1 Primitive Types

Type	Size / Storage	Notes / Range
byte	8 bits	–128 ... 127
short	16 bits	–32.768 ... 32.767
int	32 bits	-2^{31} ... 2^{31-1}
long	64 bits	-2^{63} ... 2^{63-1}
float	32 bits	IEEE 754 floating-point
double	64 bits	double-precision IEEE 754 FP
char	16 bits	characters (Unicode & ASCII), int: 0 ... 65535
boolean	8 bits	`true` or `false`

Operators

Arithmetic:	`+` `-` `*` `/` `%` (modulo)
Increment / Decrement:	`++` `--` (prefix `++var`, postfix `var++`)
Assignment & Compound:	`=` `+=` `-=` `*=` `/=` `%=`
Comparison:	`== (equality)` `!= (difference)` `>` `<` `>=` `<=`
Logical:	`&& (AND)` `|| (OR)` `! (NOT)`

Type casting

Widening (implicit):	`byte -> short -> char -> int -> long -> float -> double`
Narrowing (explicit):	`double -> float -> long -> int -> char -> short -> byte`

```
double a = 3.14;     // narrowing example
int b = (int) a;
```

2 Conditionals

Example random check

```
import java.util.Random;
public class Main {
  public static void main(String[] args) {
      Random rand = new Random();
      int value = rand.nextInt(256);    // random int (0-255)
      value = value * 2;                // multiply by 2
      if (value == 0) {                 // test if 0
         System.out.println("Value is zero");
      } else if (value < 128) {         // test if smaller
         System.out.println("Less than 128");
      } else if (value > 128) {         // test if larger
         System.out.println("Larger than 128");
      } else {                          // equal to 128
         System.out.println("Exactly: " + value );
}}}
```

3 Scanner

```
import java.util.Scanner;  // import the Scanner
Scanner scanner = new Scanner(System.in); // Creation
int x = scanner.nextInt(); // Read the next token as an int
double y = scanner.nextDouble(); // Read the next token as a double
String s = scanner.nextLine(); // Read a whole line of input as String
scanner.close() // Closes this scanner
```

4 Arrays

```
int[] a = new int[10];   // creation of an array of 10 elements
a[3] = 5;                // accessing and modifying the value at index 3
a.length;                // returns the array length -> 10
```

2D arrays

```
int[][] matrix = new int[3][3]; // definition
matrix[1][2]                    //access element at row=1, col=2
```

5 Loops

General loop structure: printing out numbers from 0 to 9

```
for (int i = 0; i < 10 ; i++) {
    System.out.println(i);
}
int i = 0;
while( i < 10 ){
    System.out.println(i);
    i++;
}
```

Jumps

```
break;      // exit the loop
continue;   // skip to next iteration
```

6 Methods

Method skeleton

```
[modifiers] [return type] name( parameter1, parameter2, ... ) {
    // body
    return value; // unless void
}
```

Call / Invocation

```
name( argument1, argument2, ...);
```

7 Strings

Creation

```
String s = "abc"; // String literal
String s = new String("abc"); // String object
char[] chars = { 'a', 'b', 'c' }; // char array
String s = new String(chars);  // creation from char array
```

Methods

Signature	Description
`int length()`	returns the length of this string
`String concat(String s)`	concatenates the specified string to the end of this string
`char charAt(int index)`	returns the `char` value at the specified index of this string
`int indexOf(char c)`	returns the index of the first occurrence of the given `char` ; –1 if the `char` does not occur
`int indexOf(char c, int fromIndex)`	returns the index of the first occurrence of the given `char` that is greater than or equal to `fromIndex` ; –1 if the `char` does not occur
`int lastIndexOf(char c)`	returns the index of the last occurrence of the given `char` ; –1 if the `char` does not occur
`int lastIndexOf(char c, int fromIndex)`	returns the index of the last occurrence of the given `char` that is less than or equal to `fromIndex` ; –1 if the `char` does not occur
`char[] toCharArray()`	converts this string to a new `char` array
`boolean equals(Object s)`	`true` if the given object represents a String equivalent to this string, `false` otherwise
`String toUpperCase()`	returns the String, converted to uppercase
`String toLowerCase()`	returns the String, converted to lowercase
`String[] split(String regex)`	returns the array of strings computed by splitting this string around matches of the given `regex`
`String substring(int index)`	returns the substring from the character at the specified index to the end of this string
`String substring(int start, int end)`	returns the substring from the character at the index `start` to character at the index `end-1`
`String replace(char oldChar, char newChar)`	returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`
`String trim()`	returns a string whose value is this string, with any leading and trailing whitespace removed

8 Type wrappers

Character

All the following methods are `static`

Signature	Description
`boolean isDigit(char c)`	`true` if the `char` is a digit; `false` otherwise.
`int getNumericValue(char c)`	returns the `int` value of the given `char`, –1 if the `char` has no numeric value.
`boolean isLetter(char c)`	`true` if the `char` is a letter; `false` otherwise.
`boolean isUpperCase(char c)`	`true` if the `char` is uppercase; `false` otherwise.
`boolean isLowerCase(char c)`	`true` if the `char` is lowercase; `false` otherwise.
`boolean isISOControl(char c)`	`true` if the `char` is ISO control (e.g., `\n`, `\t`, ...); `false` otherwise.
`boolean isWhitespace(char c)`	`true` if the `char` is a whitespace; `false` otherwise.
`char toUpperCase(char c)`	returns the uppercase equivalent of the `char`
`char toLowerCase(char c)`	returns the lowercase equivalent of the `char`
`String toString(char c)`	returns the String representation of the `char`

Integer

All fields and methods are similar in other `Numeric` subclasses

```
int a = Integer.MAX_VALUE;  // Largest value representable with an int
2147483647
int b = Integer.MIN_VALUE;  // -2147483648

static int parseInt(String s); //returns the integer value represented in the
given String
```

9 Classes and Objects

Class declaration

```
public class Triangle {
    // class variable (shared among all instances)
    static int triangleCount = 0;

    private int a,b,c; // instance fields (unique to each instance)
    public String color;

    Triangle(int a, int b, int c, String color){ // parametrized constructor
        this.a = a;      // 'this' refers to the current object instance
        this.b = b;
        this.c = c;
        this.color = color;
        triangleCount++; // increments the shared counter
    }
    public int getPerimeter(){ // instance method
        return a+b+c;
    }
    // Class (static) method: prints the number of Triangle instances created
    public static void printNumTriangles(){
        System.out.println("Number of Triangles: "+triangleCount);
    }
```

Object creation and usage

```
  public static void main(String[] args) {
        Triangle t1 = new Triangle(2,5,8,"red"); // object instantiation
        t1.color = "yellow"; // field value modification
        System.out.println( t1.color ); // field access

        int perimeter = t1.getPerimeter(); // method call
        Triangle.printNumTriangles(); // static method invocation
    }
```

10 Inheritance

Superclass

```
 public class Animal {
   String name;

   Animal(String name){
       this.name = name;
   }
   public void identify(){
       System.out.println("I'm an animal.");
   }
}
```

Abstract class

```
 public abstract class Pet extends Animal {

   Pet(String name){
       super(name); // (super) referring to the superclass constructor
   }
   public abstract void play();
}
```

Interface

```
public interface Trainable {
  void train(); // defines a trainable behavior
}
```

The Dog subclass

```
  public class Dog extends Pet implements Trainable {

    Dog(String name){
        super(name);
    }
    public void identify(){
        super.identify(); // call base version
        System.out.println("I'm a dog named " + name + ".");
    }
    public void play(){
        System.out.println(name + " plays fetch!");
    }
    public void train(){
        System.out.println(name + " learns to sit.");
    }
}
```

11 Recursion

Fibonacci

Definition:

$F_0 = 1$ and $F_1 = 1$ base cases

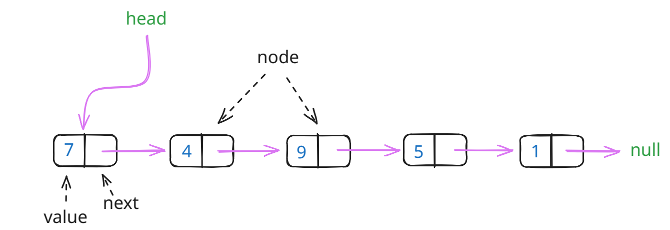
$F_n = F_{n-1} + F_{n-2}$ recursive step

Java implementation:

```
public static int fib(int n){
  if(n == 0) return 1; //base case
  if(n == 1) return 1; //base case
  return fib(n-1) + fib(n-2); // recursive call
}
```

12 Recursive Data Structures

Linked List - default implementation



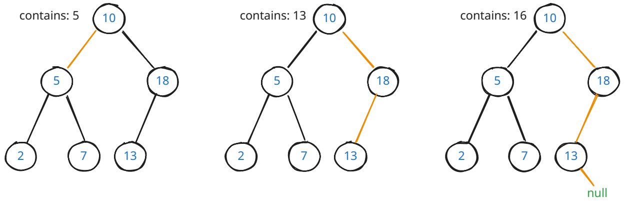
Node - pseudocode

```
class Node {
    int value    # value
    Node next    # pointer to next node

    Node(int value){  # constructor
        this.value = value
    }
    # default implmentation the new node is inserted at the end of the LL
    void insert(int value){
        if next is empty -> next = new Node(value)  # end LL -> insert node
        next.insert(value)              # propagate the insertion
    }

    # check if the value is contained in the LL
    boolean contains(int value){
        if this.value == value -> return true    # value found!
        if next is empty -> return false         # end LL -> value not found
        return next.contains(value);             # continue the search
    }
    # delete an element of the LL -> check next, bypass the target node
    boolean delete(int value){ . . . }
}
```

Binary Search Tree



BST - pseudocode

```
class Node {

    int key      # the node value
    Node left    # the left child
    Node right   # the right child

    boolean contains(int value){
        if value < this.key {  # value smaller than the current node
            if left is empty {
                return false   # end of BST
            }else{
                return left.contains(value) # search in the left subtree
            }
        }else if value > this.key {  # value larger than the current node
            if right is empty {
                return false   # end of the BST
            }else{
                return right.contains(value)  # search in the right subtree
            }
        }else{
            return true  # value match, element found!
        }
    }
    # insert: compare value → go left or right → create new node at leaf
    void insert(int value){ . . . }

  }
```

13 Exceptions

Exception example - ArrayIndexOutOfBoundsException

```
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80};
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter an index (0-7): ");
        try {
            int index = sc.nextInt();
            System.out.println("Index: "+index+" value: " + numbers[index]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid index: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Invalid input: " + e.getMessage());
        } finally {
            sc.close();
        }
    }
```

some RuntimeExceptions

- NullPointerException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException
- ArithmeticException
- ClassCastException
- NumberFormatException
- InputMismatchException