Data Types

Primitive Types
- int, double, float, boolean, char (lowercase)

Not Primitive
- String

Reference

Type Conversion

Implicit Conversion

- No information loss, no possible information loss
- Happen automatically when the program runs

```
float money;
int cash= 25;
money = cash;                    // Converts from int to float

System.out.println (money);


double num1 = 2.0;

int num2 = 6;

System.out.println (num2/num1);
```

For reference, there exist a list of rules that we can use to know the type of the operand when operations are used with +, -, *, /, %, <, <=, >, >=, == and != .
To do in the following order:

1. If one of the operands is a double, the other is converted to a double
2. If one of the operands is a float, the other is converted to a float
3. If one of the operands is a long, the other is converted to a long
4. In any other case, the two operands are converted to int

Conversion table (no info loss)

From	To
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Explicit Conversion / Cast

- It is possible to lose data, need “cast” operator
- Eg converting from float to int: decimal value is lost

```
double d = 16.987;
float f = (float) d;              // Converts from double to float without any loss of data
int i = (int) f;              // Converts from float to int with loss of data
long l = (long) i;              // Converts from int to long without any loss of data
System.out.println (d);
System.out.println (f);
System.out.println (i);

System.out.println (l);


OUTPUT

16.987

16.987

16

16
```

ERROR MESSAGE	DESCRIPTION
variable VARIABLE_NAME might not have been initialized	You tried to use the variable VARIABLE_NAME without initializing it. Careful, it might have been a typo!
';' expected	A semi-colon was expected at the end of a line.
cannot assign a value to a final variable VARIABLE_NAME	You are trying to assign a value to a constant VARIABLE_NAME that was already initialized.
incompatible types found: TYPE1 required: TYPE2	You are trying to assign a value of TYPE1 to a variable of type TYPE2
cannot find symbol symbol: WORD location: LINE	The compiler finds a word that it does not understand. Make sure that it is not a typo.
illegal start of expression	The compiler finds an element that should not be there
not a statement	The line of code is not valid
reached end of file while parsing } ^	There is probably an error in your code block. Check your brackets {}

Equals

Since in Java, every class inherits from the class **Object**, every class inherits the method **boolean equals(Object obj)**. This is true for the predefined classes, such as **String** and **Integer**, but also for any class that you will define.

The method **equals** is used to compare **references** variables. Consider two references, **a** and **b**, and the designated objects.

- If you need to compare the **identity** of two objects (do **a** and **b** designate the same object?), use "**==**" (i.e. **a == b**);
- If you need to compare the **content** of two objects (is the **content** of the two objects the same?), use the method "**equals**" (i.e. **a.equals(b)**).

idek

SELECTION SORT

```
public static void sort(int[] xs) {
        int i, j, argMin, tmp;
        for (i = 0; i < xs.length - 1; i++) {
                argMin = i;
                for (j = i + 1; j < xs.length; j++) {
                        if (xs[j] < xs[argMin]) {
                                argMin = j;
                        }
                }
                tmp = xs[argMin];
                xs[argMin] = xs[i];
                xs[i] = tmp;
        }
}
```

BINARY SEARCH

```
/* COPIED FROM BINARY SEARCH FUNCTION GIVEN BY VIDA IN ITI1120 */
private int intBinarySearch(int[] array, int x) {
        int b = 0;
        int e = array.length - 1;

        while (b <= e) {
                int mid = Integer.valueOf((b + e) / 2);

                if (x < array[mid]) {
                        e = mid - 1;
```

```
                }
                else if (x == array[mid]) {
                        return mid;
                }
                else {
                        b = mid + 1;
                }
        }
        return -1;
}
```

OOP

Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

declare class variables/attributes as private
provide public get and set methods to access and update the value of a private variable

Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

Inheritance

- A class (child) is said to be **derived** from another **if it inherits** from that class (parent).
- The key word **extends** is used to signifiy that a class inherits from the given class.
- When a class is derived from another, this class can access all the **methods** or **instance variables** that are declared **public** or **protected**.
- Inheritance makes use of **polymorphism**. It is possible to call a method of a parent class using the key word **super**.
- If you don't want other classes to inherit from a class, use the final keyword
    - final class Vehicle {}

“Super” Classes

- The keyword "super" is used to call the constructor, methods, and properties of a parent class. If your method overrides one of its parent class's methods, you can invoke the overridden method through the use of the keyword super.
- Here are some examples of how the keyword super is used:
    - super() calls the parent class constructor with no arguments.
    - It can also be used with arguments. i.e. super(argument1), and it will call the constructor that accepts 1 parameter of the type of argument1 (if it exists).
    - Also it can be used to call methods from the parent i.e. super.aMethod()
    - and variables from the parent i.e. super.aVariable
    - Important note: the constructor call super() must be the first statement in a constructor.

Polymorphism

- The word **polymorphism** derives from the words *poly* meaning "many", and *morphism* meaning "behaviour". In Java, **polymorphism** means that an **instance of a class** (an object) can be used as if it were of different types, thus having **many behaviours**. Here, a type means a class is **derived** from another **class** or an **interface.**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

-

Interfaces

- interface: implements the relationship "can be seen as" (as opposed to "is a", that class inheritance implements)
- not a child
- no concrete methods
- collection of abstract classes
- can implement from as many interfaces as you want
- ressembles an abstract class definition
- consists of the keyword "interface", instead of "class", followed by the name of the interface
-
- Used to achieve polymorphism
- Interfaces cannot implement methods, can only contain the signature
- General best practice to only declare abstract methods
    - Dont declare constants in interface
- All methods in interface are public and abstract, even if you leave out the keywords
- The methods however need to be implemented by some class before you can execute them. All the methods in an interface are public and abstract, even if you leave out the "public" and "abstract" keywords.
- To use an interface you must implement that interface in some Java class.
- Once a class implements an interface you safe the reference of an instance of that class in a reference variable whose type is the name of the interface. Here is an example of a main method:

- It is possible for an interface to inherit from another interface, just like classes can inherit from other classes. You specify inheritance using the "extends" keyword to declare your interface derives from another. Unlike classes, interfaces can inherit from multiple superinterfaces. You specify that by listing the names of all parent interfaces to inherit from, separated by comma.

In short:

- An interface contains only abstract methods. An abstract method does not have an implementation.
- The keyword implements is used to express the relationship between the class and the interface.
- When a class implements an interface, this is a sort of contract that means that the class will have to implement all the methods of the interface.
- You cannot create instances of an interface by itself. You must create an instance of a class that implements the interface, and reference this instance using a reference variable whose type is the name of the interface
- A class can implement ("implements") several interfaces
- An interface extend ("extends") several interfaces

INTERFACES vs ABSTRACT CLASSES
- both are ways to define a data type with an abstract contract (i.e. without implementation)
Interface
- problem needs multiple-inheritance
Abstract
- mixing concrete and abstract classes

Property	Concrete	Abstract	Interface
instances can be created	yes	no	no
instance variables and methods	yes	yes	no
constants	yes	yes	yes
can declare abstract methods	no	yes	yes

Abstraction

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).
- The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Abstract class vs Interface

Abstraction: Hiding the internal implementation of the feature and only showing the functionality to the users. i.e. what it works (showing), how it works (hiding). Both abstract class and interface are used for abstraction.

- Type of methods: Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
- Final Variables: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
- Type of variables: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
- Implementation: Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
- Inheritance vs Abstraction: A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
- Multiple implementation: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- Accessibility of Data Members: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

When to use what?

Consider using abstract classes if any of these statements apply to your situation:

- In java application, there are some related classes that need to share some lines of code then you can put these lines of code within abstract class and this abstract class should be extended by all these related classes.
- You can define non-static or non-final field(s) in abstract class, so that via a method you can access and modify the state of Object to which they belong.

- You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using interfaces if any of these statements apply to your situation:

- It is total abstraction, All methods declared within an interface must be implemented by the class(es) that implements this interface.
- A class can implement more than one interface. It is called multiple inheritance.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

Idk, not OOP

DATA TYPES

- a data type is characterized by:
    - a set of values
    - a set of operations
    - a data representation
- these characteristics are necessary for the compiler to verify the validity of a program---which operations
    are valid for a given data
- these characteristics are also necessary for the compiler to be able to create a representation for the data in memory;
    how much memory to allocate for example

ABSTRACT DATA TYPE (ADT)

- an ADT is characterized by:
    - a set of values
    - a set of operations
- i.e. the data representation is not part of specification of an ADT
- a concrete data type must have representation, but the ADT makes a distinction between "how it is used" and
    "how it is implemented", which is private

INSTANCEOF

s instanceof T
- returns false: s is null, s is not compatible with type T
- s is compatible with type T if:
    - T is a class and s is and instance of T, or an instance of a subclass of T
    - T is an interface, and s is an instance of a class that implements T

INSTANCEOF & TYPECAST

```
Shape s;
if (s instanceof Circle) {
        Circle c;
        c = (Circle)s;
        double radius = c.getRadius();
}
long l;
if (Integer.MIN_VALUE <= 1) && (l <= Integer.MAX_VALUE)) {
        int i;
        i = (int)l;
}
```

- always use instanceof before type casting
- this way, we ensure that there is no loss of information (long and int example)

Call-By-Value

Call-by-value

- There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Call-by-reference

- While Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.
- In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed.
- In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification

done to the parameter passed are persistent and changes are reflected in the caller's scope.

-

STACKS

- abstract data types (structures) similar to physical stacks (eg stack of books)
1) only top element is accessible
2) top element must be removed in order to access remaining elements
- eg: stack of plates

LIFO: last-in, first-out
- stacks are called LIFO data structures
- stack: linear data structure that is always accessed from the same extremity, one element
at a time. that element is called the top of the stack

```
s = new StackImp()
s.push("alpha")
s.push("beta")

stack
"beta"
"alpha"

o = s.pop()

stack
"alpha"
```

STACK APPLICATIONS

- in compilers
- implementing backtracking algroithms
- memory management
- components of memory:
    - hip
    - stack
- "undo" and "redo" operations

STACK BASIC OPERATIONS

- push: add element to top
- pop: remove and return top element
- empty: tests to see if stack is empty

STACK ADT (they must all be same type)

```
public interface Stack {
        public abstract boolean isEmpty()
        public abstract Object push(Object o)
        public abstract Object pop()
        public abstract Object peek() -> tells u which element is on top
}
public interface Stack<E>{
        public abstract boolean isEmpty()
        public abstract E push(E elem)
        public abstract E pop()
        public abstract E peek() -> tells u which element is on top
}
```

- stacks are useful for reversing things

IMPLEMENTATIONS

- 2 popular ways
    - array-based: can only add elements up to the array size
    - linked-nodes

why bother with stack implementation if ur going to make an array anyway?
- we use stack to guarantee LIFO behaviour
- need counter to keep track of stack size (not same as actual array length)
- better to have top of stack at biggest index and bottom of stack at index 0

COMMON STACK IMPLEMENTATIONS

- stacke-based algorithms are used for syntactical analysis (parsing)4
    -eg: 1 + 2 * 3 - 4
- compilers use similar alforithsm to check the syntax of ur programs andgenerate machine isntructins (executable)

- to verify that parantheses are balanced: '([])' is ok but not '([)]'

- steps of the analysis of a source program by a compiler
    1) lexical analysis (scanning):
    - source code is read left to right and characters are regrouped into tokens,
        -tokens: successvie characters that constitute numbers of identifiers
    - lexical analyser removes spaces from intput
    - eg: 10 + 2 + 300 -> [10, +, 2, +, 300]
    2) syntactical analyses (parsing)
    - regrouping the tokens into grammatical units

LR SCAN EXAMPLE

```
public static int execute(String expression) {
    Token op = null; int l = 0, r = 0;

    Reader reader = new Reader(expression);
    l = reader.nextToken().iValue();

    while (reader.hasMoreTokens()) {
        op = reader.nextToken();
        r = reader.nextToken().iValue();
        l = eval(op, l, r);
    }
    return l;
}
```

EVALUATING AN ARITHMATIC EXPRESSION: LR SCAN

left-to-right algorithm:
- decalre L, R, OP
- read L
- while not end of expression
    do:
    - read OP
    - read R
    - evaluate L OP R
    - store result in L
- at end of loop, the result can be found in L

PROBLEMS:

- reads left to right (no BEDMAS)
- assumes user enters correct expression (operand operator operand operator .... operand)

NOTATIONS

3 ways to represent the following expression: L OP R
- infix: usual notation, operator sandwiched between its operands
    L OP R
- postfix:
    L R OP
    7 - (3 - 2) = 7 3 2 - -
    (7 - 3) - 2 = 7 3 - 2-
    9 / (2 * 4 - 5) = 9 2 4 * 5 - /
    2 + (3 * 4) = 2 3 4 * +
    (2 + 3) * 4 = 2 3 + 4 *
- prefix:
    OP L R
    (- 7 (* 3 2))

EVALUATING A POSTFIX EXPRESSION

- the algorithm requires a stack (Numbers) , a variable that contains the last element that was read (X) and two more variables, L and R, whose purpose is the same as before.

Number = [

While not end-of-expression
do:
    Read X
    if X isNumber, PUSH X onto Numbers
    If X isOperator,
        R = POP Numbers (right before left?!)
        L = POP Numbers
        Evaluate L X R; PUSH result onto Numbers
To obtain the final result: POP Numbers

POSTFIX -> INFIX

Number = [

X =
L =
R =

While not end-of-expression
do:
    Read X
    if X isNumber, PUSH X onto Numbers
    If X isOperator,
        R = POP Numbers
        L = POP Numbers
        Concatenate (L X R); PUSH result onto Numbers
To obtain the final result: POP Numbers

LINKED STRUCTURES

- data structure that always uses the exact amount of memory required by the apllication
- grows one element at a time without copying any elements
- we need to create "containers" to hold one element each
- pieces of memory instead of block
- data structure that always uses the exact amount of memory required by the apllication
- grows one element at a time without copying any elements
- we need to create "containers" to hold one element each
- class with 2 instance variables
    1) holds reference to some object (eg a String)
    2) holds reference to another similar object

```
public class Elem {
    public Object value;
    public Elem next;
    // the type of the reference next is the name of the class we are currently defining

    Elem(Object value, Elem next) {
        this.value = value;
        this.next = next;
    }
```

}

ARRAY STACK IMPLEMENTATION PROBLEMS

- can access items very fast
- growing the array is inefficient
- big block of memory

IMPLEMENTATION (Elem)

```
public class Elem {
        public Object value;
        public Elem next;
        // the type of the reference next is the name of the class we are currently
defining

        Elem(Object value, Elem next) {
                this.value = value;
                this.next = next;
        }
}
```

WHAT DOES IT DO?

- node: element of linked list

Elem p;
- reference of type Elem
- not referencing anything

p = new Elem()
- p is pointing to this
- new instance of Elem
- p.value and p.next dont reference anything

Elem 0

p.value = new Time(13,0,0)
- changeing content of instance variable value

p.next = new Elem()
- p.next references a new Elem

Elem 1
p.next.value = new Time(14, 30, 0)
- changing value of next node
p.next.next = new Elem();
- now the linkedlist has 3 nodes
Elem 2
p.next.next.value = new Time(16, 0, 0)
etc...
What Does This Do?
p.next.next = p
- will compile bc p.next is an Elem and p.next.next is an Elem
- loop
- p points to Elem 0
- p.next.next is Elem 1
- Elem 1 points to Elem 0
- Elem 1 no longer points to Elem 2
- this creates a circula data structure
- Elem 2 is not accessbile anymore
- Elem 2 will be recycled by the garbage collector; gc()

TYPICAL CONSTRUCTOR FOR CLASS ELEM

```
class Elem {
        Object value;
        Elem next;

        Elem (Object value, Elem next) {
                this.value = value;
                this.next = next;
        }
}
```

TYPICAL USAGE

p = new Elem("A", null)
- new node, with string "A"
- next element is null
q = new Elem("B", p)
- q points at p

- q.next = p

PROBLEMS

- not intuitive
- first node constructed (p) is the last node in the LinkedList
- easy to construct, harder to use

COMMON ERROR

Elem p = null, q = null;
p = new Elem("A", null);
q.next = p;
- syntactically correct, produces this error at runtime:
    Exception in thread "main" java.lang.NullPointerException at T01.main(T01.java:8)

COMPARING IMPLEMENTATIONS (for 3, 2, 6)

s -> top -> 3 -> 2 -> 6 -> null
- like Array Stack Implementation
- this one is better
- all the action is happening at the top
    - eg: removing, adding
- removing a node: make current top point to next top
    - make s point to 2 instead of 3
- adding a node: make new Elem, new Elem points to current top
    - new top is this new Elem
- this is better than other implementation bc we dont have to go through the whole structure
- number of steps is constant, which is what we want
    - number of steps does not depend on stack size (unlike typical array)

s -> bottom -> 6 -> 2 -> 3 -> null
- like typical array

STACK: LINKED IMPLEMENTATION

```
public class LinkedStack implements Stack {
	private Elem top;

	public LinkedStack() {
		top = null;
	}

	public boolean isEmpty() {
		return top == null;
	}

	public Object pop() {
		// precondition: the LinkedStack is not empty
		Object val = top.value;
		top = top.next;
		return val;
	}

	public Object peek() {
		// precondition: the LinkedStack is not empty
		return.value;
	}

	public Object push(Object element) {
		top = new Elem(element, top);
		return element;
	}
}


// use generics
public class Elem<F> {
	public F value;
	public Elem<F> next;

	public Elem(F value, Elem<F> next) {
		this.value = value;
		this.next = next
	}
}
```

```
public class LinkedStack<E> implements Stacl<E> {
	private Eleme<E> top;
}

// problem: want stack to be public to LinkedStack but private to everyone else
// solution: nested class

public class LinkedStack<E> implements Stack<E> {
	public static class Elem<F> {
		private F value;
		private Elem<F> next;

		public Elem (F value, Elem<F> next) {
			this.value = value;
			this.next = next;
		}
	}
}

public class LinkedStack {
	public String toString() {
		String result;
		result = "top -> [";

		// implementation 1
		while (top != null) {
			result += top.value;
			top = top.next;
		}
		/
		prints all the elements, but also empties the stack
		/

		// implementation 2
		Elem<E> p;
		p = top;
		while (p != null) {
			result += p.value;
			if (p != null) {
```

```
                result += ", ";
            }
            p = p.next;
        }

        result += "] <- bottom";
        return result;
    }
}
```

problem with concatenating string in toString()

- inefficient
- strings are immutable
- concatenating makes a new string

solution: StringBuffer, StringBuilder
- buffer: can modify object
    - stores the strings in the .append() param in a linkedlist, or queue, or stack or smt
    - only concatenates at the end when .toString() is called
- builder: cant modify

```
public String toString() {
    StringBuffer result = new StringBuffer("[");

    if (!isEmpty()) {
        Elem<D> p = front;
        result.append(p.value);
        while (p.next != null) {
            p = p.next;
            result.append(", " + p.value);
        }

    }
    result.append("]");
    return result.toString();
}
```

ERROR PROCESSING (Exceptions)

- errors can occur at compile-time or runtime
- syntax errors are detected at compile time
- since java is a strongly typed language, the compiler will also ensure that
    each expression is valid, id only valid operations for this data type are performed,
    avoiding errors that would otherwise occur during the execution of the program.
- although type checking is an efficient way of detecting errors asap, there are certain
    verifications that can only be made at execution time
    eg; checking that a stack contains an element before removing it
- in Java, runtime errors are handled by a mechanism called Exceptions
- runtime crash bad

SOURCE OF ERRORS

- logic of program
- external events: running out of memory, write error, etc
- eg: the method pop of the ArrayStack implementation could throw an
    IndexOutOfBoundsException. The source of error likely occurred because the caller did not check if the stack was empty before calling pop.

HANDLING ERRORS

- Exceptions are objects

Object
  |
Throwable
  |
Exception
  |
.....

EXCEPTION IS A CLASS

- the class Throwable() declares the methods String getMessage() and void printStackTrace()

```
Exception e; // declaring a reference of type Exception
e = new Exception("Houston, we have a problem!"); // creating an object of the class Exception
```

SIGNALING AN ERROR

- ex: ArrayStack and pop()
- the empty stack is an illegal case (illegal state)
- in Java, the way to handle illegal cases is to "throw" an exception

```
if (top == -1) {
        throw new IllegalStateException("Empty stack");
        /
        telling the runtime "we are not in normal mode, go to exception mode"
        need to handle exception before going to back to normal
        /
}
```

- throw statement:
    - throw expression
- where expression is a reference to an object of the class Throwable, or one of its subclasses

```
IllegalStateException e;
e = new IllegalStateException("Empty stack");
if (top == -1) {
        throw e;
}
```

- its the throw that changes the flow of control

```
public class Test {
        public static void main(String[] args) {
                System.out.println("-1");
                throw new RuntimeException("an Exception");
                System.out.println("-2");
        }
        /
        Why doesnt it compile?
```

```
	unreachable statement, cant reach System.out.println("-2"); cuz of the exception
	/
	public static void main(String[] args) {
		System.out.println("-1");
		if (true) {
			throw new RuntimeException("an Exception");
		}
		System.out.println("-2");
	}
	/
	wont print -2, immediately terminates method
	/
}
```

HANDLING EXCEPTIONS

- goes back to normal mode from exception mode when exception is caught
- conitnues from where the exception is caught, NOT from where the exception is thrown

```
try {
	statements;
}
catch (exception_type1 id1) {
	statements;
}
catch (exception_type2 id2) {
	statements;
}
finally {
	statements;
}
```

- if no exception occurs only the statements of the try and finally blocks are executed

```
int DEFAULT_VALUE = 1;
```

```
int value;
try {
        value = Integer.parseInt("Team A");
}
catch (NumberFormatException e) {
        value = DEFAULT_VALUE
}

catch (Exception e) {
        value = DEFAULT_VALUE
}
/
this block catches any Exception type
howeverm the catch statement is not designed to handle any Exception
/
```

CHECKED AND UNCHECKED EXCEPTIONS

- there are "checked" and "unchecked" exceptions. All the exceptions that are subclasses of Throwable are "unchecked" (default). Except those that are subclasses of Exception, those are "checked". Except those that are from a subclass of "RuntimeException", which are "unchecked"
- a method that throws a "checked exception" must declare or handle the exception

```
class Test {
        public static void main(String[] args) throws Exception {
                System.out.println("-1");
                if (true) {
                        throw new Exception("an Exception");
                }
                System.out.println("-2");
        }
}
```

QUEUES

ARRAY IMPLEMENTATION: CIRCULAR ARRAY

```
public class CircularQueue<E> implements Queue<E> {
	private int front, rear;
	private E[] q;
	private int size;
	private int capacity;

	@SuppressWarnings("unchecked")
	public CircularQueue(int capacity) {
		this.capacity = capacity;
		front = rear = 0;
		size = 0;
		q = (E[]) new Object[capacity];
	}

	public CircularQueue() {
		this(1000);
	}

	public boolean isEmpty() {
		return size == 0;
	}

	public boolean isFull() {
		return size == capacity;
	}

	publid void enqueue ( E o) {
		if ( o == null ) {
			throw new NullPointerException("no null ref in  the queue");
		}
		if (isFull()) {
			throw new IllegalStateException("full queue");
		}

		if (isEmpty()) {
```

```
                rear = front = 0;
            }
            else {
                /
                if (rear == capcity) {
                    rear = 0;
                }
                else {
                    rear++;
                }
                /
                // can be replaced by
                rear = (rear + 1) % capacity;
            }

            q[rear] = o;
            size++;
        }

        public E dequeue() {
            if (isEmpty()) {
                throw IllegalStateException("empty queue");
            }
            E result = q[front];
            q[front] = null;
            front = (front + 1) % capacity;
            size--;
            return result;
        }
}

public String toString() {
        StringBuffer res = new StringBuffer("<");
        if (!isEmpty()) {
            res.append(q[front]);
            for (i = 1; i < size; i++) {
                res.append(", " + q[(front + i) % capacity]);
            }
        }
        res.append("<");
```

```
        return res.toString();
}

public static void newTest() {
        Queue<String> queue = new CircularQueue<String>();

        queue.enqueue("");

        int i = 0;
        while (i++ < 255) {
                String s = queue.dequeue();
                System.out.println(s);
                queue.enqueue(s + "0");
                queue.enquee(s + "1");
        }

        // PRINTS
        /
        0
        1
        00
        01
        11
        000
        010
        011
        100
        101
        110
        111
        0000
        ...
        // every possible combo of 0 and 1
        /
}
```

LABYRINTH

```
public class Solver {

	public static String solve(Labyrinth labyrinth) {
		Queue<String> queue = new LinkedQueue<String>();

		queue.enqueue("");

		while(!queue.isEmpty()) {
			String s = queue.dequeue();

			if (labyrinth.checkPath(s)) {
				if (labyrinth.reachesGoal(s)) {
					return s;
				}
				else {
					queue.enqueue(s + "U");
					queue.enqueue(s + "D");
					queue.enqueue(s + "L");
					queue.enqueue(s + "R");
				}
			}
		}

		return null; // no solution
	}

	public static String solveStack(Labyrinth labyrinth) {
		Stack<String> queue = new LinkedStack<String>();

		queue.push("");

		while(!queue.isEmpty()) {
			String s = queue.pop();

			if (labyrinth.checkPath(s)) {
				if (labyrinth.reachesGoal(s)) {
					return s;
				}
				else {
					queue.push(s + "U");
```

```
                                        queue.push(s + "D");
                                        queue.push(s + "L");
                                        queue.push(s + "R");
                                }
                        }
                }

                return null; // no solution
        }

        // queue finds shorter path than stack, optimal
        // stack finds solution faster than queue
}
```

- breadth first search (leveled search) is optimal, depth search first is not

LISTS

Singly Linked

```
public class SinglyLinkedList<E> implements List<E> {

        private static class Node<T> {
                private T value;
                private Node<T> next;
                private Node(T value, Node<T> next) {
                        this.value = value;
                        this.next = next;
                }
        }

        private Node<E> head;

        public SinglyLinkedList() {
                head = null;
        }

        public int size() {
                int size = 0;
```

```
		Node<E> p;
		p = head;
		while (p != null) {
			p = p.next;
			size++;
		}
		return size;
}

public boolean isEmpty() {
		return head == null;
}

public void addFirst(E o) {
		if(o == null) {
			throw new NullPointerException("cant add null reference");
		}

		Node<E> newNode;
		newNode = new Node<E>(o, null);

		if (isEmpty()) {
			head = newNode;
		}
		else {
			newNode.next = head;
			head = newNode;
		}

		// simpler implementation
		Node<E> newNode;
		newNode = new Node<E>(o, head);
		head = newNode;

		// simpler implementation
		head = new Node<E>(o, head);
}

public void add (E o) {
		if(o == null) {
```

```
                throw new NullPointerException("cant add null reference");
        }

        if (isEmpty()) {
                addFirst(o);
        }
        else {
                Node<E> p;
                p = head;

                while ( p.next != null) {
                        p = p.next;
                }
                p.next = new Node<E>(o, null);
        }
}

public void add (int pos, E o) {
        if(o == null) {
                throw new NullPointerException("cant add null reference");
        }

        if ((pos < 0) || (pos > size())) {
                throw new IllegalArgumentException();
                // OR
                throw new IndexOutOfBoundsException(Integer.toString(pos));
                // both are fine
        }

        if (pos == 0) {
                addFirst(o);
        }
        else {
                Node<E> p;
                p = head;

                for (int i = 0; i < pos - 1; i++) {
                        p = p.next;
                }
```

```
                p.next = new Node<E>(o, p.next);
            }
        }
}
```

Singly Linked Lists - with size variable

```
public class SinglyLinkedList<E> implements List<E> {

    private Node<E> head, tail;
    private int size;

    public SinglyLinkedList() {
        head = tail = null;
        size = 0;
    }

    public void add(E o) {

        if ( o == null) {
            throw new NullPointerException();
        }

        Node<E> newNode = new Node<E>(o, null);
        if(head == null) {
            head = tail = newNode;
        }
        else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
    }
}
```

Recursion

```
public static long fact(int i) {
    // base case; can just give solution
```

```
        if (i <= 1) {
                return i;
        }
        return i*fact(i-1)
}

public static int sum(int[] array, int index) {
        if ((array == null) || (index < 0) || (index >= array.length)) {
                throw new IllegalArgumentException();
        }

        if (index == array.length - 1) {
                return array[index];
        }
        else {
                return array[index] + sum(array, index+1);
        }

}



private int size(Node<E> p) {
        if (p == null) {
                return 0;
        }
        else {
                return 1 + size(p.next);
        }
}
public int size() {
        return size(head);
}



public E findMax() {
        if (head == null) {
                throw new IllegalStateException("list cannot be null");
        }
}
```

```
private E findMax(Node<E> p) {
	if (p.next == null) {
		return p.value;
	}
	else {
		E answer = findMax(p.next);
		if(p.value.compareTo(answer) > 0) {
			return p.value;
		}
		else {
			return answer;
		}
	}
}


public E get(int pos) {
	if (pos < 0) {
		throw new IndexOutOfBoundsException();
	}
}

private E get (int pos, Node<E> p) {
	if (p == null) {
		throw new IndexOutOfBoundsException();
	}

	if (pos == 0) {
		returng p.value;
	}

	return get(pos-1, p.next);
}


// index of the leftmost instance of obj
public int indexOf(E obj) {
	if (obj == null) {
		throw new NullPointerException("obj shouldnt be null");
	}
```

```
        return indexOf(obj, head);
}

private int indexOf(E obj, Node<E> p) {
        if (p == null) {
                return -1; // cant find obj
        }
        if(p.value.equals(obj)) {
                return 0;
        }

        int result = indexOf(obj, p.next);
        if (result == -1) {
                return -1;
        }
        else {
                return result + 1 ;
        }

}

// rightmost instance of obj
public int indexOfLast(E obj) {

}

private int IndexOfLast(E obj, Node<E> p) {
        if (p==null) {
                return -1;
        }

        int result = indexOfLast(obj, p.next)

        if (result == -1) {
                if (p.value.equals(obj)) {
                        return 0;
                }
                else {
                        return -1;
                }
```

```
        }
        else {
                return result + 1;
        }
}
```