Java

Introduction to Java technology

Overview

- Simple language
    - syntax derived from C++
    - no operator overloading and other complex constructs
    - garbage collector
- OOP
    - both OOP and primitives
- Interpreted
    - code compiled by javac to bytecode, interpreted by JVM
    - Program.java -> javac -> Program.class
    - JVM contained inside JRE
    - usually requires bunch of libraries and dependencies present at both compile and run time
    - JVM is superfast, includes JIT and other advanced optimization mechanisms
    - bytecode check can be disabled tho
- Architecture-Independent
    - no pointers, doesn't matter if it runs on 32-bit or 64-bit architecture
    - data types have strictly given sizes
    - numeric types are always signed
- Platform-Independent
    - Platform - where we run the program, could be OS
    - bytecode is independent, runs as long as JVM is present (in JRE)
- Portable
    - Bytecode is portable, no need for recompilation
    - If there is JRE, it runs
- Secure

    - strong type checks, array range controls
    - automatic memory management, no pointers
    - JVM sandbox - restricted access to OS, network etc.
    - secured bytecode, classloaders, bytecode verifiers
    - security manager - manages sources for each class, such as access to disk
- Robust
    - strong type checks
    - garbage collector
    - exception handling during runtime
    - errors caught in JVM, won't affect OS
- High-Performance
    - bytecode is highly optimized
    - JVM is highly optimized, JIT
    - multithreading, parallel programming
- Multithreaded
    - a thread is a limited process that shares the code and memory and runs in parallel
- Distributed
    - Java program can be split into multiple devices that can communicate
    - distributed programming is useful for super large applications
    - RMI (Remote Method Invocation), EJB (Enterprise JavaBeans)
    - libraries for TCP/IP communication, HTTP, FTP, ...
    - supports CI/CD
- Dynamic
    - OOP, inheritance, ...
    - reflection - objects have information about themselves during runtime
    - native methods - supports methods from other languages like C/C++, dynamically connects them during runtime
    - Java Community Process (JCP) - for companies to contribute to improvement of Java itself
    - Java Specification Request (JSR) - requests to improve Java itself

Best choice for Microservices

- Robust - strongly typed, ...
- Java can be serverless - FaaS (function as a service), ...
- The ecosystem - Jakarta EE, Spring Boot, ...
- Scalable - asynchronous code (Java Futures, Reactive streams), vertical and horizontal
- JVM - combine multiple languages (Kotlin, Scala, Groovy, Java, ...)
- Tools
    - Maven, Gradle - manage dependencies, libraries
    - Docker, Kubernetes - manage microservices, containers, orchestration
    - CI/CD - GitLab CI, Jenkins... - automation
    - Monitoring, Logging - Elasticsearch, Kibana, Logstash
    - Apache Kafka - for asynchronous communication between services
    - Spring Boot, Spring Cloud - fullstack dev for microservices

Usages

- 90% of Corporates, Banking systems, Telecommunication, Enterprise, ...
- LinkedIn - uses Spring Boot
- Netflix
- Elasticsearch - high-performance distributed real-time search engine
- Android - Kotlin, Java
- Cloud computing - Java is in TOP 3

Challenges

- Performance - interpreted is still interpreted, large overhead
- Memory - Java is memory-heavy, resource-hungry
- Syntax - long syntax, not very comfortable
- Development speed
- Best alternatives - Rust, Go, Kotlin, Scala

Quick history

- 1990 - Oak language, for embedded systems
- 1995 - Java language, presented by Sun Microsystems, for web development

- 1996 - Released first Java Development Kit (JDK)
- 2004 - Java 5
- 2013 - Java 7
- 2014 - Java 8 - LTS
- 2018 - Java 10, Java 11 - LTS
- 2021 - Java 17 - LTS
- 2023 - Java 21 - LTS

Editions

- Java Card - for smart cards, terminals
- Java 2 Micro Edition (J2ME) - for embedded, set-top box, ...
- Java Standard Edition (JSE) - desktop applications
- Java Enterprise Edition (JEE) - corporate applications

Architecture

- JRE - Java Runtime Environment
    - includes JVM and APIs (libraries)
    - for running the programs
- JDK - Java Development Kit
    - bunch of tools for Java development
    - includes JRE, javac, debugger, javadoc, tools for making jars etc...

IDEs

- NetBeans - open source, Oracle
- Eclipse - open source, IBM
- IDEA - proprietary, JetBrains

The basics

Hello World

```
// compile
javac Program.java
```

```
// run
java Program

// Hello, World!
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Data types

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

- double uses norm IEEE 754, 64 bits
- char uses UTF-16 unicode

Other

```
// constans
final int MAX = 100;

// implicit type conversion
byte -> short -> int -> long -> float -> double
```

```
char -> int -> long -> float -> double

// explicit type conversion
int i = (int)d;
```

OOP

The purpose of OOP is to provide a solid foundation for designing and building scalable, modular, maintainable software systems.

The 4 pillars of OOP

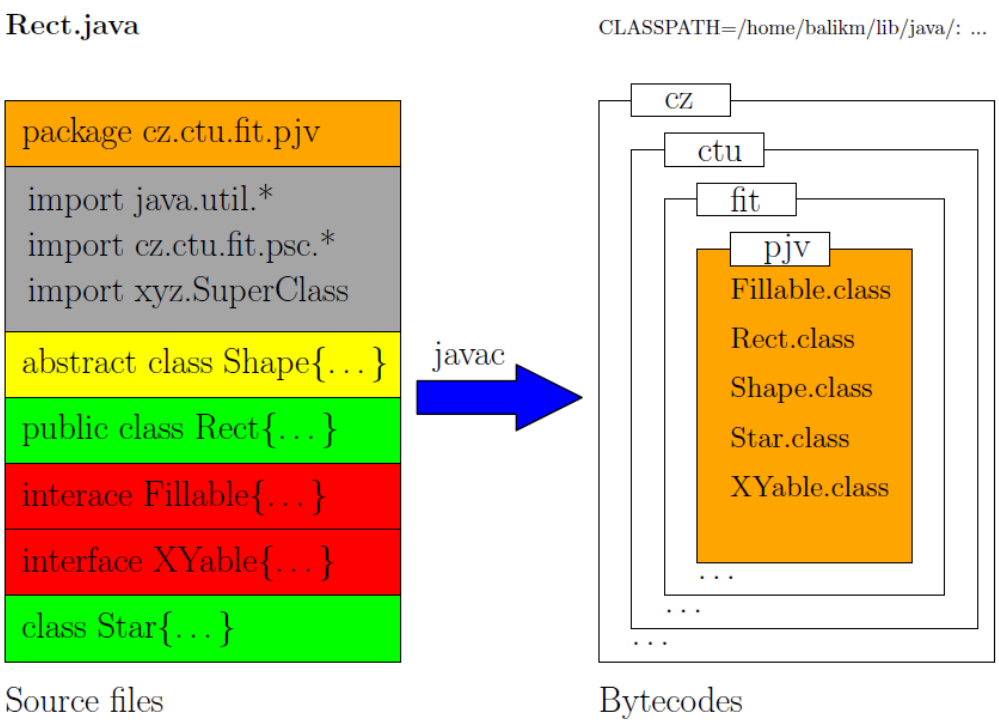
- Encapsulation
    - hide internal logic and properties (the details)
    - publish what other need to use you
    - reduces complexity, enhances security, promotes modularity
    - public - accessible by everything
    - protected - accessible by the class, package, subclass
    - nothing - accessible by the class, package
    - private - accessible by the class
- Abstraction
    - simplifying the complexity by modeling classes based on properties and behavior
    - ignore non-essential details
    - helps managing complexity by focusing on the essential features
    - defines abstract classes and interfaces
    - working with high-level
- Inheritance
    - allow a new class to inherit properties and behavior from another class
    - promotes reusability
    - establishes hierarchical relationships between classes
- Polymorphism (many forms)
    - the ability of objects to take on multiple forms

    - allows different objects to be treated as objects of their common base class
    - enables flexibility and extensibility
    - allows single interface to be used for different objects

Objects

- the real-world entities can be hierarchized into classes (concepts)
- a class is characterized by attributes (data) and behavior (methods)
- object is an instance of a class, class serves as a skeleton for objects
- static attributes and static methods = class properties/methods, not instance's
    - static methods can only access static attributes
    - can technically be called from instance but is **not recommended**
- java runs a static method Program.main() first
    - a JAR can contain multiple classes with **main** method, manifest.xml decides which one to run
- attributes
    - are accessible from **class** and **package** by default
    - **protected** attributes/methods are accessible from **class**, **package** and **subclass**
    - **public** attributes/methods are accessible from everywhere
    - **private** attributes/methods are accessible only from **class**
    - attributes are assigned null value by default (0, null or false based on type)
- constructor
    - is called, when object is created with **new**
    - can be overloaded
    - a private constructor ensures that object can only be instantiated inside its class
        - singleton, builder, uninstantiable classes (static class)
    - constructor can be called inside another constructor with **this()**
- GC is automatic but we can run it ourselves - `System.gc();`
- a Class can also just be a function library (such as java.lang.Math class)
- **Enum** is a special class
- **Record** (Java 14+) - immutable object with only attributes and name (data object)

    - final class with included getters, public constructor and equals/toString/hashCode methods
    - classes can extend Records
- **Interface** only defines the method structures
    - can have multiple interface inheritance
    - is not final
    - can contain constants (public static attributes)
- **Abstract class** defines method structures and may define their behavior as well
    - can't be final
    - has at least 1 constructor
    - can use static attributes/methods
    - abstract methods contain no implementation
- packages
    - the main package is java.lang (no Java without this one)
    - `import java.lang.* // is default`
    - order of import is not important



- source file can only have 1 top-level public class/interface
    - source file name should correspond to the class/interface name

    - each class is translated into bytecode .class file
    - JVM accepts them only if they are inside those folders that correspond to their package
    - system variable CLASSPATH must link the directory (or JAR) with the package folder
- **final** - no more modifications, no more inheritance
- Multiple references to the same object
    - Obj obj1 = new Obj();
      Obj obj2 = new Obj();
      obj1 = obj2;
      System.gc(); // optional
- Anonymous class
    - no header, no name, no inheritance
    - for creating a single object
    - inherits directly from a non-final class
    -
```
Runnable ticTak = new Runnable() {
  @Override
  public void run() {
    System.out.println("Tic");
    try{ Thread.sleep(1000);} catch(...){...}
    System.out.println("Tac");
  }
}
```

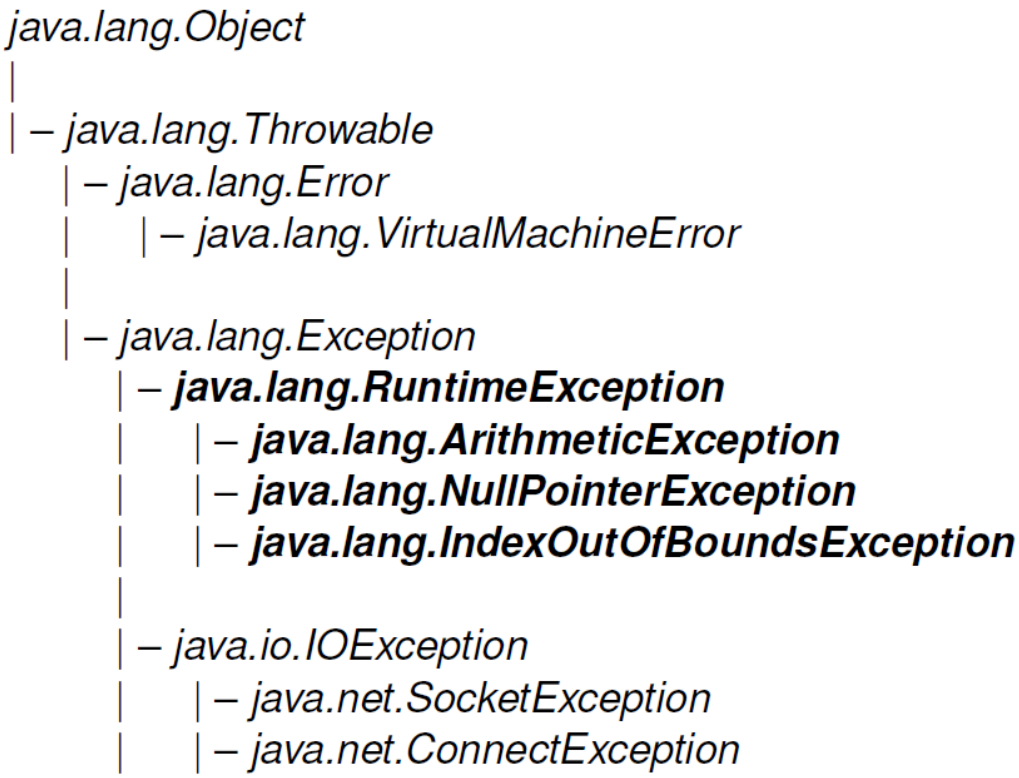
```
/** Record **/
public record Person(String name) {}

/** Class example **/
public class Rectangle {
    private static final int MAX = 10;
    private Color color;
    private int width, height;
    public Rectangle() {}
    public Rectangle(int width, int height) { this(); }
    public int getPerimeter() { return 2*(width+height); }
    public int getArea() { return width*height; }
    public void setColor(Color c) { this.color = c; }
```

```
    public static void sayRectangle() { System.out.println("Rectangle"); }
}

Rectangle rec = new Rectangle();
rec.getArea();
Rectangle.sayRectangle();

/** Interface **/
public interface IVisitable {}

/** Abstract class **/
public abstract class AbstractClass implements IVisitable {}

/** Enum **/
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
Level lowLevel = Level.Low;

/** static attributes initialization **/
class Config {
    static int j = 10;
    static {
        String val = System.getProperty("key1");
        int val1 = Integer.parseInt(System.getProperty("key2"));
    }
}
// java -Dkey1=10 -Dkey2=14 Config
```

OOP basics

- prefer **composition** over **inheritance**
- all objects inherit from java.lang.Object (defines 11 general methods)
    - toString() - returns text representation of the object in String
        - getClass().getName() + “@” + Integer.toHexString(hashCode());
    - equals() - return (this == obj), doesn’t compare attributes
    - hashCode() - returns hash of the class data
    - clone()

    - getClass()
    - finalize()
    - notify(), notifyAll()
    - wait()
- inheritance is transitive
- child classes can overshadow parent's attributes if they define their own
- OOP design - each class should perform / be responsible for one single well-defined thing
- Exceptions
    - Exceptions are objects, extends Throwable
    - RuntimeExceptions are not checked (no need to be handled or indicated in header)
        - programmer's errors, to be fixed later
        - ArithmeticException
    - Otherwise are checked (needs to be handled or indicated in header)
        - try-catch or stating "throws" in method header
        - FileNotFoundException
    - toString(), getMessage(), stack trace
    - prefer particular exception classes over generic ones
    - include meaningful messages when throwing an exception
    - never catch **throwable**, maximum is Exception or RuntimeException
        - it could catch OutOfMemoryError etc...
    - include the previous throw in the new throw, if you log the exception, don't throw it anymore



- Hierarchy
  - var Parent = Child (OK)
  - var Child = Parent (NOK)

```
/** Inheritance **/
class Animal {
    String x = "Animal";
    protected getX() { return x; }
}
class Dog extends Animal {
    String x = "Dog";
    @Override
    protected String getX() {
        return x + " and " + super.x;
    }
}

/** Composition **/
class Game {
    private Map map = new Map();
    private Player p = new Player();
```

```
}

/** Casting **/
Dog d = new Dog();
Animal a = d;
Animal a = new Dog();
Dog d = (Dog)a; // error, only if a was a Dog

/** Exception handling **/
public class MyException extends RuntimeException {}

public void doSomething() throws MyException {
    try {
        // ...
    } catch (RuntimeException e | MyException e1) {
        // ...
        throws new MyException();
    } finally {
        // clean up, even without exception
    }
}

SomeClass sc = new SomeClass();
try (
    sc;
    ResultSet rs = sc.doSomething();
 ) {
    // ... working with rs
} catch (FileNotFoundException | SomeOtherException e) {
    // ...
}

/** Accessing suppressed (previous) Exceptions **/
try(...) {
    // ...
} catch (NewException e) {
    Throwable[] suppressed = e.getSuppressed();
    // ...
}
```

Utils and Collections

java.lang.Math

- java.lang.Math
- static constants (PI, E, ...)
- static methods
- Math.sin(0.5)
- sin, cos, tan
- abs, min, max, log, sqrt, pow
- random
- ...

Classic Array

- contains either primitives or references
- elements are initialized to 0, null or false
- arrays can be referenced with java.lang.Object

```
int[] arr = new int[100];
Object o = arr;
int len = o.length;
int[] arr2 = {1, 2, 3};
Object[] arr3 = new Point[] {
    new Point(1, 2), new Point(3, 4)
};
int[][] arr2d = new int[10][5];
int[][] arr2d2 = { {1}, {2, 3}, {4, 5, 6} };
```

java.util.Arrays

- utility with a bunch of static methods to work with arrays
- `asList()` - turn array into List collection
- `toString()` - print array
- `equals(), hashCode()`
- `fill()` - fill all elements with given value

- `binarySearch()`
- `sort()`
- `copyOf()`, `copyOfRange()` - copy elements
- `deepEquals()`, `deepHashCode()`, `deepToString()`

```
int[] arr = {8, 2, 4, 1, 3};
Arrays.sort(arr, (e1, e2) -> Integer.compare(e1, e2));
Arrays.binarySearch(arr, 3);
```

Genericity

- We can put generic Object into collection and then cast it back when we read it (runtime errors if wrong type)
- Or we can define the type of objects in the collection using <Type>

ArrayList arr1 = new ArrayList();

ArrayList<Person> arr2 = new ArrayList<Person>(); // java 5+

ArrayList<Person> arr3 = new ArrayList<>(); // diamond operator java 7+

Custom generics

```
public class Interval<E extends Comparable<? super E>> {
    private E low, high;
    public Interval(E low, E high) {}
    @Override
    public String toString() { return ""; }
    public boolean contains(E x) {}
}
...
Interval<String> s = new Interval<>("prague", "brno");
Interval<Integer> i = new Interval<>(1, 20);
```

Collections (containers)

- Objects for storing elements of the same type
- Bunch of Collection algorithms for processing and manipulation
- Slower than Array
- Simplicity of programming, readability, fast algorithms, ...

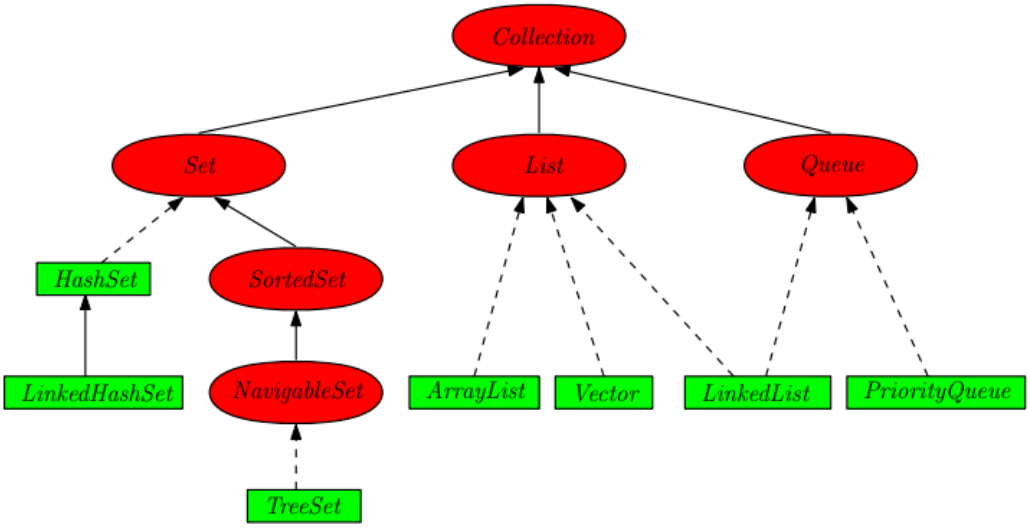
java.lang.Iterable<T>

- interface for iterable objects
- can do forEach() and return an iterator<T> or SplitIterator<T> for parallel processing
- Collections extend this interface

```
ArrayList<Integer> arr = new ArrayList<>(); // extends Iterable
for (Integer e: arr) { print(e); }
arr.forEach(e -> print(e));
```

java.lang.Collection<E> extends Iterable<E>

- interface with bunch of methods
    - `boolean add/remove(E)`
    - `boolean add/remove(Collection<? extends E>)`
    - `boolean isEmpty()`
    - `boolean removeIf(Predicate<? super E>)`
    - `void clear()`
    - `boolean contains(Object)`
    - `boolean containsAll(Collection<?>)`
    - `default Stream<E> stream()`
    - `boolean retainAll(Collection<?> c)` - removes all but c
    - `int size()`
    - `Object[] toArray()`
    - `<T> T[] toArray(T[])`
    - ...
- Basic collections that extends Collection
    - Set
        - unique elements, may be sorted
    - List
        - elements has indexes
    - Queue, Deque
        - bidirectional linked FIFO, LIFO



- Specific Collection classes
  - ArrayList<E>
    - bouncy array, amortized linear complexity (like Vector in C++)
    - get, isEmpty, size, set, add, iterator, listIterator - constant complexity
  - LinkedList<E>
    - bidirectional linked list
    - linear complexity
  - interface Set<E>
    - unique elements (e1.equals(e2))
    - need to correctly define **equals()**
    - only one **null** element
  - HashSet<E>
    - Order is not guaranteed
    - need to correctly define **hashCode()**
    - add, remove, contains, size - constant complexity (if hashCode() is good)
    - can have null elements
    - uses balanced tree for large amount of colliding keys
    - HashMap
      - computes HashCode
      - less collisions
      - pretty expensive computation

        - **Equals and HashCode**
            - Equals(a, b) => HashCode(a) == HashCode(b)
            - not the other way around
    - Map<K, V>
        - key-value collection
        - HashMap, LinkedHashMap, TreeMap, HashTable

java.util.Collections<E>

- same as java.util.Arrays for Arrays
- contains a bunch of static methods for collection manipulation
    - <T> int binarySearch()
    - <T> void copy(List<? super T> dest, ... src)
    - <T> Set emptySet()
    - <T> void fill(...)
    - <T> T max/min(Collection)
    - void reverse, rotate, shuffle (List)
    - <T extends Comparable<? super T>> sort(List<?>)
    - <T> void sort(List, Comparator)
    - <T> Set<T> synchronizedSet(Set)
    - ...

Iterators

- Iterator pattern for iterating through Collections
- Iterator<E>
    - void remove()
    - default void forEachRemaining(Consumer<? super E>)
    - boolean hasNext()
    - E next()
    - while (hasNext()) doSomething(next());
- ListIterator<E> extends Iterator<E>
    - void add(E), set(E)
    - boolean hasPrevious()

- E previous()
- int nextIndex(), previousIndex()

Comparators

- class that implements Comparator with method **compare(Object, Object)**
- each class can have 1 natural sorting way and multiple comparators
- we need equals() for searching through collection
- we need compare(), compareTo() for binarySearch and sorting
- we can sort naturally - class must implement Comparable<T> and its methods (compareTo())
- Collections.sort() uses modified QuickSort

```
Collections.sort(List<?> list, Comparator<? super T> comp);
class Product implements Comparable<Product> {
    public int compareTo(Product o) {
        return price - o.price; // or Integer.compare(price, o.price);
    }
}
...
List<Product> listOfProducts = new ArrayList<>();
Collections.addAll(listOfProducts, new Product...);
Collections.sort(listOfProducts); // natural
Collections.sort(listOfProducts, new SomeComparator()); // using comparator
```

Collection synchronization

- only for multithreaded applications
- a thread must not access an element of another thread
- collections defaultly are not synchronized - faster

```
List list = Collections.synchronizedList(new LinkedList(...));
```

Input/Output

IO vs. NIO

- IO - blocking I/O, stream
- NIO - non-blocking I/O, uses Buffer, programmers checks for completion
    - Selector - allows for a thread to manipulate multiple channels
- NIO.2 (Java7+) - Stream API, Path, Files, functional programming

Files vs. Streams

- file - data stored in external memory
- streams - utility for data transportation between files, network, memory, other programs, ...
- 3 phases
    - stream for bytes
    - stream for Java data types
    - filtration, buffering, ...
- Text streams
    - Reader (OS -> Java) and Writer (Java -> OS) must convert chars between OS and Java
- **Reader** class
    - throws IOException
    - close() - closes the stream
    - int read() - reads next character, returns -1 if finished
    - int read(char[] c, int offset, int len) - reads sequence of characters into array
    - boolean ready() - is stream ready for read
    - reset() - resets the stream to position given by mark()
    - skip(long n) - skips n characters
    - mak(int readAheadLimit) - marks position in stream
    - Reader(Object lock) - for synchronization of critical sections of the given object
- **Writer** class
    - close(), flush(), write(), append()
- **InputStream** class
    - int available() - number of available bytes
    - close, read, reset, skip, mark, ...

- **OutputStream** class
    - close(), flush(), write()
- use buffers for more effectivity

```
Reader in = new BufferedReader(new InputStreamReader(System.in));
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

- **DataInput DataOutput** interface
    - define methods for read/write primitives into stream
    - boolean readBoolean(), char readChar(), ...
- **File** class
    - represents file or directory, not the data
    - createNewFile(), mkdir(), getAbsolutePath(), isDirectory(), isFile(), exists(), canRead(), canWrite(), delete(), renameTo(), lastModified(), ...
    - Java7+ better to use **Path, Paths** and **Files** classes instead

```
Files.list(Paths.get("."))
    .map(path -> path.toAbsolutePath())
    .forEach(System.out::println);
```

- **FileSystem** factory
    - getPath(), getPathMatcher(), getFileStores(), ...
    - newWatchService() - WatchService observes changes to the object
    - FileStore - getTotalSpace(), getUnallocatedSpace(), getUsableSpace(), ...

File copying

- by bytes

```
try {
      String path = System.getProperty("user.home");
      File inFile = new File(path + "/in.txt");
      File outFile = new File(path + "/out.txt");
      try (
            FileInputStream fis = new FileInputStream(inFile);
            FileOutputStream fos = new FileOutputStream(outFile, true);
      ) {
            int c;
            while ((c = fis.read()) != -1) {
                  fos.write(c);
```

```
            }
      }
} catch (IOException ex) {...}
```

- by lines

```
BufferedReader br = null;
BufferedWriter bw = null;
try {
      br = new BufferedReader(
            new InputStreamReader(
                  new FileInputStream("C:\\in.txt")));

      bw = new BufferedWriter(
            new OutputStreamWriter(
                  new FileOutputStream("C:\\out.txt")));

      String s = null;
      while ((s = br.readLine()) != null) {
            bw.write(s);
            bw.newLine();
      }
} catch (IOException ex) {
      // ...
} finally {
      if (br != null) br.close();
      if (bw != null) bw.close();
}
```

Serialization

- Serializable - object can be transferred through a stream
- Serialize object attributes into bytes
- We can store serialized object into a file for example
- Deserialization
    - doesn't call constructor of the serializable class
    - calls constructor of the parent that isn't serializable
- serialVersionUID - is compared with the class value during deserialization (validation)

- can be set manually

```
FileOutputStream fileOutputStream = new FileOutputStream("someObject");
ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);
os.writeObject(someObject);
os.close();

FileInputStream fileInputStream = new FileInputStream("someObject");
ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
SomeObject so = (SomeObject) objectInputStream.readObject();
```

Compression

- **Deflater** class
  - creates standard zip, gzip or jar
- **Inflater** class
  - decompresses standard zip, gzip or jar

```
try {
      String inputString = "blablabla";
      byte[] input = inputString.getBytes("UTF-8");
      byte[] output = new byte[100];

      Deflater compresser = new Deflater();
      compresset.setInput(input);
      compresser.finish();
      int compressedLength = compresser.deflate(output);
      compresset.end();
} catch (UnsupportedEncodingException | DataFormatException ex) {...}

try {
      Inflater decompresser = new Inflater();
      decompresset.setInput(output, 0, compressedLength);
      byte[] result = new byte[100];
      int resultLength = decompresser.inflate(result);

      String outputString = new String(result, 0, resultLength, "UTF-8");
} catch (UnsupportedEncodingException | DataFormatException ex) {...}
```

Collator

- for sorting using locales

```
final Collator collator = Collator.getInstance(new Locale("cs", "CZ"));
Collections.sort(czechList, (s1, s2) -> collator.compare(s1, s2));
```

Java11 read/write

```
String words = "abc\ndef\nhij";
Files.writeString(Path.of("words.txt"), words);

String data = Files.readString(Path.of("words.txt"));
```

Java 8 - Functional interface, Lambda, Stream API

Lambda

- (parameters) -> function

```
/** Standard way **/
class AbsCompatator implements Comparator<Integer> {
      public int compare(Integer o1, Integer o2) {
            return Math.abs(o1) - Math.abs(o2);
      }
}
Arrays.sort(arr, new AbsCompatator());


/** Lambda **/
Arrays.sort(arr, (o1, o2) -> Math.abs(o1) - Math.abs(o2));


/** Named Lambda **/
Comparator<Integer> absComparator = (o1, o2) -> Math.abs(o1) -
```

```
Math.abs(o2);
Arrays.sort(arr, absComparator);
```

Functionals

```
@FunctionalInterface
public interface MyFunction {
      public int calculate(int input);
}

MyFunction power = (int i) -> { return i * i; }

MyFunction power2 = (i) -> i * i;

power.calculate(25);
```

Functional References:

```
Supplier<Long> supplier = () -> System::currentTimeMillis;
long time = supplier.get();
```

Constructor References:

```
Function<Long, Date> dateGenerator = Date::new;
Date date = dateGenerator.apply(System.currentTimeMillis());
```

Custom functions:

```
Function<Integer, Integer> plus1 = a -> a + 1;
Function<Integer, Integer> times2 = a -> a * 2;

int sum = plus1.apply(5);

// (1*2)+1
int result = plus1.compose(times2).apply(1);
// (1+1)*2
int result = plus1.andThen(times2).apply(1);
```

Predicates:

- returns boolean

- like a stored condition

```
Predicate<String> notNull, validLength, hasNumber, hasCapital;

notNull = Objects::nonNull;
validLength = s -> s.matches(".{8,32}");
hasNumber = s -> s.matches(".*[0-9].*");
hasCapital = s -> s.matches(".*[A-Z].*");

Predicate<String> composedPredicate =
      notNull.and(validLength).and(hasNumber.or(hasCapital));

boolean test = composedPredicate.test("testString55");
```

Stream API

- stream() or parallelStream()
- Stream.of(Object[])
- IntStream range(int, int)
- Random.ints()

```
Stream<String> myStream = names.stream();
Stream<String> myStream = Stream.of("a", "b", ...);

Stream.iterate(21, i -> i + 3) // from 21
      .filter(i -> i % 2 != 0)
      .limit(5)
      .forEach(System.out::println);

IntStream.range(3, 8).forEach(System.out::println);
```

- sorted(Comparator) - sort stream
- map(Function) - transform objects
- filter(Predicate) - filter
- flatMap() - same as map but flattens the stream so that its a top level stream (no substreams)
- peek(), limit(), skip(), ...

- forEach, count, sum, max, min, toArray, reduce, findFirst, findAny, anyMatch, noneMatch, allMatch, ...
- Storing result
    - stream().collect(Collectors.toList());
    - stream().collect(Collectors.joining(“, ”));
    - ...

```
Map<Department, Integer> totalByDept = employees
      .stream()
      .collect(Colelctors.groupingBy(
            Employee::getDept,
            Collectors.summingInt(Employee::getSalary)
      ));

Map<Boolean, List<Student>> passingFailing = students
      .stream()
      .collect(Collectors.partitioningBy(
            s -> s.getGrade() >= 50
      ));
```

- Parallel streams
    - parallelStream().filter(...).forEach(...)

Optional

- container that stores either result or nothing
- usable after findFirst(), findAny(), ...
- isPresent()
- get()

```
String expected = "hello";
Optional<String> value = Optional.of(expected);
Optional<String> defaultValue = Optional.of("default");

Optional<String> result = value.or(() -> defaultValue);
```

Functional interface

- interface with only 1 abstract method
- Comparable, Runnable
- java.util.function
- Consumer - don't return anything
    - Consumer void accept(T value)
    - BiConsumer(T, U) void accept(T t, U value)
    - ...
- Supplier - has no parameters, return given type
    - Supplier T get()
    - Function<T, R> R apply(T value)
    - ...
- Functions
    - ToIntFunction int applyAsInt(T value)
    - BiFunction<T, U, R> R apply (T t, U u)
    - ...
- Operators
    - UnaryOperator T apply (T operand) - return same type as parameter
    - BinaryOperator T apply(T left, T right) - 2 params, same type, return same type as params

StringJoiner

- Java 8+
- StringJoiner(delimiter, prefix, suffix)

```
StringJoiner sj = new StringJoiner(":", "[", "]");
sj.add("a").add("b").add("c");
String result = sj.toString(); // [a:b:c]

String result = Stream.of("a", "b", "c")
    .collect(Collectors.joining(":", "[", "]"));
```

Multithreading