but method implementation depends on runtime type A.
   - Inheritance: Re-use of implementations
   - Dequeue: Double ended Queue, We can implement a stack or a Queue by using this. This makes Dequeue a subtype of Stack and Queue.
4. Classes and Objects
   - Class: Template for a data type, How data is stored, How public functions manipulate data
   - Object: Concrete instance of template, Each object maintains separate copy of local data, Invoke method on objects.
   - Objects implicitly call a constructor function when they are created, in python the constructor has the name __init__()

2 Week 2

1. Hello World in Java
   public class helloworld{
   public static void main(String[] args){
   System.out.println("hello, world");
   }}

   - All code in Java lies within a class, Modifier public specifies visibility.
   - Fix a function name that will be called by default. convention is to call it main().
   - Need to specify input and output types for main().
   - Modifier static, function that exists independent of dynamic creation of objects.
   - Each class is defined in a separate file with the same name, helloworld.java
   - Programs are usually interpreted on Java Virtual Machine(JVM), provides uniform execution environment across operating systems, Semantics of Java is defined in terms of JVM.
   - javac compiles into JVM bytecode, javac helloworld.java creates helloworld.class.
   - java helloworld interprets and runs bytecode in helloworld.class

2. Data types

   - All data should be encapsulated as objects.
   - Scalar types: int(4), long(8), short(2), byte(1), float(4), double(8), char(2) and boolean(1).
   - single quotes represent characters and double quotes denote strings
   - Append f after number for float, else interpreted as double.
   - Modifier **final** indicates a constant.
   - When both arguments are integer, / is integer division.
   - No exponentiation operator, use Math.pow(a,n), a^n.
   - **String** is a built-in class, string constants enclosed in double quotes. + is overloaded for concatenation. Strings are **not** an array of characters, instead invoke method substring.
   - **Arrays** are also objects, int[] a or int a[]. Combine as int[] a = new int[100]; length variable gives size of array, whereas length method gives size of strings.
   - Java does automatic garbage collection.

3. Control Flow

   - if (condition){...} else if (condition){...} else{...}
   - while(condition){...}, do{...}while(condition)
   - for(initialization; condition; update){...}
   - for(type x: a){ do something with x; }
   - switch(x){case -1: {...} ...}

4. Classes and Objects

   - A class is a template for an encapsulated type, An object is an instance of a class
   - this is a reference to the current object
   - Accessor and Mutator methods
   - Constructor has the same name as the class
     Overloading: We can have multiple constructors with different signatures
     Signature: input parameters, output parameters.
     A later constructor can call an earlier one using this(params).
   - public class Date {
     private int day, month, year;
     public Date(int d, int m, int y) {
     this.day = d;
     this.month = m;
     this.year = y; }
     public int getDay() {
     return (this.day); }

```
public int getMonth() {
return (this.month); }
public int getYear() {
return (this.year); }
}
```

- new creates a new object
  Date d;
  d = new Date();

5. Basic Input and Output

   - Console cons = System.console();
     String username = cons.readLine();
     char[] passwd = cons.readPassword();
   - Scanner in = new Scanner(System.in);
     String name = in.nextLine();
     int age = in.nextInt();
   - System.out.println();
     System.out.print();
     System.out.printf();

3 Week 3

1. Philosophy of OOPS

   - Algorithms + Data Structure = Programs, Data representation comes later
   - OOPS reverses the process. First identify the data we want to maintain and manipulate, then identify algorithms to operate on the data.
   - State is the information in the instance variable.
   - Encapsulation: should not change unless a method operates on it.
   - Identity: Distinguish between different objects of the same class.
   - Robust design minimizes dependencies, or coupling between classes.

2. Subclasses and Inheritance

   - use *extends*, public class Manager extends Employee.
   - Manager objects do not automatically have access to private data of parent class.
   - Use parent class's constructor using *super*.
   - In general, a subclass has more features than the parent class. Subclass inherits instance variables, methods from parent class.
   - Subtyping, Employee e = new Manager(...) works because Manager is capable of doing whatever Employee is capable of doing.

3. Dynamic Dispatch and Polymorphism

   - Overrides definition of parent class if function signature is the same.
   - e can only refer to methods in Employee, but if methods are overridden in Manager, then Dynamic dispatch will choose the most appropriate method.
   - This type of dynamic dispatch also known as runtime polymorphism or inheritance polymorphism.
   - Java class Arrays has a method sort to arbitrary scalar arrays, made possible by overloaded methods.
   - Overloading: multiple methods, different signatures, choice is static.
   - Overriding: multiple methods, same signatures, choice is static.
   - Dynamic Dispatch: multiple methods, same signature, choice made at run-time.
   - Type casting: ((Manager) e).setSecretary(a), can check instance with *instanceof*

4. Java Class Hierarchy

   - Multiple Inheritance: C3 extends C1,C2. Java does not allow multiple inheritance. C++ allows this if C1 and C2 have no conflict.

- Universal superclass: Object
- public boolean equals(Object o), checks pointer equality
- public String toString(), converts value of instance variable to string.
- Both equals and toString are available in Object class.
- Can exploit tree structure to write generic functions, public int find(Object[] objarr, Object o)
- Overriding requires function to have the same signature. Overriding looks for the "closest" match.

5. Subtyping vs Inheritance

- If B is a subtype of A, wherever we require an object of type A, we can use an object of type B.
- B inherits from A if some functions for B are written in terms of functions of A.
- Subtyping: Compatibility of interfaces.
- Inheritance: Reuse of Implementations.
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two.

6. Java Modifiers

- *public* vs *private* to support encapsulation of data.
- *static*, for entries defined inside classes that exist without creating objects of the class.
- *final*, for values that cannot be changed.
- Modifiers static and final are orthogonal to public or private.

4 Week 4

1. Abstract Classes and Interfaces

- Sometimes we collect together classes under a common heading. We want to force every subclass to define a function.
- Provide an abstract definition: public abstract double perimeter();
  Forces subclass to provide a concrete implementation.
- If a function is abstract, the entire class must be abstract. But we can still declare variables whose type is an abstract class.
- An interface is an abstract class with no concrete components(Only contains abstract definitions). A class that extends an interface is said to implement it. Can implement multiple interfaces
  public interface Comparable{
  public abstract int cmp(Comparable s); }
  public class Circle extends Shape implements Comparable{}

2. Interfaces

- Cannot express the intended behaviour of any function explicitly.
- We can have static functions inside any interface.
- Provide a default implementation for some functions. Invoke like normal method, using object name. class can override these by providing an implementation.
  public default int cmp(Comparable s){}
- If there is a conflict between two interfaces then subclass must provide a fresh implementation.
- If there is a conflict between a superclass and an interface, we will get the function from the superclass unless it is overridden.

3. Private Classes

- Nested within public classes. Also called Inner class.
- Objects of private class can see private components of enclosing class.

4. Callbacks

- Myclass m creates a Timer t
- Start t to run in parallel, Myclass m continues to run.
- Timer t notifies Myclass m when the timer limit expires. Assume Myclass m has a function timerdone().

- Timer implements Runnable, which indicates that Timer can run in parallel.

5. Iterators

   - public interface Iterator{
     public abstract boolean has_next();
     public abstract Object get_next();}
   - Need a"pointer" to remember position of the iterator.
   - Create an Iterator object and export it.
     public Iterator get_iterator(){
     Iter it = new Iter();
     return it;}
   - new Java for over lists implicitly constructs and uses an iterator.
     for(type x : a){ do something with x;}

5 Week 5

1. Polymorphism

   - Refers to the effect of dynamic dispatch, Every object "knows" what it needs to do.
   - Refers to behaviour that depends on only a specific capabilities(Structural Polymorphism).
   - Java added Generic Programming, class LinearList<T> holds value of type T, public T head() must return a value of same type T. public <S extends T, T>.

2. Generic Programming

   - public <S extends T, T> int something(S[] src, T[] tgt). S type must extend T type.
   - LinkedList<Ticket> ticketList = new LinkedList<Ticket>()
   - public static void printList(LinkedList<? > l){}, Only use if type is not being used anywhere. ? is a wildcard type. Can use ? extends T and ? super T
   - At run time, all type variables are promoted to Object, or the upper bound if one is available.

3. Reflection

   - Reflection Programming is the ability of a process to examine, introspect, and modify its own structure and behaviour.
   - Two component involved, Introspection and Intercession.
   - Employee e = new Manager(...); if(e instanceof Manager){}
   - Can extract the class of an object using getClass(), returns an object of type Class that encodes class information.
   - Class c = obj.getClass();
     Object o = c.newInstance();
     Class a = c.forName("Manager");
   - Can extract details about constructors, methods and fields of the class. Constructor[] con = c.getConstructors(); Method[] met = c.getMethods(); Field[] field = c.getFields();
     Class params = con[i].getParameterTypes();
     met[3].invoke(c, args); field[3].set(c, value)
   - BlueJ, a programming environment to learn Java.

6 Week 6

1. Collection

   - Abstracts properties of grouped data, Arrays, lists, sets. But not key-value structures.
   - add() adds to the collection, iterator() gets an object that implements Iterator interface.
   - Iterator has a remove() method, which removes the last accessed element using next(). To remove consecutive elements, must interleave a next().
   - addAll(from) adds elements from a compatible collection. removeAll(c) removes elements present in c, remove() different from Iterator.

- To implement the Collection interface, need to implement all these methods. "Correct" solution is to provide default implementations in the interface. Java has AbstractCollection abstract class implements Collection. This class has default implementations.

2. Maps
   - Key-value structures come under the Map interface. Two type parameters, K for key type and V for Value types. get(k) fetches value for key k. put(k, v) updates value for key k also returns the old value.
   - getOrDefault(Object key, V defaultValue) returns value for key if key exists, else returns default value.
   - putIfAbsent(Object key, V value) to initialize a missing key.
   - merge(Object key, V newValue, Integer::sum), initialize to newValue if no key else combine current value with new Value using Integer::sum.
   - keySet(), values(), entrySet(), methods to extract keys and values.

7 Week 7

1. Exceptions in Java
   - User input, Device errors, Resource information, Code errors.
   - Signalling errors, code that generates an error raises or throws an exception.
   - Notify the type of error, Caller catches the exception and takes corrective action or passes the exception back up the calling chain. Declare if a method can throw an exception.
   - All exception descend from class *Throwable*, two branches *Error* and *Exception*.
   - Error is relatively rare and not the programmer's fault. Exception has two sub-branches *RunTimeException*, checked exception.
   - Enclose code that may generate an exception in a *try* block, exception handler in *catch* block. Can catch more than one type of exception. Order catch blocks by argument type, more specific to less specific.
   - Create an object of exception type and throw it, *throw new EOFException(errormsg)*. Can also pass diagnostic string errormsg in the object.
   - Declare exception thrown in header. *String readFile(String filename) throws FileNotFoundException, EOFException* {}. Can throw any subclass of declared error.
   - If method that can throw an error is called, then the error must be handled. Need not advertise unchecked exceptions Error, RunTimeExceptions.
   - Customized Exceptions: Define a new class extending Exception.
   - *Throwable* has additional methods to track chains of exceptions getCause(), initCause().
   - To clean up resources, add a *finally* block.

2. Packages
   - Java has an organizational unit called package.
   - Can use *import* to use packages directly, *import java.math.BigDecimal*. All classes in *import java.math.**
   - To include a class in a package add a package header, *package in.ac.iitm.onlinedegree*.
   - *protected* means visible within subtree, so all subclasses. protected can be made *public*.

3. Assertions
   - *assert* the property you assume to hold. *assert* $x \geq 0$ *: x;*.
   - If assertion fails, the code throws *AssertionError*. This should not be caught.
   - enabled or disabled at runtime, *java -enableassertions MyCode*. Can selectively turn on assertions for a class *java -ea:MyClass MyCode* or a package.

4. Logging
   - Typical to generate messages within code for diagnosis.
   - Naive approach is to use print statements. Instead, log diagnostic message separately.
   - *Logger.getGlobal().info("Edit→Copy menu item selected);*
   - Suppress logging execution, *Logger.getGlobal().setLevel(Level.OFF);*
   - Can create custom logger,
     private static final Logger myLogger = Logger.getLogger("in.ac.iitm.onlinedegree");
   - Seven logging levels: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. By default, the first three are logged. Can set a different level, *logger.setLevel(Level.FINE);*.

8 Week 8

1. Cloning
    - Normal assignment creates two references to the same object.
    - Object defines a method *clone()* that returns a bitwise copy of the object. Bitwise copy is **shallow copy**, as it copies the references to nested objects instead of initializing fresh ones.
    - **Deep copy** recursively clones the nested objects. Override the shallow clone() from object. *public Employee clone()*{} and cast *super.clone()*.
    - To allow clone() to be used, a class has to implement a *Cloneable* interface. Object.clone() throws CloneNotSupportedException, catch or report this exception.
2. Type Inference
    - Use generic *var* to declare variables.
    - Only allowed for local variables that are initialized when they are declared.
3. Higher Order Functions
    - *Comparator<T>* interface provides signature for comparison function. public int compare(T s1, T s2). Can pass to *Arrays.sort(someArray, Tcompare)*
    - Interfaces that define a single function are called functional interfaces.
    - Instead of implementing functional interfaces we can use lambda expressions. (Parameters) -> body, return value and type are implicit. *Arrays.sort(strArray, (s1, s2) -> s1.length() - s2.length())*. More complicated functions can be defined as a block.
    - If the lambda expression consists of only a single function call, we can pass that function by name. *map<String, Integer> scores = scores.merge(bat, newscore, Integer::sum)*, the corresponding lambda expression is *(i,j) -> Integer::sum(i,j)*.
4. Streams
    - Alternative approach to Iterators. *words.stream().filter(w -> w.length > 10).count()*
    - Stream processing is declarative, Processing can be parallelized. *words.parallelStream().filter(w -> w.length > 10).count()*
    - Apply *stream()* to a collection, use static method *Stream.of()* for Arrays. *Stream.generate(Math::random)* generates a stream from a function, provide a function that produces value on demand with no argument. *Stream.iterate(0,n -> n<100,n -> n+1)* generates a stream of dependant values, requires an initial value, a function to determine next value based on previous value, and terminate using a predicate.
    - *filter()* to select elements, *map()* applies a function to each element in the stream. *flatMap()* collapses nested list into a single stream in case map generates a list for each element.
    - *limit(n)* makes a stream finite. *skip(n)*, discard first n elements. *takeWhile(n ->n>=0.5)* stops when element matches the criterion. *dropWhile(n -> n<=0.05)* starts when element matches the criterion.
    - *count()* counts the number of elements. *max()* and *min()*, largest and smallest value seen, requires a comparison function. *findFirst()* gets the first element.

9 Week 9

1. Optional Types
    - If a stream is empty, max of this stream would be null. To handle this, we use optional type *Optional<Double> maxNum*
    - Use orElse to pass a default value, *Double val = maxNum.orElse(6.5)*
    - orElseGet to call a function which generates replacement for missing value.
    - orElseThrow to generate an exception whenever missing value is encountered.
    - ifPresent, to test if the value is present and process it.
    - isPresentOrElse, to specify an alternative action if value is not present.
    - Optional.of(v) creates an optional type of variable v, Optional.empty() creates an empty optional, Optional.ofNullable(v) creates empty optional if v is null.

- *map* applies function to value if present
- or returns value as is or specified value if original is null.
- Use *flatmap(T::g)* to invoke function g from class T using Optional<T>.

2. Collecting Output
   - Can convert a stream into an array using *toArray()*. Pass array constructor to get more specific type of array, *toArray(String[]::new)*
   - Can also use forEach with a suitable function.
   - Can also convert to a collection, *mystream.collect(Collection.toList())* or *mystream.collect(Collection.toCollection(TreeSet::new))*
   - *summarizingInt*, creates *IntSummaryStatistics* that stores max, min, sum, average. Retrieve using *getMax*, *getSum*. Similar for Double and Long
   - Convert a stream to a map using *mystream.collect(Collection.toMap(key, value, (existingVal, NewVal) -> existingVal))*. Use *Function.identity()* to store the object as value
   - Cam also group stuff, *mystream.collect(Collection.groupingBy(some function))*

3. Input/Output Streams
   - Input and Output are raw uninterrupted bytes of data. Different from streams generated by Stream object.
   - Read raw bytes from a file, pass to a Stream that reads text, Some process, Generate binary data, pass to a Stream that writes raw bytes to a file.
   - *var in = new FileInputStream("input.class");*
     InputStream in = ...
     int bytesAvailable = in.available()
     while(bytesAvailable > 0){
     var data = new byte[bytesAvailable];
     in.read(data);}
   - *var ot = new FileInputStream("output.bin", false);* for overwrite, true for append.
     OutputStream ot = ...
     byte[] values = ...
     out.write(values);
     in.close();
     out.flush();
   - Use Scanner class for taking input
     var scin = new Scanner(in);
     Can use methods nextLine, next, nextInt, hasNext.
   - Use PrintWriter to write text
     var pot = new PrintWriter(ot);
     Use println, and print.
   - To read and write binary data, use DataInputStream and DataOutputStream. Contains methods read, readInt, readDouble, readChar etc. similar for write.
   - Buffering an input stream, reads blocks of data which is more efficient.
     var din = new DataInputStream(new BufferedInputStream(new FileInputStream("input.class")));

4. Serialization
   - To write objects, Java has ObjectOutputStream
     var ot = new ObjectOutputStream(new FileOutputStream("employee.dat"));
     Use writeObject to write an object.
   - Similarly, ObjectInputStream, readObject.
   - Class has to allow serialization
     public class Employee implements Serializable
   - Some objects should not be serialized, mark such as transient.
     private transient label
     defaultWriteObject writes out the object without all transient fields, then explicitly write all transient fields.
     defaultReadObject reads out the object without all the transient fields, then explicitly read all transient fields.

10 Week 10

1. Concurrency: Threads and Processes
   - Multiprocessing: Single processor executes several computations "in parallel", Time slicing to share access.
   - Process: private set of local variables, time slicing involves saving the state of one process and loading the suspended state of another.
   - Thread: Operated on same local variables, Communicate via "shared memory", Context switches are easier.
   - Have a class extend Thread, Define a function run(). Invoking object.start() initiates object.run() in a separate thread. Directly calling run() also works. Thread.sleep(t) suspends the thread for t milliseconds.
   - Cannot always extend Thread, instead implement Runnable, to use it explicitly create a Thread and start it.
   - Race Condition: concurrent update of shared variables, unpredictable outcomes. Avoid by insisting no two functions with shared variables interleave. Mutually exclusive access to critical regions of code.
2. Mutual Exclusion
   - At most one thread at a time can be in a critical section
   - shared variable turn, which decides which thread gets to access the function. But one thread would be locked out permanently if the other shuts down(Starvation).
   - Make a variable for both thread that indicated whether that thread is currently accessing the function or not. If both threads try to access simultaneously, both of them would be locked out permanently(Deadlock).
   - Peterson's Algorithm, combine the previous two approaches.
   - Lamport's Bakery Algorithm: Each new process picks up a token that is larger than all waiting processes. Lowest token number gets served next, still need to break ties.
3. Test and Set
   - At most one thread at a time can be in a critical section, At most one thread at a time can be in a critical section.
   - Semaphores: Programming language support for mutual exclusion.
   - Dijkstra's semaphores: Integer variable with atomic test-and-set operation.
   - A semaphore supports two atomic operations
     to pass, P(s):
     if(s > 0){decrement s}else{wait for s to become positive}
     to release, V(s):
     if(there are threads waiting for s to become positive){wake one of them up}else{increment s}
   - Semaphores guarantee Mutual exclusion, Freedom from starvation and deadlock.
   - Too low level, No clear relationship between a semaphore and the critical region that it protects, All threads must cooperate to correctly reset semaphore, Cannot enforce that each P(S) has a matching V(S), Can even execute V(S) without having done P(S).
4. Monitors
   - Attach synchronization control to the data that is being protected
   - Monitor is like a class,
     Data Definition: To which access is restricted
     Collections of functions operating on this data all are implicitly mutually exclusive
     monitor bank_account{}
   - Implicit queue associated with each monitor, Contains all processes waiting for access
   - *wait()*: All other processes are blocked out while this process waits, Need a mechanism for a thread to suspend itself and give up the monitor, A suspended process is waiting for monitor to change its state, Have a separate internal queue, as opposed to external queue where initially blocked threads wait
   - *notify()*: notifying process immediately exits the monitor, notifying process swaps roles and goes into the internal queue of the monitor, notifying process keeps control till it completes then one of the notified processes steps in
   - Should check the wait() condition again on wake up

11 Week 11

1. Monitors in Java

   - Monitor incorporated within existing class definitions.
   - Functions declared *synchronized* are to be executed atomically.
   - wait, notify, and notifyAll
   - Use object locks to synchronize arbitrary blocks of code
     synchronized(o){}
     Each object has its own internal queue
     o.wait(), o.notifyAll()
   - wait can be interrupted by InterruptedException, catch it.
   - *private Lock bankLock = new ReentrantLock();*
     bankLock.lock() behaves as P(s)
     bankLock.unlock() behaves as V(s)

2. Thread in Java

   - A thread can be in six states, get via *t.getState()*
     New: Created but not started
     Running: Started and ready to be scheduled
     Blocked: Waiting for a lock
     Waiting: Suspended by wait
     Timed Wait: Within sleep
     Dead: Thread terminates
   - One thread can interrupt another using *interrupt()*, Raises InterruptedException within wait, sleep, no exception raised if thread is running.
   - *interrupted()* checks the interrupt flag and clears it.
   - *isInterrupted()* check interrupt flag.
   - *yield()* gives up active state to another thread.
   - *t.join()* waits for t to terminate

12 Week 12

1. Swing Toolkit

   - *Swing* Toolkit to define high level components, built on top of lower level event handling system called *AWT*.
   - *JButton* is a Swing class for Buttons, Corresponding listener class is *ActionListener*, we implement it, Only one type of event(button push) invokes *actionPerformed()* in listener, Button push is an ActionEvent.
   - *import java.awt.**
     *import java.awt.event.**
     *import java.swing.**
     public class ButtonPanel extends JPanel implements ActionListener {
     private JButton redButton;
     public ButtonPanel() {
     redButton = new JButton("Red");
     redButton.addActionListener(this);
     add(redButton);
     }
     public void actionPerformed(ActionEvent event) {
     Color color = Color.red;
     setBackground(color);
     repaint();
     }}
     public class ButtonFrame extends JFrame implements WindowListener {
     private Container contentPane;
     public ButtonFrame() {
     setTitle("ButtonTest");

```
setSize(300, 200);
// ButtonFrame listens to itself
addWindowListener(this);
// ButtonPanel is added to the contentPane
contentPane = this.getContentPane();
contentPane.add(new ButtonPanel());}
// Six out of the seven methods required for
// implementing WindowListener are stubs
public void windowClosing(WindowEvent e) {
System.exit(0);}
public void windowActivated(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}
}
public class ButtonTest {
public static void main(String[] args) {
EventQueue.invokeLater(() -> {
JFrame frame = new ButtonFrame();
frame.setVisible(true);
});
}}
```

- Can get source by *event.getSource()*