**Assertions**- used for testing for errors during programming stage of computer software development. These errors should not occur in the final user used version of the program. Once an assertion is *thrown* the program terminates along with the assertion’s string statement and the stack trace.

**Exception**- is an error that can be *thrown* by a method.

Ex. Integer.parseInt(“abc”) will throw an exception called IllegalArgumentException (a kind of exception)

To throw one:

**String message = “An SSN length must be 9”;**

**IllegalArgumentException iae; //creates an IllegalArgumentException object called iae**

**Iae = new IllegalArgumentException(message);** //instantiating the object

**Throw iae;**

Or:

**Throw new IllegalArgumentException (“An SSN length must be 9”);**

Complete Example:

**public class SSN{**

**static String ssn;** /\*we're inputing ssn as a string so that leading zeroes dont end

up getting stored as place holders\*/

**static String[] ssnList;**

**static int ssnSize;**

**public static boolean isValidSSN(String s){**

**if (ssn.lenght() != 9){**

**throw new IllegalArgumentException ("An SSN length must be 9");**

**/\*System.out.println("An SSN length must be 9");**

**return (false);\*/**

**}**//length if

**for (int i=0; i<9; i++){**

**if(! Character.isDigit(ssn.charAt(i))){**

**throw new IllegalArgumentException ("SSN must have only digits");**

**//return (false);**

**}**//digit check

**return (true);**

**}**//valid ssn

**public static boolean isValidList(String[] list){**

**if(list == null){**

**/\*throw new IllegalArgumentException(“List can’t be null”);**

**System.out.println("Array is null.");**

**System.exit(0); \*/**

**}**

**if (ssnSize == list.length){**

**System.out.println("Can't store any more SSNs.");**

**System.exit(0);**

**}**

**return true;**

**}**//valid list

**public static String readSSN(){**

**ssn = (JOptionPane.showInputDialog(null,"Enter SSN:"));**

**if (ssn == null)**

**return "000000000";**

**else if (isValidSSN(ssn))**

**return ssn;**

**else** //if it’s not valid and not equal to null then return a null ssn since it wasn’t valid

**return null;**

**}**//read ssn

**public static void storeSSN(String s, String[] list){**

**assert (isValidSSN(s)): "The SSN is not valid";**

**assert (isValidList(list)): "The array is not valid";**

**if(isValidSSN(s) && isValidList(list)){**

**//s is a real ssn and there is room in the real array**

**list[ssnSize++]=s;** /\*your not just passing in any int that will get passed by value and

therefore won't have permanent changes made to it, your passing in the original static int that's

one throughout the program that you initially declared\*/

**}**

**assert (isValidList(list)) : "Resulting list not valid";**

**}**

**public static void printSSNList(String[] list, int size){** /\*we're not changing the size, so

size can be a regular int instead of something static runningthroughout.\*/

**if(!isValidList(list)){**

**System.out.println("Can't print from invalid list.");**

**System.exit(0);**

**for(int i=0; i<size; i++){**

**if(!isValidSSN(list[i]))**

**System.out.println("Invalid SSn: "+list[i]);**

**else**

**System.out.println(list[i]);**

**}**

**public static void main (String[] args){**

**initialize();**

**do{**

**ssn=readSSN();** /\*ssn is checked that it's a real ssn number and not just any string **\*/**

**storeSSN(ssn, ssnList, ssnSize);** /\*need to check inputs again-despite the fact that there

is a real ssn in the program- becuase user may not enter the checked ssn as the input to this

function, also make sure that the array entered exists and has available space for more input\*/

**printSSNList(ssnList, ssnSize);**

**}**

**while(!ssn.equals("000000000");**

**}//main**

**}//SSN class**

(**Assertions and Exceptions**

*Assertions* are used during code development to make sure that the logic is correct and to take care of things which can’t be a possibility- causes the program to end. *Exceptions* are for during the program execution, to inform the user about some error such as data input problem, lost file, parameter error ect.- errors that can happen.

Examples of exceptions: **IllegalArgumentException**, **IndexOutOfBoundsException**, **NullPointerException**

The Runtime System looks for methods that can handle the exception. It looks at the most recently called method and backs up all the way to the main program (exceptions get propagated). If no such method is found then the Runtime System handles the exception and terminates the program.

**Try/Catch Block**- the try block contains a statement which may throw an exception, the catch block (parmeters exception type name) takes care of what should be done if an exception happens. Try to make your catches specific to different types of exceptions. Also have one general exception (last), for exceptions which don’t fit into the specific ones.

Always *make* sure that necessary program info, isn’t written after something which can cause an error exception. If the exception is thrown, the necessary block of coding will be skipped since we jump to the end of the catch statement. The **Finally** keyword executes even if there is a return statement prior to the final code.

Excluding exceptions in the class RuntimeException, the compiler must find a catcher or a propagator for every exception. RuntimeException is an ***unchecked*** exception- are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur. All other exceptions are ***checked*** exceptions- These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Ex. **ssnRead = inFile.readLine();**

**while (ssnRead != null) {**

**try {**

**mySSN = new SSN(ssnRead);**

**mySubscripts.append(Integer.toString(subscript++)+"\n");**

**myTextArea.append(mySSN+"\n");**

**ssnRead = inFile.readLine();** /\*if the lines above cause an exception, this

line will be skipped\*/

**}//try**

**catch (IllegalSSNException issne) {**

**System.out.println(issne.getMessage());**

**}//catch**

**finally {**

**ssnRead = inFile.readLine();**

**}**/\*now it won’t get skipped\*/

**}//while** )

Dynamic Allocation

The size of an array is fixed from the time it gets instantiated. If you ever needed to increase space in the array, you’d have to move everything over to a new array which could take a lot of time and coding. Also, if the array is organized in a certain way, you can’t just add to it flexibly anywhere you want (ex. Sorted array- can’t quickly switch something in the middle, would need to re-sort the whole thing)

**Dynamic Structures** can be a solution to this problem. They only use as much memory as they need. They rely on themselves to determine the order of the data items they contain- they are not stored in contiguous memory. You would just have to make “fat” point to what “cat” was pointing to and make “cat” point to “fat”; No timely loop involved.

**Node** is it’s own class. It contains the actual data it holds (“bat”) and a reference linking it to the next node on the list (“cat”).

Class Hierarchy

Extend a class based on the “is a” relationship: a *Cat* is a *Pet* so class Cat extends Pet

**Protected** modifier means only descendant classes have access to the variable

**Public** modifier grants access to the variable from any class

**Private** modifier grants access to the variable only from instances of the same class

When a class is instantiated the first thing it does is construct its super class

**Super(<optional parameters>)** //calls one of the constructers from the super class

**Abstract Class** = cannot be instantiated

An abstract class is declared abstract if **it contains an abstract method**- is only declared, not defined or if it inherits an abstract method and doesn’t overload it.

If you inherit an abstract class –‘extends blablabla (which is an abstract class)’ - then you must define the abstract methods in order for the new class to no longer be abstract. A class which defines its abstract methods is called a concrete class and can create objects.

Abstract classes are used as something to extend onto a ‘child’ class and ensures that the child class will personalize the abstract method from the abstract class

Ex. a class called animals might have an abstract method called bark, and all children classes of animal will have to define their animals bark if they want to create a child of the animal class

Ex. when you declare a ‘child’ class, the parent classes are automatically constructed and exist, but you can’t instantiate them

You can make an array of objects from the general parent class (ex. CUNYStudents), and fill it with objects from mixes of child classes (ex. QueensUndergraduate or QueensGraduate)

The **instanceOf** operator defines which child of a general parent class is being used

Ex. public static void whatKindOfStudent (CUNYStudent[] studentList, int numFilled) {

String kindOfStudent = null;

for (int i=0; i<numFilled; i++) {

if (studentList[i] **instanceof** QueensUndergraduate)

kindOfStudent = "an undergraduate";

if (studentList[i] **instanceof** QueensGraduate)

kindOfStudent = "a graduate";

Event-Driven Programming

An **Event** is something that happens while the program is running. What the user does.

Ex. the user clicks on the X to close the window, the user chooses an item from a menu

**JFrames** are window objects. They don’t do anything until told to. An event such as a menu choice signals the JFrame to respond.

An **Interface** is a collection of method headings only (not bodies). They are implemented by a Java class. If an interface is implemented, all methods specified in the interface must be provided by that class. An interface, if implemented, guarantees that all methods will be defined.

Ex. public interface X {

public int y (int z);

public void q();

}

public class A implements X {

public int y (int z) { return z+1 )

public void q() { }

}

An **Event Hadler** is the coding of what to do when an event occurs. It is the class that implements an interface called **Action Listener.** It defines the **actionPerformed(ActionEvent)** method titled in the Action Listener interface, with instructions on what to do when an event happens.

EventHandlers are attached to an item and listens in case the event occurs. If the event occurs then the actionPerformed method is automatically executed, as defined by the EventHandler class which defined/implemented the actionPerformed method that was in the ActionListener interface.

The ActionListener interface contains the undefined method actionPerformed(ActionEvent Event). The eventHandler class implements the ActionListener interface by defining the ActionPerformed method. The EventHandler is then attached to an item and listens in case the event occurs and the eventHandler needs to be executed.

Ex. FileMenuHandler fmh = new FileMenuHandler(this); /\*FileMenuHandler is an event handler

attached to this JFrame which implements the ActionListener interface- defines the

actionPerformed method\*/

item.addActionListener( fmh ); /\*attaches the eventHandler fmh to the item and sets it to

listen in case a certain event occurs to this item\*/

When one of the items is clicked, and ActionEvent object is created. The actionPerformed method of the handler that is registered with the event is called.

* Main program starts and instantiates a GUI
* GUI creates its JMenuBar, JMenu and JMenuItems
* GUI registers each JMenuItem with one or more ActionListeners
* User clicks on a JMenuItem
* The actionPerformed method in the handler registered with that JMenuItem is called
* GUI waits for another event to happen

Regex - Match exactly the string inside the regex

[ ] = **Character classes**: match any character inside [ ]

[abc] a, b, or c (simple class)

[^abc] Any character except a, b, or c (negation)

[a-zA-Z] a through z, or A through Z, inclusive (range)

[a-d[m-p]] a through d, or m through p: [a-dm-p] (union)

[a-z&&[def]] d, e, or f (intersection)

[a-z&&[^bc]] a through z, except for b and c: [ad-z]

[a-z&&[^m-p]] a through z, and not m through p: [a-lq-z]

Predefined Character Classes:

. Any character (may or may not match line end)

\d A digit: [0-9]

\D A non-digit: [^0-9]

\s A whitespace character: [ \t\n\x0B\f\r]

\S A non-whitespace character: [^\s]

\w A word character: [a-zA-Z\_0-9]

\W A non-word character: [^\w]

Quantifiers:

X? X, once or not at all

X\* X, zero or more times

X+ X, one or more times

X{n} X, exactly n times

X{n,} X, at least n times

X{n,m} X, at least n but not more than m times

^ = beginning of regex, $ = end

Uses **split** to break up a string of input separated by commas and/or whitespace.

**import java.util.regex.\*;**

**public class Splitter {**

**public static void main(String[] args) {**

String myString = “one,two, three four , five”;

Pattern p = Pattern.compile("[,\\s]+");

String[] result = p.split(myString);

**for (int i=0; i<result.length; i++)**

System.*out.println(result[i]);*

}

}

**import java.util.regex.\*;**

**public static isValidSSN(String ssn) {**

**Pattern p;**

**Matcher m;**

**String *SSN\_PATTERN =* "(\\d{3}) ([\\.-]?) \\d{3}\\2\\d{4}";**

**p= Pattern.compile(SSN\_PATTERN);**

**m = p.matcher(ssn);**

**return matcher.matches();**

**}**

m.find() = Boolean found a match or not

m.group() = the matching piece

m.group(0)/m.group(1) = a section of the matching piece

((?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z])(?=.\*[@#$%]).{6,20})

(?=.\*\d) # must contain one digit from 0-9

(?=.\*[a-z]) # must contain one lowercase character

(?=.\*[A-Z]) # must contain one uppercase character

(?=.\*[@#$%]) # must contain one special symbol "@#$%"

. # match anything else

{6,20} # length at least 6 and maximum 20

**Associative Collection** – when a value is associated with a key that is used to look up the value. Ex. You look up a person’s phone number in a telephone book, using their name. Name = key Phone Number = value you are looking for

Map Interface –uses associative connections

Interface Map<k,v> //k,v = type parameters for the key and the value, don’t have to be the same types

Put(k key, v map) = main method for adding data to a map

Get(k key) = return key value if key is found

Remove(object key) = removes the mapping for a key from this map if it is present

Int size() = returns the number of key-value mappings there are in this map

Values() = return a collection view of the values contained in the map

KeySet() = returns a set view of the keys contained in the map

entrySet() = returns a collection of key/value pairs

**TreeMap <String, String> french = new TreeMap<String, String> ( );** //create a new tree map

**french.put("cat","chat");**

**french.put("water","eau");**

**french.put("moon","lune");**

**String frenchWord = french.get("water");**

**Set set = french.entrySet();** //creates a group of items as one variable

**Iterator i = set.iterator();** //attach the iterator to the set

**Map.Entry <String,String> me;** //me is an entry pair

**while(i.hasNext()) {**

**me = (Map.Entry) i.next();** //cast what the iterator is reading out, into a Map.Entyr

**System.out.print(me.getKey() + ": ");**

**System.out.println(me.getValue());**

**}**

TreeMap <SSN, Integer> treeMap = new TreeMap (new SSNComparator());

A **Comparator** tells the TreeMap how the keys are ordered. It is a class that implements comparator.

**import java.util.Comparator;**

**public class SSNComparator implements Comparator <SSN> {**

**public int compare(SSN num1, SSN num2) {**

**return num1.compareTo(num2);**

**}**

**public Data search (key, root) {**

**if (root == null) return null;**

**if (root.key == key) return root.data;**

**if (key < root.key) return search (key, root.left);**

**if (key > root.key) return search (key, root.right);**

**}**

**Abstract Class**

**public abstract class QueensStudent extends CUNYStudent {**

**private String semesterEntered;**

**private String major;**

**protected float gpa;**

**public QueensStudent () {**

**super ("Queens");**

**}**

**public QueensStudent (String theSemester,**

**String theMajor,**

**float theGPA) {**

**super ("Queens");**

**semesterEntered = theSemester;**

**major = theMajor;**

**gpa = theGPA;**

**}**

**}**

(Java Notes from class:

O(2^n) is how long it would take for the towers of Hannoin problem.

**File Input and Output**

A **File** can be stored on a variety of devices: Hard drive, CD/DVD ect.

File Name = Project1.java

File path: /usr/smith/java/Project1.java

**Charset** = a named mapping between sequences of sixteen-bit Unicode units and sequenced bytes

Constructor for TextFileInput:

**public TextFileInput(String filename){**

**this.filename = filename;**

**try {**

**br = new BufferedReader( new InputStreamReader( new FileInputStream(filename)));**

**}**

**//FileInputStream(String name)- opens a connection to an actual file. File name is the pathname name in the file system. If something goes wrong a FileNotFoundException is thrown. An object is created to represent the file connection called a FileDescriptor. The method read() reads a byte of data from the input stream. The FileInputStream reads ASCII bytes from the file and delivers a stream of 32-bit int values. Throws IOException if error occurs.**

**FileInputStream fs = new FileInputStream(input.txt) //file descriptor representing the file**

**try { connection**

**int myChar = fs.read(); //reads in a byte of data from the input stream**

**}**

**catch (IOException ioe) {**

**}**

**//InputStreamReader- a bridge from byte streams to character streams. It reads bytes and decodes them into characters using specified charset. The InputStreamReader converts the ints into a stream of Unicode characters.**

**//BufferReader seperates the stream of Unicode characters into “lines” readLine() reads a line of text up to some sort of terminator. It then returns a string containing the contents of the line. Throws IOException if error occurs.**

Need to **import java.io.\*** whenever dealing with input/output stuff.

Class File – an abstract representation of file and directory pathnames

**File myFile = new File (“/usr/smith/java/Project1.java”); //path of where the file is**

**Generic** – a general form of something

Instead of an object being locked into a set data type (ex. int) of private variables, or a too general parent class of an object (ex. object e), a generic version of an object assigns a specific type for that case. So each case is associated with one type of variable, but every case can be locked into a different variable.

Ex. **public class SSNListNode {**

**SSN data;**

**SSNListNode next;**

**public SSNListNode(SSN mySSN) {**

**data=mySSN;**

**next=null;**

**}}**

//the list can only handle type SSN

Vrs.

**public class ListNode {**

**Object data;**

**ListNode next;**

**public ListNode(Object myObject) {**

**data=myObject;**

**next=null;**

**}}**

//this is too general, runt time exceptions can occur because of this

So…

**public class ListNode <E> {**

**E data;**

**ListNode next;**

**public ListNode(E myData) {**

**data=myData;**

**next=null;**

**}**

**public ListNode() {**

**data=null;**

**next=null;**

**} }**

//Now every instance of ListNode can lock itself into any type

**public class StringListMain {**

**public static void main(String[] args) {**

**String[] strings = {"bat","cat","fat","hat","mat","sat"};**

**LinkedList<String> stringList = new LinkedList<String>();**

**for (int i=0; i<strings.length; i++) {**

**stringList.append(strings[i]);**

**}**

**stringList.print();**

**} }**

Recursive Functions

For recursive functions you first need a base case. Then you can continue calling on yourself until the base case eventually stops the never ending loop.

**private static int factorial (int n) {**

**if (n==0)**

**return 1;**

**else**

**return n\*factorial(n-1);**

**} }**

**Fibonacci(n) = Fibonacci(n-1)+Fibonacci(n-2)**

**private static int fibonacci (int n) {**

**if (n==0)**

**return 0;**

**if (n==1)**

**return 1;**

**return fibonacci(n-1)+fibonacci(n-2);**

**}**

Euclidian Algorithm:

Given two natural numbers *a* and *b*: check if *b* is zero; if yes, *a* is the gcd. If not, repeat the process using (respectively) *b*, and the remainder after dividing *a* by *b*.

**private static int gcd (int n, int d) {**

**if (d == 0) return n;**

**else return gcd(d, n % d);**

**}**

Binomial Expansion

private static int bc (int n, int r) {

if (n==0 || r==0 ||n==r) {

return 1;

}

else

return bc(n-1,r)+bc(n-1,r-1);

}

Towers of Hanoi

private static void moveRings(int numberOfRings, String fromTower, String toTower, String

tempTower) {

if (numberOfRings == 0) return;

moveRings(numberOfRings-1, fromTower, tempTower, toTower);

System.out.println("Move a ring from tower "+ fromTower+" to tower "+toTower);

moveRings(numberOfRings-1, tempTower, toTower, fromTower);}