MISC

```
<T extends Comparable & Collection> // multiple type bounds

Set<Integer> set = new Set<>();
Integer[] c = set.toArray(new Integer[0]);
```

RECORDS

```
public record Person (String name, String address) {}
```

GENERICS STREAM TOMFOOLERY

```
List<? extends Media> mediaList;
public <S extends T> List<S>
  filterBySubtype(Class<S> subtype) {
    return mediaList.stream()
            .filter(subtype::isInstance)
            .map(subtype::cast)
            .collect(Collectors.toList());
}
```

GENERICS

```
// class
class Box<T> { }
// variable
Box<String> b = new Box<String>();
// method
public <T> void doStuff(T value) { }
```

Example 1:

```
class OrderedPair<T, U> {
  T first;
  U second;
  public OrderedPair(T t, U u) {
    this.first = f;
    this.second = s;
  }
  public T getFirst() { return first; }
  public U getSecond() { return second; }
  @Override
  public int hashCode() {
    return Objects.hash(first.hashCode(),
                        second.hashCode());
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null ||
        obj.getClass() != getClass()) return false;
    @SuppressWarnings("unchecked")
    var other = (OrderedPair<T, U>)obj;
    return Objects.equals(first, other.first) &&
           Objects.equals(second, other.second);
  }
}

public class Employee
    implements Comparable<Employee> {
  // ...
  @Override
  public int compareTo(Employee other) {
    return Double.compare(salary, other.salary);
  }
}
```

Example 2:

```
class Graphic {
  public void draw() { }
}
class Rectangle extends Graphic { }
class Circle extends Graphic { }
class GraphicStack<T extends Graphic>
    extends Stack<T> {
  public void drawAll() {
    for (T item : this) item.draw();
  }
}
```

JAVA BEING STUPID

```
T obj = new T();          // error
obj instanceof T;         // error
T heh = (T) "asdf";       // no typechecking
var s = Stack<int>();     // no primitive types
private static T x;       // static no workey
Node<Number> s = new Node<Integer>; // error
```

```
class IThrowAnErrorBecauseWhyNot {
  void m(List<String> list) { }
  void m(List<Integer> list) { }
} // error because of type erasure & identifiability
static <T extends Comparable<T>> T m(T x, T y, T z) {
  return x > y ? z : x;
}
Integer n = m(1, 3.141, 0);          // error
Number n = m(1, 3.141, 0);           // ok
```

COMPARABLE

```
interface Comparable<T> { int compareTo(T o); }
static <T extends Comparable> T doStuff(T a, T b) {
  // compareTo is possible with all types
  return a.compareTo("b") > 0 ? a : b;
}
static <T extends Comparable<T>> T doStuff(T a,T b) {
  // compareTo is possible only with T
  return a.compareTo(b) > 0 ? a : b;
}
```

BYTECODE

JVM Architecture:

- Operand Stack (LIFO op vals)
- Local variables (op results)
- Constant pool (runtime refs)

TYPE ERASURE

- Introduced because of “bAcKwArDs CoMpAtAbIlItY”
- No type information at runtime (Non-Reifiable Type)

Example 3:

```
class MyStack<T> {
  Entry<T> top;
  int push(T value) { }
}
class MyStack { // becomes this when compiling
  Entry top;
  int push(Object value) { }
}
```

Example 4: With bounds

```
class MyStack<T extends Comparable<T>> {
  Entry<T> top;
  int push(T value) { }
}
class MyStack { // becomes this when compiling
  Entry top;
  int push(Comparable value) { }
}
```

Example 5: Casts

```
MyStack<String> s = new MyStack<String>();
s.push("Eh");
String x = s.pop();
// becomes this when compiling
MyStack s = new MyStack();
s.push("Eh");
String x = (String) s.pop();
```

GENERIC VARIANCE

```
void printGs(List<Graphic> gs) { }
List<Graphic> gs1 = new ArrayList<>();
printGs(gs1);   // ok
List<Circle> gs2 = new ArrayList<>();
printGs(gs2);   // error
/* solution */
void printGs(List<? extends Graphic> gs) { }
```

	Type	*Compatible Type-Args*	*R*	*W*
Invariance	C<T>	T	✓	✓
Covariance	C<? extends T>	T and sub-types	✓	✗
Contravariance	C<? super T>	T and basis-types	✗	✓
Bivariance	C<?>	All	✗	✗

INVARIANCE

```
static <T> void move(Stack<T>, from, Stack<T> to) {
  while(!from.isEmpty()) to.push(from.pop());
}
```

Example 6:

```
var rectS = new Stack<Rectangle>();
var graphS = new Stack<Graphic>();
graphS.push(new Rectangle());          // ok
move(rectS, graphS);                   // error
Stack<Object> objS = graphS;           // error
```

Example 7: Runtime errors

```
String[] strA = new String[10];
Object[] objA = strA;                  // ok
objA[0] = Integer.valueOf(2);          // runtime err
```

Example 8: Even more

```
ArrayList<String> sA = new ArrayList<>();
ArrayList<Object> oA = sA;             // error
// error
List<Graphic> l = new ArrayList<Rectangle>();
// ok
Rectangle[] rectangleStack = new Rectangle[10];
Graphic[] graphicStack = new Graphic[10];
graphicStack = rectangleStack;
// ok
Stack<Rectangle> rectangleStack = new Stack<>();
Stack<Graphic> graphicStack = new Stack<>();
for (Rectangle r : rectangleStack)
  graphicStack.push(r);
```

COVARIANCE

- Allows a subtype to be treated as a supertype. Eg. `List<Dog>` can be used where a `List<Animal>` is expected.

```
public class Stack<E> {
  public void pushAll(Iterable<? extends E> src) { }
}
```

Example 9:

```
Stack<? extends Graphic> graphS;
graphS = new Stack<Rectangle>();       // ok
graphS = new Stack<Circle>();          // ok
graphS = new Stack<Object>();          // error (duh)
```

Example 10: Read-only

```
graphS.push(new Graphic());            // error
graphS.push(new Rectangle());          // error
graphS.push(new Triangle());           // error
```

CONTRAVARIANCE

- Allows a supertype to be treated as a subtype. Eg. `List<Animal>` can be used where a `List<Dog>` is expected.

```
static <T> void addToC(List<? super T> list, T e) {
  list.add(e);
}
```

Example 11: Ok

```
addToC(new ArrayList<Number>(), 3); // ok
addToC(new ArrayList<Object>(), 3); // ok
```

Example 12: Shenanigans

```
Stack<? super Graphic> s = new Stack<Object>();
s.add(new Graphic());                  // ok
s.add(new Rectangle());                // ok
Graphic g = s.pop();                   // error
Object g = s.pop();                    // ok
```

BIVARIANCE

- If concrete type doesn't matter

```
static void printList(List<?> list) {
  for (Object elem : list)
    System.out.print(elem + " ");   // ok
}
```

Example 13: Shenanigans

```
static void appendNewObject(List<?> list) {
  list.add(new Object());           // error
}
```

TYPICAL “DOES THIS COMPILE MR. LINTER?” QUESTIONS

```
public class GenericsSyntax<T extends Iterable<T>> {
    public void fun(Iterable<? super T> i) { } // ok
}
// ok
List<? extends Object> v = new ArrayList<String>();
```

Example 14: Complete example

```
public class VarianceExamples {
  public static double sum(
      Collection<? extends Number> numbers) {
    double sum = 0;
    for (Number num : numbers)
      sum += num.doubleValue();
    return sum;
  }
  public static void addNrs(List<? super Integer>
      list, Collection<? extends Integer> source) {
    list.addAll(source);
  }
  public static <T extends Comparable<T>> T
      findMax(Collection<T> coll) {
    return coll.stream()
               .max(Comparator.naturalOrder())
               .orElse(null);
  }
  public static <T> List<T> filterByType(
      Collection<?> source, Class<T> clazz) {
    List<T> result = new ArrayList<>();
    for (Object obj : source)
      if (clazz.isInstance(obj))
        result.add(clazz.cast(obj));
    return result;
  }
}
```

ANNOTATIONS

- Metadata
- Macros for the poor people forced to work corporate
- `@Override`, `@Test`, `@Json...`

DEFINING

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
  String name();
  String date() default "N/A";
}

@Author(name = "UwU", date = "idon'tvalidateinput")
public class WeeOoo { }
```

REFLECTION

- Information of Class (private and public): Methods, Attributes, Parent class, Implemented interfaces
- Calling methods possible
- Usages: Debugger, Serialization, Frameworks, Remote Procedure Call, ORM

```
// all of these `throws SecurityException`
public Field[] getDeclaredFields()
public Method[] getDeclaredMethods()
public Constructor<?>[] getDeclaredConstructors()

System.out.println(dump(new Student("a", "b"), 0));
// {
//   firstName = a
//   lastName = b
// }
Class c = "s".getClass(); // java.lang.String
```

METHOD

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Init { }
// ...
public class User {
  private String needsInit;
  @Init
  private void initObj() {
    this.needsInit = "wowie";
  }
}
```

CLASS

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface JsonSerializable { }
// ...
@JsonSerializable
public class User { }
```

Also, this is a thing:

```
Class z = Class.forName("Test");
Test c = z.getDeclaredConstructor().newInstance();
// ==
Test c = new Test();
```

ATTRIBUTE

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD })
public @interface JsonElement {
    public String key() default "";
}
// ...
public class User {
    @JsonElement
    private String firstName;
}
```

VALIDATION

```
@Min(value = 18, message = "Age must be ⩾ 18")
@Max(value = 99, message = "Age must be ⩽ 99")
private int age;
@NotNull(message = "Name cannot be null")
private String name;
```

Example 15: Complete example

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface WebController { }
// ...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Get {
  String path();
  String contentType() default "application/json";
}
// ...
@WebController
public class Controller {
  private User user = new User("frank", "meier");
  @Get(path = "/user")
  public String user() {
    return new ObjectToJsonConverter()
                .convertToJson(user);
  }
}
// ...
public class WebFramework {
  public void addCtrl(Class<?> controllerClass)
              throws Exception {
  if (!controllerClass
        .isAnnotationPresent(WebController.class))
    throw new Exception("Is not a WebController");

  for (Method method : controllerClass
                .getDeclaredMethods())
    if (method.isAnnotationPresent(Get.class)) {
      method.setAccessible(true); // best practice
or sth, idk, i write code in actually good languages
      var annot = method.getAnnotation(Get.class);
      routeHandlers.put(
        annot.path(),
        new WebHandler<>(method,
                         controllerClass
                           .getDeclaredConstructor()
                           .newInstance(),
                         annot.contentType())
      );
    }
  }
}
```

ALGORITHMS

Brute-force: slow, stupid, go shame yourself
Greedy: Take best option each step (optimize locally). Fast. Not necessarily global optimum
Backtracking: Trial and error. Best overall option. Higher compute costs
Divide and conquer: Binary search
Dynamic programming: Can be used as recursion optimization. Uses Tabulation (bottom-up) or Memoization (top-down) or more generally a reuse of previous results

Example 16: **DP: fibonacci**

```
public static long fib(int n) {
  if (n ⩽ 1) return n;
  long prev2 = 0;
  long prev1 = 1;
  for (int i = 2; i ⩽ n; i++) {
    long cur = prev1 + prev2;
    prev2 = prev1;
    prev1 = cur;
  }
  return prev1;
}
```

Example 17: **DP: minJumps**

```
public int minJumps(int[] nums) {
  int n = nums.length;
  int[] dp = new int[n];
  Arrays.fill(dp, Integer.MAX_VALUE);
  dp[0] = 0;
  for (int i = 0; i < n; i++)
    for (int j = 1; j ⩽ nums[i] && i + j < n; j++)
      dp[i + j] = Math.min(dp[i + j], dp[i] + 1);
  return dp[n - 1];
}
```

BACKTRACKING

1) Choose one of the possible paths to solution
2) If path leads to goal → done
3) Else → backtrack to previous decision
4) Choose next path, iterate further from Step 1)

Example 18: **Sudoku**

```
boolean checkRow(int row, int num) {
  for (int c : arr[row])
    if (c == num) return false;
  return true;
}
boolean checkCol(int col, int num) {
  for (int row : arr)
    if (row[col] == num) return false;
  return true;
}
boolean checkBox(int row,int col, int num) {
  row = (row / 3) * 3;
  col = (col / 3) * 3;
  for (int i = row; i < row + 3; i++)
    for (int j = col; j < col + 3; j++)
      if (arr[i][j] == num) return false;
  return true;
}
boolean solve(int row, int col) {
  if (row == 8 && col == 9) return true;
  if (col == 9) {
    row++;
    col = 0;
  }
  if (arr[row][col] =! 0) {
    return solve(row, col + 1);
  } else {
    for (int num = 1; num < 10; num++) {
    if (checkRow(row, num) && checkCol(col, num)
        && checkBox(row, col, num)) {
      arr[row][col] = num;
      if (solve(row, col + 1)) return true;
    }
  }
  arr[row][col] = 0;
  return false;
}
```

Example 19: **KnightsTour**

1	6	15	10	21
14	9	20	5	16
19	2	7	14	11
8	13	24	17	4
H	18	3	12	23

```
boolean tour(int[][] visited, int x,int y,int pos) {
  visited[x][y] = pos;
  if (pos ⩾ N * N) return true;
  for (int k = 0; k < 8; k++) {
    int newX = x + row[k];
    int newY = y + col[k];
    if (isValid(newX, newY)
        && visited[newX][newY] == 0
        && tour(visited, newX, newY, pos + 1))
        return true;
  }
  visited[x][y] = 0;
  return false;
}
```

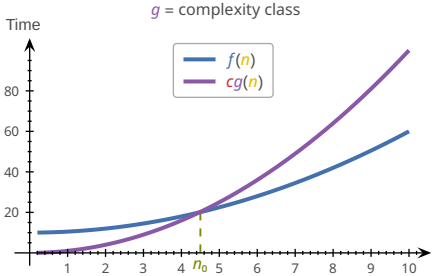
BIG O NOTATION

– Upper bound of f
– Worst case scenario
– Atomic operations = constant time
– Runtime measured as sum of primitive operations

Notation

f	Algorithm	n	Size of Problem
g	Complexity Class	c	> 0
n_0	≥ 1		

$f(n)$ = Number of steps
g = complexity class



$f(n)$ is $O(g(n))$ if $c > 0 \wedge n_0 \geq 1$
$f(n) \leq cg(n)$ for $n \geq n_0$

f in order g:

$$f, g : \mathbb{N} \to \mathbb{N}$$
$$n_0, c \in \mathbb{N} \wedge \forall n \geq n_0.\ (f(n) \leq c \cdot g(n))$$
$$\Rightarrow f \in O(g) \Leftrightarrow f = O(g)$$

$O(n)$ = Complexity class

RULES

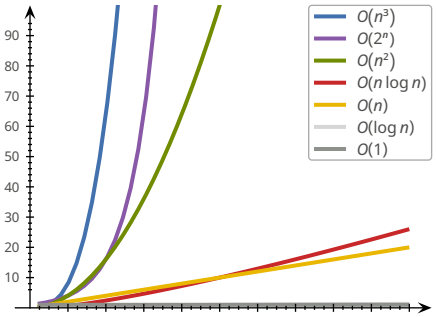
If $f(n)$ is polynomial of degree d, then $f(n) \in O(n^d)$
– ignore lower powers
– ignore constants

Use most optimal function (lowest power)
– $2n$ is $O(n)$ better than $2n$ is $O(n^2)$

Simplify as far as possible
– $3n + 5$ is $O(n)$ instead of $3n + 5$ is $O(3n)$

COMPLEXITY CLASSES

Name	Class	Example
Constant	1	Array read
Logarithmic	$\log n$	Binary search
Linear	n	Linear search
Log-linear	$n \log n$	Merge+Quick sort
Quadratic	n^2	Bubble+Insertion sort
Cubic	n^3	Solve equation
Polynomial	n^k	Various algorithms
Exponential	2^n	Calculate all subsets
Factorial	$n!$	Calculate all permutations



MAFS

$$1 + 2 + \ldots + n = \sum_{k=1}^{n} = \frac{n(n+1)}{2}$$
$$1 + q + q^2 + \ldots + q^n = \sum_{k=0}^{n} q^k = \frac{1 - q^{n+1}}{1 - q}$$
$$\overset{q=2, n=\log_2(n)}{=} 2n - 1$$

EMPIRICAL MEASUREMENTS

Measured performance from experiments/benchmarks on real data and a specific environment. Common drawbacks:

Environment dependence: Different hardware, OS, etc. can change timings.
Input bias: Benchmarks may use "easy" or unrepresentative data: performance can degrade on real-world distributions.
Noise & variance: Background processes, thermal throttling, and random effects can make results inconsistent.
Limited coverage: You only measure a finite set of n values and cases. Behavior outside that range may be different.
Hard to compare fairly: Small differences in implementation details (memory layout, allocations, logging) can dominate.
Constant factors can mislead: Empirical "looks linear" in a range, even if the true asymptotics are worse (e.g., $O(n^2)$ but small n won't show it).
Non-repeatability over time: Updates to runtimes/CPUs/frameworks can change outcomes.

SORTING ALGORITHMS

BOGOSORT

The best sorting algorithm

STALINSORT

The most efficient sorting algorithm

MIRACLE SORT

The most magical sorting algorithm

SELECTION SORT

– Search for smallest elem in unsorted part and move in front
– Less mutations in array
– Same performance even if list almost sorted

1	3	9	6	2
1	2	3	9	6
1	2	3	9	6
1	2	3	6	9

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} \sim O(n^2)$$

Example 20:

```
public static void selectionSort(int[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++) {
    int minimum = i;
    for (int j = i + 1; j < n; j++)
      if (a[j] < a[minimum])
        minimum = j;
    swap(a, i, minimum);
  }
}
```

INSERTION SORT

– Take elem and put into correct place in sorted part
– Good for already partially sorted arrays

1	3	9	6	2
1	3	9	6	2
1	3	9	6	2
1	3	6	9	2

$$1 + \ldots + (n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} \sim O(n^2)$$

```
public static void insertionSort(int[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++)
    for (int j = i; j > 0 && a[j] < a[j - 1]; j--)
      swap(a, j, j - 1);
}
```

BUBBLE SORT

– Compare neighboring elems and swap if needed

1	3	9	6	2
1	3	9	6	2
1	3	9	6	2
1	3	6	9	2

```
public static void bubbleSort(int[] a) {
  for (n = a.length; n > 1; n--)
    for (i = 0; i < n - 1; i++)
      if (a[i] > a[i + 1])
        swap(a, i, i + 1);
}
```

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} \sim O(n^2)$$

COUNTING SORT

– For int-like data only i guess
– Count how often (n) values of index i appear in array, then insert i n times in new array

1	3	2	0	2
1	1	2	1	0
0	1	2	2	3

```
public static void countingSort(int[] a) {
  int k = Arrays.stream(arr).max().getAsInt();
  int[] c = new int[k + 1];
  for (int i : a) c[i] += 1;
  int pos = 0;
  for (int i = 0; i < k; i++)
    for (int j = 0; j < c[i]; j++) {
      a[pos] = i;
      pos++;
    }
}
```

$$n + n + k + n = 3n + k \sim O(n + k)$$

SHELL SORT

– Sort pairs of far away elems
– Sort the partially sorted array through insertion sort

1	3	9	6	2
1	3	9	6	2
1	2	9	6	3
1	3	9	6	2

```
public static void shellSort(int[] a) {
  int n = a.length;
  int h = 1;
  while (h < n/3) h = 3 * h + 1;
  while (h ⩾ 1) {
    for (int i = h; i < n; i++)
      for (int j = i; j ⩾ h && a[j] < a[j - h];
           j = j - h) swap(a, j, j - h);
    h /= 3;
  }
}
```

$O(n^2)$

BINARY SEARCH

```
// recursive
static <T extends Comparable<T>> int bsRec(
    List<T> data, T target, int low, int high) {
  if (low > high) return -1;
  int pivot = low + ((high - low) / 2);
  if (target.equals(data.get(pivot)))
    return pivot;
  else if (target.compareTo(data.get(pivot)) < 0)
    return bsRec(data, target, low, pivot - 1);
  else
    return bsRec(data, target, pivot + 1, high);
}
// iterative
static <T extends Comparable<T>> int bsIter(
    List<T> data, T target) {
  int low = 0;
  int high = list.size() - 1;
  while (low ⩽ high) {
    int pivot = low + ((high - low) / 2);
    int res = target.compareTo(data.get(pivot));
    if (res > 0)
      low = pivot + 1;
    else if (res < 0)
      high = pivot - 1;
    else
      return pivot;
  }
  return -1;
}
```

$O(\log n)$

RECURSION

When a function calls itself. FP <3

LINEAR RECURSION

Recursive call starts ***at most one*** further recursive call

RECURSIVE → ITERATIVE

```
static int[] reverseArrRec(int[] a, int i, int j) {
  if (i ⩾ j) return a;
  int temp = a[j];
  a[j] = a[i];
  a[i] = temp;
  return reverseArrRec(a, i + 1, j - 1);
}
static int[] reverseArrIter(int[] a, int i, int j) {
  while (i < j) {
    int temp = a[j];
    a[j] = a[i];
    a[i] = temp;
    i = i + 1;
    j = j - 1;
  }
  return a;
}
```

TAIL RECURSION

Linear recursion where the recursive call is ***last***

```
static int recsum(int x) {
  if (x == 0) return 0;
  else return x + recsum(x - 1);
  // no tail recursion because "+" is evaluated last
}
static int tailrecsum(int x, int total) {
  if (x == 0) return total;
  else return tailrecsum(x - 1, total + x);
}
```

BINARY RECURSION

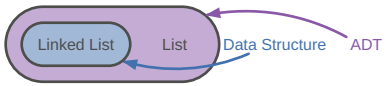
All non-terminal calls have ***two*** recursive calls

```
static int binsum(int low, int high) {
  if (low > high)
    return 0;
  else if (low == high)
    return low;
  else {
    int mid = (low + high) / 2;
    return binsum(low, mid) + binsum(mid + 1, high);
  }
}
```

DATA STRUCTURES

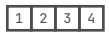
ADT: Abstract Data Type (e.g. Interface)

- Describes "What", not "How"
- Describes Attributes, Operations and Exceptions
- Doesn't provide implementation details, that is done by the concrete data type



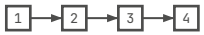
Data structure	*Access*	*Search*	*Insertion*	*Deletion*
Array	*O*(1)	*O*(*n*)	*O*(*n*)	*O*(*n*)
Linked list	*O*(*n*)	*O*(*n*)	*O*(*n*)	*O*(*n*)
Doubly Linked List	*O*(*n*)	*O*(*n*)	*O*(1)	*O*(1)
Stack	*O*(*n*)	*O*(*n*)	*O*(1)	*O*(1)
Queue	*O*(*n*)	*O*(*n*)	*O*(1)	*O*(1)
Binary Tree	*O*(*n*)	*O*(*n*)	*O*(*n*)	*O*(*n*)
Binary Search Tree	*O*(*n*)	*O*(*n*)	*O*(*n*)	*O*(*n*)
Hash Table	*O*(*n*)	*O*(*n*)	*O*(*n*)	*O*(*n*)

ARRAY




```
int[] aAaAhrray = {69, 420, 1337};
```

LINKED LIST




```
class LinkedList<T> {
  Node head;
  static class Node {
    T data;
    Node next;
    Node(T d) { data = d; }
  }
  public T insert(T data) {
    Node n = new Node(data);
    if (list.head == null)
      list.head = n;
    else {
      Node last = list.head;
      while (last.next != null)
        last = last.next;
      last.next = n;
    }
    return data;
  }
}
```

DOUBLY LINKED LIST


```
class DoublyLinkedList<T> {
  Node head;
  static class Node {
    T data;
    Node next, prev;
    Node(T d) { data = d; }
  }
  public T insert(T data) {
    Node n = new Node(data);
    if (list.head == null)
      list.head = n;
    else {
      Node last = list.head;
      while (last.next != null)
        last = last.next;
      n.prev = last;
      last.next = n;
    }
    return data;
  }
}
```

STACK

LIFO 


```
class Stack<T> {
  private int maxSize;
  private T[] arr;
  private int top;
  public Stack(int size) {
    maxSize = size;
    arr = (T[]) new Object[maxSize];
    top = -1;
  }
  public void push(T value) {
    if (top == maxSize - 1)
      throw new StackOverflowError();
    arr[++top] = value;
  }
  public T pop() {
    if (top == -1)
      throw new StackUnderflowError();
    return arr[top--];
  }
}
```

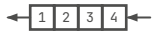
PRIORITY QUEUE

- Saves priority with each element
- Returns elements with lowest/highest prio first

Method	*Unsorted List*	*Sorted List*
insert	*O*(1)	*O*(*n*)
min	*O*(*n*)	*O*(1)
removeMin	*O*(*n*)	*O*(1)

```
public interface Entry<K,V> {
  K getKey();
  V getValue();
}
public interface PriorityQueue<K,V> {
  int size();
  boolean isEmpty();
  Entry<K,V> insert(K key, V value);
  Entry<K,V> min();
  Entry<K,V> removeMin();
}
```

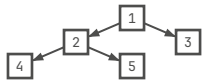
QUEUE

FIFO 


```
class Queue<T> {
  private int maxSize;
  private T[] arr;
  private int front, rear;
  public Queue(int size) {
    maxSize = size;
    arr = (T[]) new Object[maxSize];
    front = rear = -1;
  }
  public T front() {
    if (front == -1)
      throw new StackUnderflowError();
    return arr[front];
  }
  public T rear() {
    if (rear == -1)
      throw new StackUnderflowError();
    return arr[rear];
  }
  public void enqueue(T value) {
    if (this.empty()) {
      front = 0;
      rear = 0;
      arr[front] = value;
    } else {
      front++;
      if (front < maxSize) arr[front] = value;
      else // rebuild array
    }
  }
  public T dequeue() {
    if (this.empty())
      throw new StackUnderflowError();
    T ret = arr[rear];
    if (front == rear) front = rear = -1;
    else rear++;
    return ret;
  }
  boolean empty() {
    return front == -1 && rear == -1;
  }
}
```

TREE

0 (Root)
1 (Inner)
2 (Leaf)

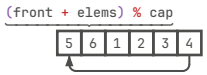



```
public class Tree<T> {
  private Node<T> root;
  public Tree(T rootData) {
    root = new Node<T>();
    root.data = rootData;
  }
  public static class Node<T> {
    private T data;
    private Node<T> left, right;
  }
}
```

REVERSING BINARY TREE

```
public static void swap(Node root) {
    if (root == null) return;
    Node tmp = root.left;
    root.left = root.right;
    root.right = tmp;
}
public static void mirror(Node root) {
    if (root == null) return;
    mirror(root.left);
    mirror(root.right);
    swap(root);
}
```

RING BUFFER

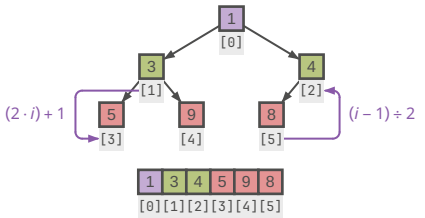



```
class RingBuffer {
  private int[] buffer;
  private int size;
  private int start;
  private int end;
  private boolean isFull;

  public RingBuffer(int size) {
    if (size ⩽ 0)
      throw new IllegalArgumentException("size > 0");
    this.size = size;
    this.buffer = new int[size];
    this.start = 0;
    this.end = 0;
    this.isFull = false;
  }

  public void append(int item) {
    buffer[end] = item;
    end = (end + 1) % size;
    if (isFull)
      start = (start + 1) % size;
    if (end == start)
      isFull = true;
  }

  public List<Integer> getData() {
    List<Integer> items = new ArrayList<>();
    if (!isFull) {
      for (int i = 0; i < end; i++)
        items.add(buffer[i]);
    } else {
      for (int i = start; i < size; i++)
        items.add(buffer[i]);
      for (int i = 0; i < end; i++)
        items.add(buffer[i]);
    }
    return items;
  }
}
```

HEAP

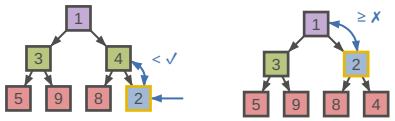
- Binary tree
- Layers 0, ..., *h* – 1 fully populated, layer *h* filled from left
- Operations always lowest RHS so that tree stays consistent

Min-Heap: Keys of nodes are **smaller** or equal than children's
Max-Heap: Keys of nodes are **larger** or equal than children's



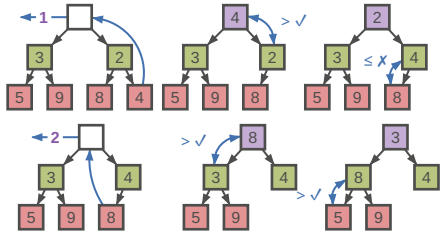
ENQUEUE

1) Insert into lowest RHS
2) Compare inserted elem recursively with parent
3) Swap if parent is larger, stop when parent is smaller



DEQUEUE

1) Remove root
2) Put lowest RHS into root pos
3) Compare moved elem recursively with smaller child
4) Swap if child is smaller, stop when no child is smaller
5) Tree is now heapified again, repeat at step 1



HEAP SORT

Dequeue elems from head into list, resorting tree on each iter

```
public class HeapSort {
  private static <T> int getLargest(T[] array, int n,
      int parent, Comparator<T> comparator) {
    int left = 2 * parent + 1;
    int right = 2 * parent + 2;
    int largest = parent;
    if (left < n && comparator.compare(array[left],
                                       array[largest]) < 0)
      largest = left;
    if (right < n && comparator.compare(array[right],
                                        array[largest]) < 0)
      largest = right;
    return largest;
  }

  static <T> void heapify(T[] arr,
      Comparator<T> comparator) {
    int n = arr.length;
    for (int i = n / 2 - 1; i ⩾ 0; i--)
      heapify(arr, i, comparator);
  }

  public static <T> void heapify(T[] array,
      int parent, Comparator<T> comparator) {
    int largest = getLargest(array, array.length,
      parent, comparator);
    if (largest == parent) return;
    T temp = array[parent];
    array[parent] = array[largest];
    array[largest] = temp;
    heapify(array, largest, comparator);
  }

  static <T> void percolate(T[] array, int parent,
      int n, Comparator<T> comparator) {
    int largest = getLargest(array, n,
      parent, comparator);
    if (largest == parent) return;
    T temp = array[parent];
    array[parent] = array[largest];
    array[largest] = temp;
    percolate(array, largest, n, comparator);
  }

  static <T> void sort(T[] arr,
      Comparator<T> comparator) {
    int n = arr.length;
    heapify(arr, comparator.reversed());
    while (n > 1) {
      T temp = arr[0];
      arr[0] = arr[n - 1];
      arr[n - 1] = temp;
      n = n - 1;
      percolate(arr, 0, n, comparator.reversed());
    }
  }
}
```

BINARY SEARCH TREE (BST)

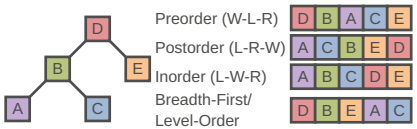
– All subnodes of **left** subtree are **smaller** than root
– All subnodes of **right** subtree are **larger** than root

Implementations:

Array	*Node*
Needs good API	Intuitive
Not memory-efficient	Only uses as much memory as needed
Locally faster	Also fast
Low allocation overhead	Do java devs really care about this?

Array-based implementation vs Node-based implementation

TRAVERSING



Example 21:

```
private void enlargeArray() {
  Integer[] tmp = tree;
  tree = new Integer[tmp.length * 2];
  for (int i = 0; i < tmp.length; i++)
    tree[i] = tmp[i];
}
private int getLeftIndex(int parentIndex) {
  return 2 * parentIndex + 1;
}
private int getRightIndex(int parentIndex) {
  return 2 * parentIndex + 2;
}
public void insert(int value) {
  insertRec(value, 0);
}
private void insertRec(int v, int i) {
  if (i ≥ tree.length) enlargeArray;
  int curv = tree[i];
  if (curv == null) {
    tree[i] = v;
  } else if (curv > v) {
    insertRec(v, getRightIndex(i));
  } else {
    insertRec(v, getLeftIndex(i));
  }
}
```

MORE DATA STRUCTURES

Set: Elements of same type, no duplicates, without order
Multiset: Set with duplicates
Map: KV-pairs with unique keys (put, get, remove $O(n)$)
Multimap: Key can have multiple values

HASHING

Hash-function h maps e onto $[0, n-1]$. $h(e)$ = position of e

$$e \xrightarrow{\text{hash code}} \mathbb{Z} \xrightarrow{\text{compression function}} \overbrace{[0, n-1]}^{\text{Index}}$$

Properties of good hash functions:
– Equal distribution
– Fast computation
– Constant time complexity
– Deterministic
– Incorporates as much information from key as possible

HASH FUNCTIONS

– Modulo primes x mod p
  – Primes reduce collisions and distribute values better
– Memory address (`Object.hashCode()` default)
– Byte/Integer cast `ByteBuffer.wrap(b).getInt()`
– Component sum (eg. sum string char codepoints)
– Polynomial accumulation $a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
  – $z = 33$ for strings
  – Usually less collisions than component sum
– Horner-Schema (identical, but easier to compute):
  – $a_0 + z \cdot (a_1 + z \cdot (a_2 + \ldots + z \cdot a_{n-1}))$

EQUALITY

It is generally necessary to override `hashCode` whenever `equals` is overridden

```
x.equals(y) => x.hashCode() == y.hashCode()
```

HASHING OBJECTS

– primitive types → builtin hash-code
– arrays → apply on each elem
– objects → apply recursively

```
// Horner schema with z = 31 and start = 17
@Override
public int hashCode() {
  int result = 17;
  for (Object field : fields) result = 31 * result
      + (field == null ? 0 : field.hashCode());
  return result
}
```

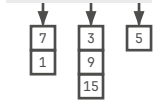
ANOMALIES

– Empty spaces
– Collisions
– Overflow

MANAGING HASH COLLISIONS

CLOSED ADDRESSING/SEPARATE CHAINING

Each bucket is a list.
Search: O(entries/buckets) = $O(^n/_N)$
"Lastfaktor" $a = n/N$ (a can be > 1)
Large N: more storage, faster
Small N: less storage, slower



– Acceptable perf. under high load
– Larger storage overhead

OPEN ADDRESSING

e "overflows" into next empty bucket
→ Must probe 2 times in example

Problem: Primary Clustering

	put 89	*put 69*	*put 10*	*put 58*	*put 11*	*put 12*
0		69	69	69	69	69
1			10	10	10	10
2					11	11
3						12
4						
5						
6						
7						
8				58	58	58
9	89	89	89	89	89	89

Linear probing: $s(k, i) = h(k) + i$
Linear probing backwards: $s(k, i) = h(k) - i$
Quadratic probing: $s(k, i) = h(k) + i^2$

Deletion

– Only mark as "deleted"
– Move next best candidate to deleted position

Access

Probability for finding free spot:

n = entries $\quad N$ = buckets

$a = \frac{n}{N}$ = probability: occupied $\quad p$ = probing steps

$P(X = p) = a^{p-1}(1 - a)$

$E(X) = \sum_{x=1}^{\infty} p \cdot P(X = p) = \frac{1}{1-a}$ = Nr. accesses

a can never be bigger than 1

– No storage overhead
– Resize/Rehash compute intensive
– Bad performance under high load

After $a > 0.8$ the probing steps increase exponentially.

Example 22:

0	1	2	3	4	5	6	7	8	9	
10	20	2	12		15		7	17		$N = 10$ (Buckets)
										$n = 7$ (Entries)

$a = \frac{N}{n} = \frac{7}{10} = 0.7 = 70\%$ = Lastfaktor

$\frac{1}{1-a} = \frac{1}{1-0.7} = \frac{1}{0.3} = 3.\overline{3}$ = Anzahl Sondierungen

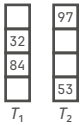
How high is the probability that the first two positions are occupied and the third one is free with an occupancy rate of 0.2?

$$(a^{p-1}) \cdot (1 - a) = 0.2^2 \cdot 0.8 = 0.04 \cdot 0.8 = 0.032$$

seems kinda wrong, innit

CUCKOO-HASHING

$h_1(x) = x \bmod 5 \qquad h_2(x) = x/5 \bmod 5$



Insert: If $h_1(x) =$ null ⇒ T_1, else push current y into T_2. If no cycle → $O(1)$, if rehash is needed → $O(n)$ (`MAX_LOOPS`)
Search: $O(1)$ ⇒ only has to check 2 positions
Rehashing: $x \bmod n \Rightarrow x \bmod (n + m)$

– Constant time (if no rehashing) due to only 2 possible places
– No long probing chains
– Potential of rehashing

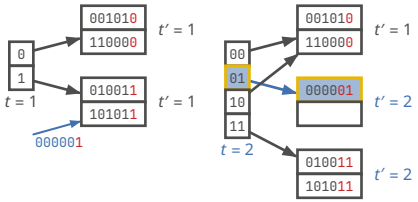
Example 23:

```
public void insert(int key) {
  int MAX_LOOPS = 100;
  for (int i = 0; i < MAX_LOOPS; i++) {
    var t1 = table1[hash1(key)];
    table1[hash1(key)] = key;
    if (t1 == null) return;
    key = t1;
    var t2 = table2[hash2(key)];
    table2[hash2(key)] = key;
    if (t2 == null) return;
    key = t2;
  }
  rehash();
  insert(key);
}
```

EXTENDIBLE HASHING

– Resize/Rehash easier
– Good performance under high load
– Larger storage overhead

```
// getting index
int idx = key.hashCode() & ((1 << globalDepth) - 1);
// when to rehash
if (bucket.getLocalDepth() == globalDepth)
  growHashTable();
else
  splitBlock();
// splitting block
int highBit = 1 << localDepth;
for (Entry e : page.entries) {
    int h = e.key.hashCode();
    Page newPage = (h & highBit) != 0 ? p1 : p0;
    newPage.put(e.key, e.value);
}
```



DESIGN PATTERNS

Splitting algorithms and datastructures. Every time an OOP programmer has an idea of how to do something better they inevitably re-invent a concept from the functional paradigm.

TYPES

Creational: Abstraction and instantiation (e.g., Factory, Singleton)
Structural: Composition of classes and objects into larger structures (e.g., Adapter, Facade)
Behavioral: Algorithms and distribution of responsibility among objects (e.g., Iterator, Visitor)

ITERATOR

```
interface Iterable<T> {
  Iterator<T> iterator();
}
public interface Iterator<E> {
  boolean hasNext();
  E next() throws NoSuchElementException;
  void remove() throws
    UnsupportedOperationException,
    IllegalStateException;
}
```

Example 24: **Usage**

```
for (String s : stringList) { }       // ==
for (Iterator<String> i = stringList.iterator();
    i.hasNext()) String s = i.next(); // ==
Iterator<String> iter = stringList.iterator();
while (i.hasNext()) String s = i.next();
// Tree → Depth-first Iterator
//      → Breadth-first Iterator
```

Example 25: **Implementation**

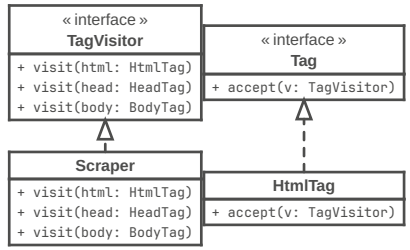
```
public class AttackOrderIterator
       implements Iterator<Enemy> {
  private List<Enemy> attackOrder;
  private int idx = 0;
  public AttackOrderIterator(Enemy enemy){
    this.attackOrder = getAttackOrder(enemy);
  }
  private List<Enemy> getAttackOrder(Enemy enemy) {
    // ...
  }
  @Override
  public boolean hasNext() {
    return idx < this.attackOrder.size();
  }
  @Override
  public boolean next() {
    return this.attackOrder.get(idx++);
  }
}

public abstract class Enemy
       implements Iterable<Enemy> {
  abstract Enemy[] getSubordinates();
  abstract void attack();
  @Override
  public Iterator<Enemy> iterator() {
    return new AttackOrderIterator(this);
  }
}
```

PRODUCER / CONSUMER

`from` **produces** entries, `to` **consumes** entries

```
<T> void move(Stack<? extends T> from,
              Stack<? super T> to) {
  while (!from.isEmpty())
    to.push(from.pop());
}
```

VISITOR

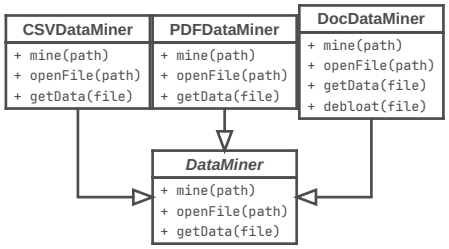


– separate algorithms and data
– keep algorithms in one place and not distributed over objs

```
public interface TagVisitor {
  void visit(HtmlTag html);
  void visit(HeadTag head);
  void visit(BodyTag body);
}
public class HtmlTag implements Tag {
  @Override
  public void accept(TagVisitor visitor) {
    visitor.visit(this);
    headTag.accept(visitor);
    bodyTag.accept(visitor);
    visitor.leave(this);
  }
}
```

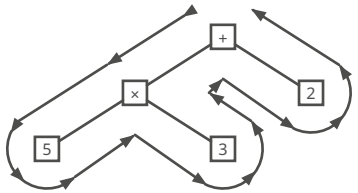
TEMPLATE METHOD

Break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method.

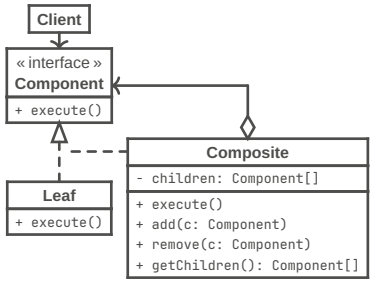


EULER TOUR TRAVERSING

Each node is traversed 3 times
– left (preorder)
– bottom (inorder)
– right (postorder)



COMPOSITE




```
interface Component {
  void printInfo();
}
class Item implements Component {
  private String name;
  public Item(String name) {
    this.name = name;
  }
  @Override
  public void printInfo() {
    System.out.println("Item: " + name);
  }
}
class Box implements Component {
  private List<Component> contents =
    new ArrayList<>();
  public void add(Component component) {
    contents.add(component);
  }
  public void remove(Component component) {
    contents.remove(component);
  }
  @Override
  public void printInfo() {
    for (Component c : contents)
    c.printInfo(); // Delegate to children
  }
}
```

more than enough room left for music

el comite

turnstile

condor

hirihiri

retrogott

wisdom in chains

tombeau

siggy retts

missing persons

hands of god

prison affair

der taeubling

nargaroth

ITERATORS

```
Iterator<String> it = stringList.iterator();
while (it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

Mutating Collection while iterating over it: ConcurrentModificationException

EXCEPTIONS

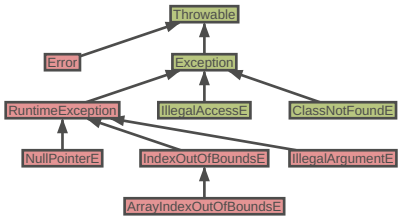
Error	*Exception*
Critical, don't handle	Runtime, handleable
OutOfMemoryError, StackOverflowError, AssertionError	IOException

Checked	*Unchecked*
Must be handled (or throws-declaration)	Not necessary
Checked by compiler	Compiler doesn't check
Exception, not RuntimeException	RuntimeException, Error

Child Exception gets caught in catch clause with parent class

```
void test() throws ExceptionA, ExceptionB {
  String c = clip("asdf");
  throw new ExceptionB("wack");
}

// finally ALWAYS executes, even on unhandled Exc.
try {
  test();
} catch (ExceptionA | ExceptionB e) {
  // ...
} finally { }

try { ... } catch(NullPointerException e) {
  throw e; // ⟶leaves blocks⟶
} catch (Exception e) {
  // above e won't get caught!
} finally {
  // will still get executed
}

1 / 0; // ArithmeticException div by zero

String s = "";
s = null;
s.toUpperCase(); // NullPointerException

int[] arr = new int[] {1, 2, 3};
int elem = arr[8]; // ArrayIndexOutOfBoundsException
```

Unchecked **Checked**



IMPORTANT STUFF

- Hashing should be added to equals fn's for strict equality
- Check if `input == null`
- Check if `array.length == 0`
- `IllegalArgumentException("reason")`
- try/catch finally block **always** executes

IO

```
try (var fr = new FileReader("text.txt")) {
  int input = fr.read();
  while (input ⩾ 0) {
    if (input == ';') { /* do something */ }
    input = fr.read();
  }
}

try (FileWriter writer = new FileWriter("out.txt",
    StandardCharsets.UTF_8, true)) { // append
  writer.write("weeoo\n");
}

try {
  var input = new FileInputStream("text.txt");
  int i = input.read();
  while(i =≠ -1) {
     System.out.print((char)i);
     i = input.read();
  }
  input.close();
} catch (Exception e) {
  e.printStackTrace();
}

try (BufferedReader reader = new BufferedReader(
    new FileReader("text.txt",
      StandardCharsets.UTF_8))) {
  String line;
  while ((line = reader.readLine()) =≠ null) {
      System.out.println(line);
  }
}

try (
    FileReader reader = new FileReader("in.txt");
    FileWriter writer = new FileWriter("out.txt")
  ) {
  int i = input.read();
  while(i ⩾ 0) {
    writer.write(i);
    i = input.read();
  }
}
```

TRY WITH

```
try (var output = new FileOutputStream("f.txt")) {
  output.write("Hello".getBytes());
} catch (IOException e) {
  System.out.println("Error writing file");
} finally {
  System.out.println("Done");
}
```

SERIALIZING

```
class X implements Serializable { }
// Serializing
try (var stream = new ObjectOutputStream(
    new FileOutputStream("s.bin"))) {
  stream.writeObject(new X());
}
// Deserializing
try (var stream = new ObjectInputStream(
    new FileInputStream("s.bin"))) {
  X x = (X) stream.readObject();
}
```

FUNCTION

```
public interface Function<T, R> {
  R apply(T t);

  static <T> Function<T, T> identity();

  <V> Function<T, V> andThen(
    Function<? super R, ? extends V> after);

  <V> Function<V, R> compose(
    Function<? super V, ? extends T> before);
}
```

PREDICATE

```
public interface Predicate<T> {
  boolean test(T t);
}

static void removeAll(Collection<Person> collection,
    Predicate criterion) {
  var it = collection.iterator();
  while (it.hasNext())
    if (criterion.test(it.next()))
      it.remove();
}
```

COMPARABLE

```
public interface Comparable<T> {
  int compareTo(T obj);
}

var l = new ArrayList<Integer>(asList(3,2,4,5,1));
l.sort((a, b) ⟶ a > b ? 1 : -1); // ==
l.sort((a, b) ⟶ a - b);          // 1,2,3,4,5

class Person implements Comparable<Person> {
  private final String firstName, lastName;
  @Override
  public int compareTo(Person other) {
    int result = lastName.compareTo(other.lastName);
    if (result == 0)
      result = firstName.compareTo(other.firstName);
    return result;
  }

  static int compareByAge(Person a, Person b) {
    return Integer.compare(a.getAge(), b.getAge());
  }
}
List<Person> people = ...;
Collections.sort(people);
people.sort(Person::compareByAge);
```

COMPARATOR

```
class AgeComparator implements Comparator<Person> {
   @Override
   public int compare(Person a, Person b) {
     return Integer.compare(a.getAge(), b.getAge());
   }
}
Collections.sort(people, new AgeComparator());
people.sort(new AgeComparator());

people.sort(Comparator
  .comparing(Person::getAge)
  .thenComparing(Person::getFirstName)
  .reversed())

Comparator.comparing(Person::getName,
  (s1, s2) ⟶ s2.compareTo(s1));
// ==
Comparator.comparing(Person::getName).reversed();

Comparator<T> nullsLast(Comparator<T> c);
Comparator<T> nullsFirst(Comparator<T> c);
Comparator<T> comparing(Function<T,U> keyExtractor,
  Comparator<U> c);
Comparator<T> comparingInt(ToIntFunction<T,U> f);
```

FUNCTIONALINTERFACE

Any interface with a single abstract method is a functional interface

```
@FunctionalInterface
public interface ShortToByteFunction {
    byte applyAsByte(short s);
}
@FunctionalInterface
public interface PersonStringifier {
    String getNameAndAge(Person p);
}
```

COLLECTION

```
boolean add(E e);
boolean remove(Object o);
boolean equals(Object o);
int hashCode();
int size();
boolean isEmpty();
Object[] toArray();
void clear();
boolean contains(Object o);
boolean addAll(Collection<? extends E> c);
boolean containsAll(Collection<?> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);

Set<String> noDup = new HashSet<>();
```

COLLECTION IMPLEMENTATIONS

```
// List
int indexOf(Object o);
int lastIndexOf(Object o);
E get(int index);
subList(int from, int to);
void sort(Comparator<? super E> c);

// Stack
E peek();
E pop();
E push(E item);
boolean empty();
int search(Object o);

// Queue
E element();      // throws ⟶ peek();     doesn't
E remove();       // throws ⟶ poll();     doesn't
boolean add(E e); // throws ⟶ offer(E e); doesn't

// Set
// (I) SortedSet ⟶ (C) TreeSet
// (C) HashSet, (C) LinkedHashSet

// Map
// HashMap
boolean containsKey(Object key);
boolean containsValue(Object value);
Set<Map.Entry<K, V>>> entrySet();
V get(Object key);
V put(K key, V value);
V putIfAbsent(K key, V value);
V replace(K key, V value);
V remove(Object key);
V getOrDefault(Object key, V defaultValue);
Set<K> keySet();
Collection<V> values();
```

LAMBDAS

```
String pattern = readFromConsole();
//     vvv not final ⟶ Error
while (pattern.length() == 0)
  pattern = readFromConsole();
Utils.removeAll(people, p ⟶
    p.getLastName().contains(pattern));
// local variable ... referenced from a lambda
expression must be final or effectively final

// Predicate      :: a ⟶ boolean
Predicate<Integer> isLarge = (v) ⟶ v > 69420;
// Function       :: a ⟶ b
Function<Integer, String> str = (v) ⟶ "" + v;
// Supplier       :: a
Supplier<String> hello = () ⟶ "Hello, World!";
// Consumer       :: a ⟶ void
Consumer<Integer> consoomer = (v) ⟶ log(v);
// UnaryOperator  :: a ⟶ a
UnaryOperator<Integer> more = (v) ⟶ v * v;
// BinaryOperator :: a ⟶ a ⟶ a
BinaryOperator<Integer> less = (a, b) ⟶ a - b;
```

STREAMS

```
import java.util.stream.*;

people
  .stream()
  .distinct()
  .filter(p ⟶ p.getAge() ⩾ 18)
  .skip(5)
  .limit(10)
  .map(p ⟶ p.getLastName())
  .sorted()
  .forEach(System.out::println);

people
  .stream()
  .reduce(0, (acc,cur) ⟶ acc + cur.getAge());

list.stream().mapToInt(Integer::intValue);
list.stream().mapToInt(Integer::parseInt);
```

OPTIONAL (HASKELL / RUST IN BLOAT)

```
T get(); // NoSuchElementException
boolean isPresent();
void ifPresent(Consumer<T> consumer);
T orElse(T other);
static Optional<T> empty();
static Optional<T> of(T value);
```

METHODS

```
boolean allMatch(Predicate<T> predicate);
boolean anyMatch(Predicate<T> predicate);
boolean noneMatch(Predicate<T> predicate);
static Stream<T> concat(Stream<T> a, Stream<T> b);
Stream<T> distinct();
Stream<R> flatMap(Function<T, R> mapper);
Stream<T> limit(long maxSize);
Stream<T> skip(long maxSize);
Stream<T> sorted(Comparator<T> comparator);
```

TERMINAL OPERATIONS

```
Optional<T> min(Comparator<T> comparator);
Optional<T> max(Comparator<T> comparator);
Optional<T> findAny();
Optional<T> findFirst();
void forEach(Consumer<T> action);
long count();
void forEachOrdered(Consumer<T> action);
// IntStream
average();
sum();
```

COLLECTORS

```
// List
s.collect(Collectors.toList());
// TreeSet
s.collect(Collectors.toCollection(TreeSet::new));
// String
s.collect(Collectors.joining(", "));
// Integer
s.collect(Collectors.summingInt(Person::getAge));
// Map<String, Person>
s.collect(Collectors.groupingBy(Person::getCity));
// Map<String, Integer>
s.collect(Collectors.groupingBy(Person::getCity,
  Collectors.summingInt(Person::getSalary));
// Map<boolean, List<Person>>
s.collect(Collectors.partitioningBy(s ⟶
  s.getAge() > 18))
```

MORE API'S WE "SHOULD" BE PROVIDED WITH

Yes I'm salty, thanks for pointing that out

```
// Integer
static int parseInt(String s);
// System
arraycopy(Object src, int srcPos, Object dest,
    int destPos, int length);
// Arrays
static <T> List<T> asList(T... a);
// example usage
String[] B = new String[4];
HashSet<String> H = new HashSet<>(Arrays.asList(B));
```