Java Class Design

Encapsulation

- Access modifiers enforce effective encapsulation
- Combining data/functions operating as single unit

Access Modifiers

- public (class)
- protected
- default (class)
- private

Access modifiers/ accessibility	Within the same class	Subclass inside the package	Subclass outside the package	Other class inside the package	Other class outside the package
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	No	Yes	No
Private	Yes	No	No	No	No

```
class C { public void x() {} } → package-protected
```

Inheritance

- **IS-A** relationship
- reusability mechanism in OOP
- hierarchical relationships (easy to understand)

Polymorphism

- "several forms" of an entity
- Interpreting same message/method with different meanings depending on the context

Dynamic / runtime polymorphism

- overriding methods -> extending functionality
- nonstatic and nonfinal method
- abstract methods
- late binding
- dynamic method resolution/invocation

Static / compile-time polymorphism

- function overloading

```
c1.fillColor(0, 255, 255);
c2.fillColor(0.5f, 0.5f, 1.0f);
```

- constructor overloading
- **overload resolution** - closest match by using upcasts:

```
byte b=10; person.m(b);
1. void m(short x) { }
2. void m(int x) { }
3. void m(Integer x) { }
4. void m(Number x) { }
5. void m(Object x) { }
```

```
public static void aMethod (byte val );
… aMetyhod(9); -> CE
```

Overriding Methods in Object Class

@Override

- Help to find bugs on compile time!

```
@Override
public boolean equals(Point other) -> CE
```

- **Object:** clone(), equals(), hashCode(), toString() and finalize()
- **Can't final**: getClass(), notify(), notifyAll(), wait()
- In HashSet/HashMap used: equals(), hashCode()

```
public String toString();
protected String toString(); -> fail
public Object toString() -> fail
public String ToString() -> different
```

Override rules

- names should exactly match
- same argument list types (or compatible types)
- return type can be a subclass-covariant
- should not throw new or broader checked exceptions (fewer or narrower checked exceptions, or any unchecked Exception)

Covariant return type example:

```
abstract class Shape {
    public abstract Shape copy();
}
class Circle extends Shape {
    public Circle copy() { /* return a copy */}
}
Circle c1 = new Circle();
Circle c2 = c1.copy(); -> ok

Shape s = new Circle();
Circle c3 = s.copy(); -> CE
```

```
public boolean equals(Object obj);
@Override
public boolean equals(Object other) {
  if(other == null)
    return false;
  else if(this == other)
    return true;
  if(other instanceof Point) {
    Point p2 = (Point) other;
    if((xPos == p2.xPos) && (yPos == p2.yPos))
    return true;
  } else
    return false;
}
```

```
public int hashCode();
```

- hashCode() method should return the **same hash** value for two objects if the **equals() method returns true** for them

```
class Circle {
    public boolean equals(Object arg) { … }
    public int hashCode();
}
Set<Circle> circleList = new HashSet< >();
circleList.add(new Circle(10, 20, 5));
circleList.contains(new Circle(10, 20, 5)); ->true

---
public int hashCode() {
    return center.hashCode() ^ radius;
}
---
public int hashCode() {
  long bits = Double.doubleToLongBits(getX());
  bits ^= Double.doubleToLongBits(getY()) * 31;
  return (((int) bits) ^ ((int) (bits >> 32)));
}
```

Bitwise Operators

```
a    = 0011 1100
b    = 0000 1101
---
a&b  = 0000 1100 (AND)
a|b  = 0011 1101 (OR)
a^b  = 0011 0001 (XOR)
~a   = 1100 0011 (compliment/ flipping)
---
a<<2 = 1111 0000 (left shift)
a>>2 =      1111 (right shift)
a>>>2 = 0000 1111 (right shift, fill 0)
```

Composition

- HAS-A relationships
- composite object that is made up of other smaller objects
- Computer (CPU+RAM+HDD+…)

Example:

```
class Point {…}
class Circle { private Point point; … }
```

Composition vs. Inheritance

- nothing is a silver bullet
- depends on design
- favor composition over inheritance

Composition:

A computer HAS-A CPU
A circle HAS-A point

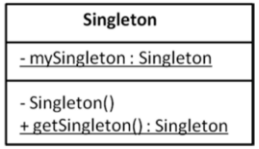
Inheritance:

A circle IS-A shape
A laptop IS-A computer
A vector IS-A list

Singleton

- only one instance created for class
- global single point of access to that object
- private constructor
- complex task in multi-threaded environment

Singleton Example 1




```
public static synchronized Singleton getSingleton(){
  if(myInstance == null)
    myInstance = new Singleton();
  return myInstance;
}
```

Singleton Example 2

Singletons for multi-threaded applications.
Initialization on demand holder

```
public class Logger {
        private Logger() {
                // private constructor
        }
        public static class LoggerHolder {
                public static Logger logger = new Logger();
        }
        public static Logger getInstance() {
                return LoggerHolder.logger;
        }
        public void log(String s) {
                // log implementation
                System.err.println(s);
        }
}
```

Immutable Classes

- once initialized, cannot be modified
- can't modify state of object
- Example: Wrapper classes

Pros:

- safier to use, rely on unchanged content
- thread-safe
- save memory/space str.intern()

Cons:

- modification slow (create new copy)

> ***Effective Java:*** *"Classes should be immutable unless there's a very good reason to make them mutable… If a class cannot be made immutable, you should still limit its mutability as much as possible"*

Definition Rules:

- final fields (initialize in constructor)
- provide getters/accessors (all fields private)
- if reference types that are mutable (List..)
  - don't change content
  - don't provide references to outside
  - deep copy of the object (return new)
- final class (avoid extension/modification)

Static

- class variables
- static methods
- static initializers (when load class into mem)
- suitable for utility methods
- calling slightly efficient (static vs instance)

Can't:

- Can't **access instance** variables/methods
- Can't **override static method** of base class
- Can't use **this/super** keywords

Advanced Class Design

Abstract Classes

- abstraction with some common functionality
- can't create new instance
- abstract methods without body
- abstract keyword: class, non-static methods
- **not allowed:** static methods

Final

Final Classes
- cannot extend / non-inheritable class
- assured the unchanged behavior (Wrappers)
- improved performance (methods resolved on compile)
- String, System, Number, Integer, ...
- public abstract final class Shape { } -> CE

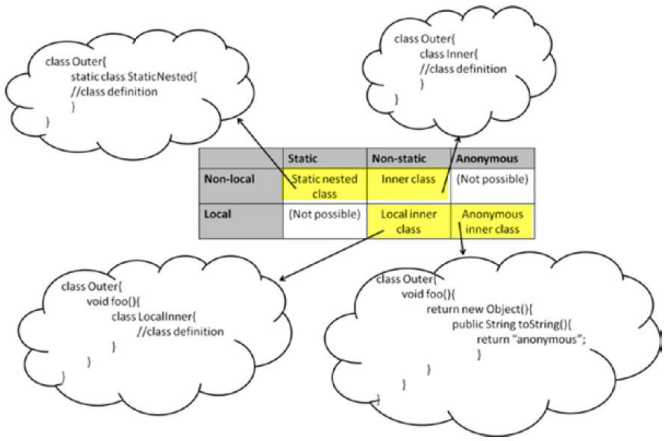
Final Methods
- cannot override
- all methods imlicitly final in final class

Final Variables
- class, instance or parameter
- constant (can be assigned only once)
- must be initialized in:
  initializer OR all constructors
- final/effectively final expected in labda, inner…

Nested/Innner Classes

- within body of another class/interface
- accessibility defined by outer class
- can be used as base classes
- can be final/abstract
- in interface members explicitly: public + static
- inner could have inner

Pros:
- put related classes as single logical group
- can access all members of the enclosing class
- simplify code



Static nested class/interface
- def: inside class (static)
- can have static members
- can access outer class static members
- can create objects of B without A.
  For example: A.B b = new A.B();
- instanceof don't care about outer instance

```
class Outer {
  static class Inner {}
  static interface Inner2 {} -> explicit ????
  interface Inner3 {} -> same
  static abstract class InnerX { }
  static final class InnerY { }
}
…
interface Outer {
  public static final VAR=1; -> explicit
  public static class Inner {} -> explicit
  public static interface Inner2 {} -> explicit
}
```

```
abstract class Abstract {
  static class InnerStatic {
        static int var=1;
        int nonStaticVar=2;
  }
}
...
Abstract.InnerStatic.var = 5;
Abstract.InnerStatic.nonStaticVar = 5; -> CE
Abstract.InnerStatic x = new
Abstract.InnerStatic();
     NOTE: similar to outer class
x.var = 1;
x.nonStaticVar = 7; -> ok
Abstract.InnerStatic.var = 2;
→ same value for all reference
```

```
static class Outer { } -> CE
```

Inner class/interface
- def: inside class (non-static)
- **can extend** any (outer/inner) non-static class!
- associated with instance of outer class
- ignore private acces
- Can't have static declarations (except **final constants**)

```
class Outer {
  class Inner {
    static int i = 10; -> CE
  }
  ...
  Inner inner = this.new Inner();
  Inner inner2 = new Inner(); -> CE
}

class Outer {
  interface Inner {}
}
NOTE: in interface: Explicitly STATIC! NOT INNER!
```

Local inner class / interface

- defined in code block (method, constructor, or initialization block)
- not members of an outer class
- not available outside block
- Can't: define static variable/class
- Interfaces cannot have local classes/interfaces
- Cannot create local interfaces
- Variables accessed by local inner classes are considered effectively final

```
class SomeClass {
  void someFunction() {
    class Local { }
    interface LocalInterface { } -> CE
  }
}
interface MyInterface {
  default void defMet() {
    class Local() { }; -> CE
  }
}

static Shape.Color getColDesc(Shape.Color color){
  class DescriptiveColor extends Shape.Color {
    public String toString() {
      return "RGB values" + color;
    }
  }
  color = null; -> CE effectively final!
  return new DescriptiveColor();
}
```

Anonymous inner class / interfaces

- synonym: anonymous classes
- no class name, just body
- characteristics: local inner class
- no constructor!
- instance-creation expression
  return new Object() { ... }
- cannot extend/implement

```
Car getDesc(final Car color) {
  return new Car () {
         public String toString() {
           return "RGB values" + color;
         }
       };
}
```

Enum Data Type

- typesafe solution for constants
- improves code readability
- explicitly: public, static, and **final**
- implicitly derived from: **java.lang.Enum**
- could have member attributes and methods
- private construcor only
- **can't**: extend
- **can't**: new / instantiate
- **can't**: implement inside method (only inside of top-level class or interface)
- Internally, enumerations converted to classes

```
enum PrinterType {A, B, C};
--
switch(x){ case DOTMATRIX: …
--
If (PrinterType.A == enumVar) …
--
enum CarType {A, B, C};
CarType.A != PrinterType.A
```

```
enum PrinterType {
  DOTMATRIX(5), INKJET(10), LASER(50);

  private int pagePrintCapacity;

  private PrinterType(int pagePrintCapacity) {
    this.pagePrintCapacity = pagePrintCapacity;
  }

  public int getPrintPageCapacity() {
    return pagePrintCapacity;
  }
}
-- "LASER": enum textual name
        PrinterType.LASER
        PrinterType.LASER.toString()
        PrinterType.LASER.name()
        PrinterType.valueOf("LASER")
        PrinterType.LASER.valueOf("LASER")

-- 50: getters
        PrinterType.LASER.getPrintPageCapacity()

-- 2: enum ordinal position [0..N]
        PrinterType.LASER.ordinal();
```

Interfaces

- set of abstract methods that defines protocol (contract for conduct)
- @Override annotation
- methods: abstract, default, static
- **all members** explicitly: public static final
- interfaces: public or default
- **can't**: use synchronized or final
- **can't**: override static methods
- **can't**: constructor
- class can implement N interfaces
- reference to interface can refer any class (implementing it)

Diamond problem

- same method signature twice in baseclass/interface
- abstract class method + interface default method

```
***
interface Int1 { default public void foo() {}}
interface Int2 { default public void foo() {}}
public class My implements Int1, Int2 {
  public void foo() {
    Int1.super.foo();
  }
}
***
interface Int1 { default public void foo() {}}
class Base { public void foo() {} }
class My extends Base implements Int { }
    -> OK (class wins, interface ignored)
***
```

Functional Interfaces

- SAM (Single Abstract Method)
- must have 1 abstract + 0…N from Object
- could have static/default methods
- @FunctionalInterface optional annotation

java.util.function.Predicate

```
boolean test(T t)
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> negate()
default Predicate<T> or(Predicate<? super T> other)
static <T> Predicate<T> isEqual(Object targetRef)
```

```
@FunctionalInterface
class My Class() { } -> CE only for interfaces
```

```
Interface Int0 {
   void test(boolean val);
}
@FunctionalInterface
Interface Int1 extends Int0 {
    void run(); -> CE only 1 abstract expected
}
```

---

must have 1 abstract + 0…N from Object

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);     -> OK: abstract
    boolean equals(Object obj); -> OK: from Object
}
***
@FunctionalInterface
public interface Comparator<T> {
    boolean equals(Object obj); -> CE 0 abstract
}
***
public interface Comparator<T> {
    boolean equals(Object obj); -> OK: no annotation
}
```

---

default -> abstract is OK:

```
interface DoNothing {
  default void doNothing() { }
}

@FunctionalInterface
interface DontDoAnything extends DoNothing {
  @Override
  abstract void doNothing(); -> OK
}
```

Lambda Functions

- functional programming paradigm (code-as-data)
- Java integrate functional -> OO paradigm
- pass "executable code segments"
- anonymous function / unnamed function
- implemented in language, libraries, VM
- "->" lambda functions, "::" method references
- implemented using invokedynamic (introduced instructions in Java 7)
- parallelism (multi cores)
- java.util.function
- java.util.streams (Stream API)
- thread safe mutations (final/effective final)

Examples lambda expressions:

```
(int x) -> x + x        -> explicitly typed
x -> x % x              -> implicitly typed
x -> (x>7?1:0)
() -> 7
() -> {return 7;}
(int a1, int a2) -> (a1 + a2) / (a1 - a2)
(int x) -> (this.myVar1 > x)
    --> this refers to object in scope/instance
```

```
CE: Failed
(a1, int a2) -> a1 / a2
(int a1) -> {a1>0}
-> 7

-   Duplicate identifier
-   Must: final/effective-final
-   Checked exception must be defined in interface
```

```
String word = "hello";
Int suffixFunc = ()->System.out.println(word + "x");
word = "e"; -> CE not final
suffixFunc.call();
```

Compare Analymous Class (J7) vs Labda (J8)

```
interface Function {
    void call();
}

class AnonymousInnerClass {
    public static void main(String []args) {
        Function function = new Function() {
            public void call() {
                System.out.println("Hello world");
            }
        };
        function.call();
    }
}


class FirstLambda {
    public static void main(String []args) {
        LambdaFunction lambdaFunction = () -> System.out.println("Hello world");
        lambdaFunction.call();
    }
}
```

Generics and Collections

Generic Classes

- to ensure type safety
- J5+
- type placeholder: <T>, <T1,T2,T3>, <KUKU>
- diamond syntax J7+: Pair x = new Pair<>();
- generic type parameters should match exactly for Assignments (**not covariant**)
- introduces an abstraction
- type erasure: Java compiler replace on compile
- **can't:** instantiate type "new T()"
- **can't:** instantiate array "new T[100]"
- **can't:** static fields;
- **can't:** generic exception classes
- **can't:** primitive data types
- **type erasure** by compiler: List<String> x → List x;

```
List<Double> list = new LinkedList<>();
System.out.println(list.getClass());
    class java.util.LinkedList  (note: type erasure)
```

```
class BoxPrinter<T> {
  private T val;
  public BoxPrinter(T arg) {
    val = arg;
  }
  public String toString() {
    return "[" + val + "]";
  }
}
…
new BoxPrinter<Integer>(new Integer(10));
new BoxPrinter<String>(new Integer(10)); -> CE
```

```
class Pair<T1, T2> {
  Pair(T1 one, T2 two) { }
}

new Pair<Integer, String>(2018, "Russia");
new Pair<Number, String>(2018, "Russia");

new Pair<String, String>(2018, "Russia"); -> CE

Pair<Integer, String> x =
    new Pair<Number, String>(1, "A"); -> CE
```

Generic Limitations:

```
T mem = new T();                    -> CE
T[] amem = new T[100];              -> CE
class X<T> {
    T instanceMem; -> OK
    static T statMem;               -> CE
}
class MyEx<T> extends Throwable { }  -> CE
List<Object> y = new ArrayList<Integer>(); -> CE
List<int> x;                        -> CE
```

```
new Pair<>(2018, "Russia"); -> Warning
-   unchecked conversion
-   will use Object
```

Generic's raw type

```
-   avoid it
-   type not specified
-   loose type safety / not recommended
-   coult throw: RTE
-   backward compatibility (to support J5)
-   Can use @SuppressWarnings({"unchecked"})

List<String> strList2 = new LinkedList<>();
strList2.add("A");
strList2.add("B");
List list2 = strList2;
list2.add(10); -> RTE ClassCastException
      generic type parameters should match exactly
for assignments
```

Generic Methods

```
class Utilities {
  public static <T> void fill(List<T> list, T val) {
    for(int I = 0; I < list.size(); i++)
        list.set(I, val);
  }
}
…
Utilities.fill(intList, 100); -> OK: as Interger
```

Wildcard Parameters

- can match for any type
- List<?> - list of unknowns
- ignoring type information
- **Can't:** call methods that modify the object
- **OK:** call methods that access object
- not covariant (subtyping doesn't work)

```
List<?> z = new ArrayList<Integer>(); -> OK
List<Number> x = new ArrayList<Integer>(); -> CE
List<Object> y = new ArrayList<Integer>(); -> CE
---
void printList(List<?> list){
    for(Object element: list)
      System.out.println("[" + element + "]");
}
---
List<?> wildCardList = new ArrayList<Integer>();
wildCardList.add(new Integer(10)); -> CE Can't call
methods that modify the object
public interface Collection<E> extends Iterable<E> {
…
  default boolean
    removeIf(Predicate<? super E> filter);
…
}
```

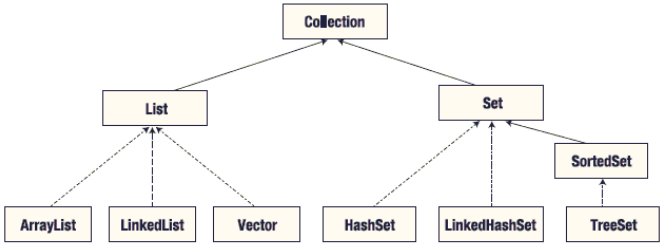
Collection Classes

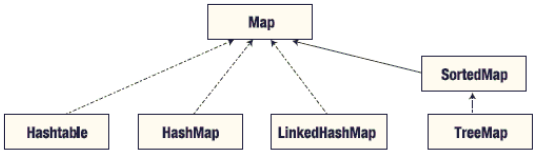
Terminology:

1) java.util.**Collection**<E> - root interface in collection hierarchy

2) java.util.**Collections** - utility class with static members

3) **general term collection**(s) - containers like map, stack, and queue





Interface	Duplicates Allowed?	Null Values Allowed?	Insertion order preserved?	Iterator	Data Structure
List	Yes	Yes, Multiple null values are allowed	Yes and can retrieve using index	Iterator, ListIterator	Array
Set	No	Yes but only once	No	Iterator	Underlying Map implementation
Map	Not for keys	Yes but only once for keys, can have multiple null values	No	Through keyset, value and entry set	Hashing techniques

Interfaces

Iterable	iterating with a foreach
Collection	base interface in collection hierarchy (eg. max() method in Collections takes a Collection)
List	sequence of ordered elements
Set, SortedSet, NavigableSet	unique elements (standard, sorted order, search closest matches)
Queue, Deque	Doubly ended queue, sequence of elements for processing (LIFO, FIFO). Deque from both ends
Map, SortedMap, NavigableMap	map keys to values **NOT** extend Collection
Iterator, ListIterator	traverse over the container Interator: forward ListInterator: forward/reverse

Classes

ArrayList	Ordered(remember insertion order) sequence of elements, resizable array, allows duplicates- Fast: search, slow: modify
LinkedList	doubly (двойная) linked list data structure. Fast: modify, slow: seach remember insertion order?
HashSet	Unordered unique hash-table Fast: all No sorting
TreeSet	Sorted Unique set (values only) Red-black tree SortedSet interface doesn't remember insertion order
HashMap	Unordered hash-table key+value Use hash for search/store Fast: all doesn't remember insertion order
TreeMap	Ordered by keys, key+value red-black tree, allows duplicate values **Slow** (vs HashMap): search, insert
PriorityQueue	heap data structure, retrieving elements by priority
ArrayDeque	Doubly ended queue retrieving elements by priority *[diff vs ArrayList: ths can add front/end only]*

java.util.Collection<E>

Method	Short description
boolean add(Element elem)	Adds elem into the underlying container.
void clear()	Removes all elements from the container.
boolean isEmpty()	Checks whether the container has any elements or not.
Iterator<Element> iterator()	Returns an Iterator<Element> object for iterating over the container.
boolean remove(Object obj)	Removes the element if obj is present in the container.
int size()	Returns the number of elements in the container.
Object[] toArray()	Returns an array that has all elements in the container.

Method	Short Description
boolean addAll(Collection<? extends Element> coll)	Adds all the elements in coll into the underlying container.
boolean containsAll(Collection<?> coll)	Checks if all elements given in coll are present in the underlying container.
boolean removeAll(Collection<?> coll)	Removes all elements from the underlying container that are also present in coll.
boolean retainAll(Collection<?> coll)	Retains elements in the underlying container only if they are also present in coll; it removes all other elements.

- if read-only: add(), remove() **RTE**
- stream() / parallelStream()

ArrayList class

- ordered sequence of elements
- resizable array, allows duplicates
- Fast: search
- slow: addition/deletion

```
List<String> myList = new ArrayList<>();
myList.add("A");
myList.add("B");
…
System.out.print(languageList); -> [A, B]
…
for(String txt : myList) {
    System.out.println(txt);
}
…
// iteration idiom:
for(
  Iterator<String> i = myList.iterator();
  i.hasNext();
) {
  String txt = i.next();
  System.out.println(txt);
}
```

```
while(i.hasNext()) {
   i.next();
   i.remove();-> OK (without next() -> RTE)
}
```

Arrays.asList()
- fixed-size list
- arr->list: for Collections utils
- **Can't:** add / remove elements
- Ok: modify

```
Double [] arr = {31.1, 30.0};
List<Double> list = Arrays.asList(arr);
list.set(0, 35.2); -> OK
list.remove(0); -> RTE
System.out.println(Arrays.toString(list));
     -> [35.2, 30.0]
```

TreeSet Class
- unique elements
- doesn't remember the insertion order
- stores elements in sorted order (SortedSet interface)

```
String txt = "xgaa 1X";
Set<Character> set = new TreeSet<Character>();
for(char gram : txt.toCharArray())
    set.add(gram);

System.out.print(set); -> [ , 1, X, a, g, x]
```

HashMap Class
- Unordered hash-table key+value
- Use hash for search/store
- Fast: all
- **doesn't** remember insertion order
- allows duplicate values

TreeMap Class
- Ordered by keys
- key+value red-black tree
- allows duplicate values
- Slow (vs HashMap): search, insert

NavigableMap + TreeMap
- can navigate the Map

```
NavigableMap<Integer, String> examScores = new
TreeMap<Integer, String>();
examScores.put(90, "Sophia");
examScores.put(20, "Isabella");
examScores.put(10, "Emma");
examScores.put(50, "XXXX");
examScores.put(50, "Olivea"); -> overwrite

System.out.println(examScores.descendingMap());
    {90=Sophia, 50=Olivea, 20=Isabella, 10=Emma}
System.out.println(examScores.tailMap(40));
    {50=Olivea, 90=Sophia}
System.out.println(examScores.firstEntry());
    10=Emma
System.out.println(examScores.firstEntry());
    90=Sophia
```

Deque interface + ArrayDeque class
- Doubly ended queue
- diff ArrayList: can add/remove front/back
- Fast: insert
- Slow: navigation

```
Deque<String> q = new ArrayDeque<>();
q.addLast("A");
q.addLast("B");
q.addFirst("X");
System.out.println(q);        -> [X, A, B]

String item = q.removeLast();
System.out.println(item);     -> B
System.out.println(q);        -> [X, A]
```

Comparable and Comparator Interfaces
- searching or sorting
- define criteria for object comparison
- java.lang.**Comparable**<T>:
  int compareTo(Student that)
  For natural/single ordering
- java.util.**Comparator**<T>:
  int compare(Student s1, Student s2)
  If multiple alternative orderings needed

java.lang.Comparable<T>:
- if one option how to compare
- natural order

```
int compareTo(Element that)
     1 if current >  passed object
     0 if current == passed object
    -1 if current <  passed object
----
class Student implements Comparable<Student> {
    String id;
    …
    public int compareTo(Student that) {
        return this.id.compareTo(that.id);
    }
}
…
Student []students = ...
Arrays.sort(students);
Arrays.sort(students, null); -> OK (natural sorting)
```

java.util.Comparator<T>:
- if multiple alternative ways to compare two similar objects
- don't change original class
- special (not natural) ordering is required

```
class CGPAComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2) {
        return (s1.cgpa.compareTo(s2.cgpa));
    }
}
…
Arrays.sort(students, new CGPAComparator());
Arrays.sort(students, null); -> OK (natural sorting)
```

Collection Streams and Filters

- **java.util.stream** J8+
- Stream<T> interface
- IntStream, LongStream, and DoubleStream (for int, long, double primitives)
- Collection: stream(), parallelStream()
- List, Set, Deque, and Queue extend Collection
- stream is sequence of elements (pipeline)
- pipelining capability (filter, map, search)
- can use streams API independent of collections

---

forEach

- supports internal iteration (functional programming J8+)

```
list.forEach(s -> System.out.println(s));
list.forEach(System.out::println);
```

Method References :: with Streams

- route the given parameters
- syntax makes it easier to use

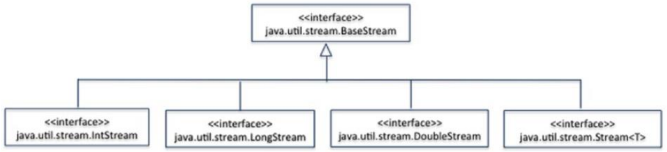
```
list.forEach(System.out::println);
       = string -> System.out.println(string)
```

```
Supplier<String> newString = String::new;
```

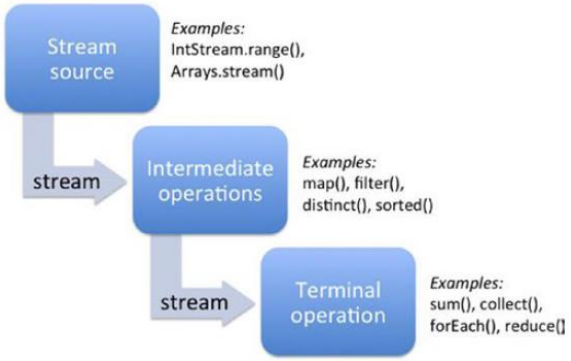
```
strings.forEach(string ->
System.out.println(string.toUpperCase()));
…
class MethodReference {
 public static void printMe(String s) {
   System.out.println(s.toUpperCase());
 }
}
…
list.forEach(MethodReference::printMe);

--------
static <T> Predicate<T> isEqual(Object targetRef) {
     return (null == targetRef)
             ? Objects::isNull
             : object -> targetRef.equals(object);
}
```

Stream<T> Interface

- pipeline lazily evaluated (if absent terminal operation count(), forEach(), reduce(), or collect(), then entire pipeline not evaluated)
- could be terminated once (



Stream Pipeline:




```
Stream.of(1, 2, 2, 9).distinct().count(); -> 3
```

```
Arrays.stream(Object.class.getMethods()) // source
   .map(m -> m.getName())      // intermediate op
   .distinct()                 // intermediate op
                    … chained in any order …
   .forEach(System.out::println); // terminal op

Same as:
  Method[] m = Object.class.getMethods();
  Stream<Method> ms = Arrays.stream(m);
  Stream<String> names = ms.map(i -> i.getName());
  Stream<String> unique = names.distinct();
  unique.forEach(System.out::println);

map != java.util.Map
```

Stream Sources

```
IntStream.range(1, 6)          1..5
IntStream.rangeClosed(1, 6)    1..6
IntStream.iterate(1, i -> i + 1).limit(5)

Arrays.stream(new int[] {1, 2, 3, 4, 5})
Arrays.stream(new Integer[] {1, 2, 3, 4, 5})

Stream.of(1, 2, 3, 4, 5)
Stream.of(new Integer[]{1, 2, 3, 4, 5})
Stream.of("A","B").forEach(System.out::println);

Stream.builder().add(1).add(2).add(3).build()
```

Other Examples:

```
Files.lines(Paths.get("./FileRead.java"))
  .forEach(System.out::println);

Pattern.compile(" ")
  .splitAsStream("java 8 streams")
  .forEach(System.out::println);

new Random().ints().limit(5)
  .forEach(System.out::println);

"hello".chars().sorted()
  .forEach(ch -> System.out.printf("%c ", ch));
```

Stream Intermediate Operations

- transform elements in a stream
- optional
- could be chained
- can use a stream only once

Method	Short description
`Stream<T> filter(Predicate<? super T> check)`	Removes the elements for which the `check` predicate returns false.
`<R> Stream<R> map(Function<? super T,? extends R> transform)`	Applies the `transform()` function for each of the elements in the stream.
`Stream<T> distinct()`	Removes duplicate elements in the stream; it uses the `equals()` method to determine if an element is repeated in the stream.
`Stream<T> sorted()` `Stream<T> sorted(Comparator<? super T> compare)`	Sorts the elements in its natural order. The overloaded version takes a `Comparator`—you can pass a lambda function for that.
`Stream<T> peek(Consumer<? super T> consume)`	Returns the same elements in the stream, but also executes the passed `consume` lambda expression on the elements.
`Stream<T> limit(long size)`	Removes the elements if there are more elements than the given `size` in the stream.

```
Stream.of(1, 2, 3, 4, 5)
  .map(i -> i * i)
  .peek(i -> System.out.printf("%d ", i))
  .count();
1 4 9 16 25
```

```
Stream.of(1, 2, 3, 4, 5)
  .peek(i -> System.out.printf("%d ", i))
  .map(i -> i * i)
  .peek(i -> System.out.printf("%d; ", i))
  .count();
1 1; 2 4; 3 9; 4 16; 5 25;
```

```
IntStream.rangeClosed(0, 10)
  .filter(i -> (i % 2) == 0)
  .forEach(System.out::println);

      IntStream filter(IntPredicate predicate)
      boolean test(int value);

IntStream.rangeClosed(0, 10)
  .map(i -> i * i)
  .filter(i -> (i % 2) == 0)
  .forEach(i -> System.out.printf("%d ", i));
0 4 16 36 64 100
```

Stream Terminal Operations

- pipeline lazily evaluated (if absent terminal operation, then entire pipeline not evaluated)
- end of a pipeline
- terminal operation consider stream as "consumed" after it **can't** "use" it agail

Method	Short description
`void forEach(Consumer<? super T> action)`	Calls the `action` for every element in the stream.
`Object[] toArray()`	Returns an `Object` array that has the elements in the stream.
`Optional<T> min(Comparator<? super T> compare)`	Returns the minimum value in the stream (compares the objects using the given `compare` function).
`Optional<T> max(Comparator<? super T> compare)`	Returns the maximum value in the stream (compares the objects using the given `compare` function).
`long count()`	Returns the number of elements in the stream.

+ InttStream, LongStream, and DoubleStream have methods such as sum(), min(), max(), and average()

+ reduce(), collect(), findFirst(), findAny(), anyMatch(), allMatch(), and noneMatch()

```
Object [] words = Pattern.compile(" ")
  .splitAsStream("1 2 3 4 5")
  .toArray();  -> Stream<String> -> Object[]

System.out.println(
  Arrays.stream(words)
    .mapToInt(str -> Integer.valueOf((String)str))
    .sum()
); -> 15
```

```
IntStream chars = "bookkeep".chars();
System.out.println(chars.count()); -> "consumed"
System.out.println(chars.count()); -> RTE
IllegalStateException (stream has already been
operated upon or closed)
```

```
"abracadabra".chars()
   .distinct()
   .peek(ch -> System.out.printf("%c ", ch))
   .sorted();

 -> no output, because pipeline lazily evaluated:
      if absent terminal operation,
      then entire pipeline not evaluated
```

Lambda Built-in Functional Interfaces

- build-in func interf: java.util.function.*
- **used by** java.util.stream.*
- 1 abstract method + 0..N default/static/Object
- before developing own, check existing

Predicate<T>	"tests" **boolean test(T t)** and(Predicate p) or(Predicate p) negate()	.removeIf(…) .anyMatch(…) .allMatch(…) .noneMatch(…)
Long/Double IntPredicate<T>		
BiPredicate<T, U>	boolean test(T t, U u)	
Consumer<T>	"consumes" **void accept(T t)** andThen(Consumer c)	.forEach(…)
Long/Double IntConsumer<T>	void accept(int value) ObjIntConsumer<T> void accept(T t, int v)	
BiConsumer<T, U>	void accept(T t, U u)	
Function<T,R>	"operates" **R apply(T t)** andThen(Consumer c) ->* compose(Consumer c) <-* identity() (a->a)	.map(…) .mapToInt*(…)
Long/Double IntFunction<T>	IntFunction f=Math::abs; ToIntDunction T->int IntToDouble int->double …	
BiFunction<T,U,R>	R apply(T t, U u);	
UnaryOperator<T> Long/Double/ IntUnaryOperator	**T apply(T t)** operand/result T same	lst.replaceAll(Math::abs)
BinaryOperator	extends BiFunction<T,T,T> T apply(T t, T t);	
Supplier<T>	"supplies" **T get()**	.generate(…)
Long/Double/ Boolean IntSupplier<T>	BooleanSupplier: Boolean getAsBoolean() IntStream.generate(ints: :incrementAndGet)	
No: BiSupplier		

Using Build-In Functional Interfaces

interface Predicate<T>
- checks condition and return a boolean value

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

```
and(Predicate p2);     -> &&
or(Predicate p2);      -> ||
negate();              -> !
```

```
Predicate<String> nullCheck =arg -> arg != null;
Predicate<String> emptyCheck =arg -> arg.length()>0;
Predicate<String> xxx = nullCheck.and(emptyCheck);
String helloStr = "hello";
System.out.println(xxx.test(helloStr));
       -> true

String nullStr = null;
System.out.println(xxx.test(nullStr));
       -> false
```

```
list.removeIf(str -> !str.startsWith("h"));
```

```
list.removeIf(
   ((Predicate<String>) str -> str.startsWith("h"))
    .negate()
);
list.removeIf(
   (str -> str.startsWith("h")).negate() -> CE
);
```

interface Consumer<T>
- take 1 argument, without return

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
  …
}
```

```
Stream.of("A","B").forEach(System.out::println);
Same as:
  Stream<String> s = Stream.of("A", "B");
  Consumer<String> print = System.out::println;
  s.forEach(print);
```

```
Consumer<String> printUpperCase
  = str -> System.out.println(str.toUpperCase());
printUpperCase.accept("hello"); -> HELLO
```

```
andThen(); allows to chaining calls to
Consumer objects
```

```
Consumer<String> c1 = String::toLowerCase;
Consumer<String> c2 = System.out::print;
Stream.of("A", "B").forEach(c1.andThen(c2));
       -> AB

Consumer<String> c1= s->System.out.print("["+s+"]");
Consumer<String> c2= System.out::print;
Stream.of("A", "B").forEach(c1.andThen(c2));
       -> [A]A[B]B
```

interface Function<T, R>
- takes argument T and returns value of type R

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    …
}
```

```
Arrays.stream("4, -9, 16".split(", "))
  .map(Integer::parseInt)
  .map(i -> (i < 0) ? -i : i)
  .forEach(System.out::println);
      -> 4 9 16
```

```
Arrays.stream("4, -9, 16".split(", "))
  .map(Function.identity())  - just for testing
  .forEach(System.out::println);
      -> 4 -9 16
```

```
Function<String, Integer> strLength
    = str -> str.length();
System.out.println(strLength.apply("abc")); -> 3
```

```
Function<String, Integer> a  = Integer::parseInt;
Function<Integer,Integer> b = Math::abs;
Function<String, Integer> c  = a.andThen(b);

Function<String, Integer> c2  = b.compose(a);

Arrays.stream("4, -9, 16".split(", "))
  .map(c)
  .forEach(System.out::println); -> 4,9,16
```

interface Supplier<T>
- no argument, returns value of type T

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
    …
}
```

```
Random random = new Random();
Stream.generate(random::nextBoolean)
   .limit(2)
   .forEach(System.out::println);

    static <T> Stream<T> generate(Supplier<T> s)
```

```
Supplier<String> currentDateTime
    = () -> LocalDateTime.now().toString();

System.out.println(currentDateTime.get());
```

Constructor ::new References

```
Supplier<String> newString = String::new;
Supplier<String> newString = () -> new String();
String x = newString.get();
```

```
Function<String, Integer> xxx = Integer::new;
Integer my = xxx.apply("100");
```

Primitive Versions of Functional Interfaces

```
Why separate classes are needed:
 1) can't use primitive type values in F<T>
 2) if use Integer/Double/Long, then:
    implicit autoboxing/unboxing  (int->Integer)
    performance can suffer
    not recommended
- primitive versions of functional interfaces
```

Predicate for primitives

```
IntPredicate evenNums = i -> (i % 2) == 0;
IntStream.range(1,10)
   .filter(evenNums)
   .forEach(System.out::println);

IntPredicate: boolean test(int value)
LongPredicate: boolean test(long value)
DoublePredicate: boolean test(double value)
```

Function for primitives

```
AtomicInteger ints = new AtomicInteger(0);
Stream.generate(ints::incrementAndGet)
   .limit(10)
   .forEach(System.out::println);
-> BAD(int->Integer)
Good (aovid autobixing/unboxing)
IntStream.generate(ints::incrementAndGet)
   .limit(10)
    .forEach(System.out::println);
---
Bad: Function<Integer, Integer> absInt = Math::abs;
Good: IntFunction absInt = Math::abs;
---
```

Functional Interface	Abstract Method	Brief Description
IntFunction<R>	R apply(int value)	Operates on the passed int argument and returns value of generic type R
LongFunction<R>	R apply(long value)	Operates on the passed long argument and returns value of generic type R
DoubleFunction<R>	R apply(double value)	Operates on the passed double argument and returns value of generic type R
ToIntFunction<T>	int applyAsInt(T value)	Operates on the passed generic type argument T and returns an int value
ToLongFunction<T>	long applyAsLong(T value)	Operates on the passed generic type argument T and returns a long value
ToDoubleFunction<T>	double applyAsDouble(T value)	Operates on the passed generic type argument T and returns an double value
IntToLongFunction	long applyAsLong(int value)	Operates on the passed int type argument and returns a long value
IntToDoubleFunction	double applyAsDouble(int value)	Operates on the passed int type argument and returns a double value
LongToIntFunction	int applyAsInt(long value)	Operates on the passed long type argument and returns an int value
LongToDoubleFunction	double applyAsLong(long value)	Operates on the passed long type argument and returns a double value
DoubleToIntFunction	int applyAsInt(double value)	Operates on the passed double type argument and returns an int value
DoubleToLongFunction	long applyAsLong(double value)	Operates on the passed double type argument and returns a long value

Consumer for primitives

Functional Interface	Abstract Method	Brief Description
IntConsumer	void accept(int value)	Operates on the given int argument and returns nothing
LongConsumer	void accept(long value)	Operates on the given long argument and returns nothing
DoubleConsumer	void accept(double value)	Operates on the given double argument and returns nothing
ObjIntConsumer<T>	void accept(T t, int value)	Operates on the given generic type argument T and int arguments and returns nothing
ObjLongConsumer<T>	void accept(T t, long value)	Operates on the given generic type argument T and long arguments and returns nothing
ObjDoubleConsumer<T>	void accept(T t, double value)	Operates on the given generic type argument T and double arguments and returns nothing

Supplier for primitives

Functional Interface	Abstract Method	Brief Description
BooleanSupplier	boolean getAsBoolean()	Takes no arguments and returns a boolean value
IntSupplier	int getAsInt()	Takes no arguments and returns an int value
LongSupplier	long getAsLong()	Takes no arguments and returns a long value
DoubleSupplier	double getAsDouble()	Takes no arguments and returns a double value

```
int -> char, byte, or short
double -> float
```

Binary Versions of Functional Interfaces

```
- "Bi" takes "two" arguments
- BiFunction != BinaryFunction
```

Functional Interface	Abstract Method	Brief Description
BiPredicate<T, U>	boolean test(T t, U u)	Checks if the arguments match the condition and returns a boolean value as result
BiConsumer<T, U>	void accept(T t, U u)	Operation that consumes two arguments but returns nothing
BiFunction<T, U, R>	R apply(T t, U u)	Function that takes two argument and returns a result

```
BiFunction<String,Integer,String> f=(x,y)->x+(y*2);
System.out.println(f.apply("A", 2));
    -> A4
----
BiFunction<Double,Double,Integer> c=Double::compare;
System.out.println(c.apply(10.0, 10.0));
```

```
BiPredicate<List<Integer>,Integer> f=List::contains;
List aList = Arrays.asList(10, 20, 30);
System.out.println(f.test(aList, 20));
     -> true
NOTE: BiFunction<T,U,Boolean> as option, but slower
```

```
BiConsumer<List<Integer>, Integer> f = List::add;
List aList = new ArrayList();
f.accept(aList, 10);
System.out.println(aList); -> [10]
```

Stream API

Extract Data (peek/map)

```
- peek() for debugging (not recommended in prod)

long count = Stream.of(1, 2, 3, 4, 5)
   .peek(i -> System.out.printf("%d-", i))
   .map(i -> i * i)
   .peek(i -> System.out.printf("%d ", i))
   .count();
System.out.printf("; Count=%d", count);
     1-1 2-4 3-9 4-16 5-25; Count=5
```

Search Data

Method Name	Short Description
`boolean anyMatch(Predicate<? super T> check)`	Returns true if there is any elements in the stream that matches the given predicate. Returns false if the stream is empty or if there are no matching elements.
`boolean allMatch(Predicate<? super T> check)`	Returns true only if *all* elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
`boolean noneMatch(Predicate<? super T> check)`	Returns true only if *none* of the elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
`Optional<T> findFirst()`	Returns the first element from the stream; if there is no element present in the stream, it returns an empty `Optional<T>` object.
`Optional<T> findAny()`	Returns one of the elements from the stream; if there is no element present in the stream, it returns an empty `Optional<T>` object.

anyMatch/ allMatch/ noneMatch

```
boolean b1 = IntStream.of(1,2,3).anyMatch(e-> e>0);
boolean b2 = IntStream.of(1,2,3).allMatch(e-> e>0);
boolean b3 = IntStream.of(1,2,3).noneMatch(e-> e>0);
```

findFirst / findAny / Optional / OptionalInt*

```
- findAny faster in parallel
- short-circuiting (Terminates/Closes Stream)
- java.util.Optional

Method[] methods = Stream.class.getMethods();
Optional<String> methodName = Arrays.stream(methods)
   .map(method -> method.getName())
   .filter(name -> name.endsWith("Match"))
   .sorted()
   .findFirst();
System.out.println(methodName.orElse("Not found"));
```

```
OptionalDouble temp = DoubleStream.of(1.0,-2.0)
    .filter(t -> t > 0)
    .findAny();
System.out.print(temp.orElse(0.0));
```

```
IntStream s = IntStream.of(1,2);
s.findAny();
s.findAny(); -> RTE IllegalStateException
```

```
boolean result =
  Stream.of("do", "re")
    .filter(str -> str.length() > 5)    (empty stream)
    .allMatch(str -> str.length() > 5); empty = true
System.out.println(result);           -> true
```

Optional / OptionalInt*

```
- Optional<T>
- OptionalInt, OptionalLong, OptionalDouble

Optional<String> empty = Optional.empty();
Optional<String> nonEmpty = Optional.of("XXX");

       Optional.of("XXX");       -> Optional[XXX]
       Optional.of("XXX").get()  -> XXX

---
Optional<String> nullStr = Optional.of(null); -> RTE
Optional<String> empt = Optional.ofNullable(null);
System.out.println(empt); -> Optional.empty
```

```
IntStream.of(1,2).max(); -> OptionalInt[2]
IntStream.of().max(); -> OptionalInt.empty

Optional<T> max(Comparator<? super T> comparator);
Stream.of(1,2).max(Integer::compareTo);
       -> Optional.empty
```

```
Stream<Double> temp = Stream.of(1.0,-2.0);
Optional<Double> max = temp.max(Double::compareTo);
max.ifPresent(System.out::println);
//same:
if(max.isPresent()) {
    System.out.println(max.get());
}
```

```
Optional<String> s = Optional.of(" A ");
s.map(String::trim).ifPresent(System.out::println);
```

```
Optional<Integer> x =
   Stream.of(1,2,3).reduce((a,b)->a+b).get(); -> 6
```

When operation fail:

```
Optional<String> s = Optional.ofNullable(null);
System.out.println(
    s.map(String::length).orElse(-1));
---
s.map(String::length)
  .orElseThrow(IllegalArgumentException::new)
```

```
public class StringToUpper {
    public static void main(String args[]){
        Stream.of("eeny ","meeny ",null).forEach(StringToUpper::toUpper);
    }
    private static void toUpper(String str) {
        Optional <String> string = Optional.ofNullable(str);
        System.out.print(string.map(String::toUpperCase).orElse("dummy"));
    }
}
```

This program prints: EENY MEENY dummy

Calculation Methods

```
- Stream<T>: count(), min() and max()
- Int*Stream:+ sum(), average(),summaryStatistics()
- Int*SummaryStatistics: sum,count,average…

String[] s = "Z0 A1 B C3".split(" ");
Arrays.stream(s)
  .min(String::compareTo).get() -> A1

Comparator<String> lengthCompare =
   (s1,s2) -> s1.length()-s2.length();

Arrays.stream(s)
   .min(lengthCompare).get() -> B
```

```
IntSummaryStatistics stats =
  Pattern.compile(" ")
    .splitAsStream(limerick)
    .mapToInt(word -> word.length())
    .summaryStatistics(); -> only primitive has it
---
        stats.getCount()
        stats.getSum()
        stats.getMin()
        stats.getMax()
        stats.getAverage()
```

```
IntStream s = IntStream.of(1, 2, 3);
---
s.sum();                    -> 7 implicit reducer
s.reduce(0, Integer::sum); -> 7 explicit reducer
---
s.reduce(0, ((sum, val) -> sum + val));
s.reduce((sum, val) -> sum + val);
    -> Optional[7]
---
s.reduce((x, y) -> (x*y)).getAsInt() -> 12
```

Sorting

```
List words = Arrays.asList("AA A AB B".split(" "));
---
words.stream()
  .distinct()
  .sorted().reversed()
  .forEach(System.out::println);      -> A AA AB B
---
Comparator<String> lenCmp =
   (s1, s2) -> s1.length()-s2.length();
---
words.stream()
  .distinct()
  .sorted(lenCmp.thenComparing(String::compareTo))
  .forEach(System.out::println);      -> A B AA AB
---
.sorted(
       lenCmp
         .thenComparing(String::compareTo)
         .reversed()
)                                     -> AB AA B A
```

Save to Collection / Grouping

```
- stream to collection
- toList(), toSet(), toMap(), toCollection()

      <R, A> R collect(
      Collector<? super T, A, R> collector);
```

```
String txt = "a:b:b";
List<String> list = Pattern.compile(":")
 .splitAsStream(txt)
 .collect(Collectors.toList());
System.out.println(list);              -> [a, b, b]
```

```
Set<String> set = Pattern.compile(":")
 .splitAsStream(txt)
 .collect(Collectors.toSet());         -> [a, b, b]
```

```
Map<String, Integer> txtLen =
   Stream.of("A", "BB", "CCC")
   .collect(Collectors.toMap(a->a, a->a.length()));
                                    -> {BB=2, A=1, CCC=3}
       NOTE: Map does not maintain insertion order

// Same (a->a):
Collectors.toMap(Function.identity(),a->a.length())
```

```
Set words = stream
  .collect(Collectors.toCollection(TreeSet::new));
```

---

groupingBy / partitioningBy

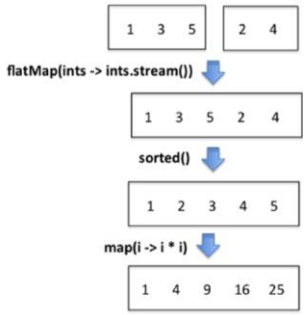
```
- groupBy - classifies by Function
- partitioningBy - by Predicate true/false

String[] txt = "a b cc".split(" ");
Stream<String> words= Arrays.stream(txt).distinct();
Map<Integer, List<String>> wordGroups = words
   .collect(Collectors.groupingBy(String::length));

wordGroups.forEach((count, words) -> {
  System.out.printf("word length %d %n", count);
  words.forEach(System.out::println);
});
```

```
Map<Boolean, List<String>> wordBlocks = words
    .collect(
  Collectors.partitioningBy(s->s.length() > 1)
);
wordBlocks.get(true);  -> [cc]
wordBlocks.get(false); -> [a, b]
```

flatMap

- flattens the streams that result from mapping each of its elements into one flat stream

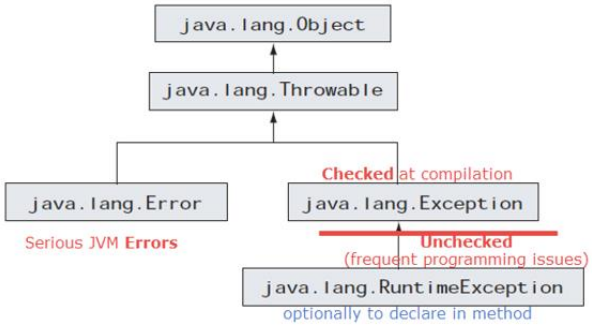
```
List<List<Integer>> intsOfInts =
  Arrays.asList(
        Arrays.asList(1, 3, 5),
        Arrays.asList(2, 4)
  );

intsOfInts.stream()
  .flatMap(ints -> ints.stream())
  .sorted()
  .map(i -> i * i)
  .forEach(System.out::println); -> 1, 4, 9, 16,25
```



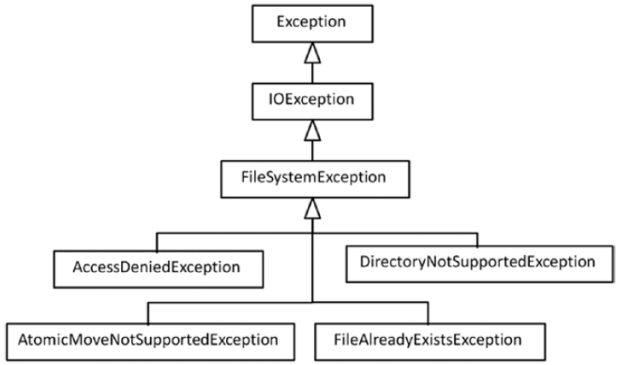

```
String []string= "aa bb cc a".split(" ");
Arrays.stream(string)
  .flatMap(word -> Arrays.stream(word.split("")))
  .distinct()
  .forEach(System.out::print);  -> a,b,c
```

Exceptions

- Terminate program: System.exit(-1);
- stack trace (hierarchy of methods + line numbers)
- no throws clause, means that method don't throws checked exceptions (can throw other like unchecked)
- catch block: handle/retrow (BAD: hide/swallow)
- GOOD: chained exceptions: wrap one into another (can get cause / stack)
- try-with-resources suppressed exceptions: getSupressed() – list of exceptions thrown in try(…) finally{…}




```
java.lang.Object -> java.lang.Throwable:
Exception                - checked exceptions
  IOException
        FileNotFoundException
  <...>
  RuntimeException       - unchecked exceptions
    IndexOutOfBoundsException          -> list.get(-1)
        ArrayIndexOutOfBoundsException -> arr[-1]
    ClassCastException
    IllegalArgumentException
        NumberFormatException
    NullPointerException
    ArithmeticException          -> 1/0
    NumberFormatException        -> parseInt("x");
    NoSuchElementException
        InputMismatchException -> scanner.getInt()
    IllegalStateException -> nextInt() after close()

Error                    - JVM serious errors
  LinkageError
    ExceptionInInitializerError -> static block/var
    NoClassDefFoundError -> NOT as Class.forName..
  VirtualMachineError
    StackOverflowError    -> infinite recusion calls
    OutOfMemoryError      -> heap size
  AssertionError
```



Catch order matters for IS-A classes!
Watch for unreachable statements
Validation is top-down.

Exception in overridden methods / i:

Unchecked/Runtime:
OK: Unchecked exceptions can still be added or removed from the contract

Checked:
OK: more detailed exception in impl/derived class
In interface: IOException
In class: FileNotFoundException
baseclass: void m1() throws Exception;
subclass: void m1() throws IOException; -> ok

OK: no throws clause in the overriding method
baseclass: void m1() throws Exception;
subclass: void m1(); -> ok

Fail: more general exception than specified in the base class will result in a CE
baseclass: void m1() throws IOException;
subclass: void m1() throws Exception; -> CE

Fail: if checked exception is not specified in baseclass -> CE

```
interface Int1 {
        void m() throws IOException;
}
interface Int2 {
        void m() throws FileAlreadyExistsException;
}
class Class1 implements Int1, Int2 {
  public void m() throws FileAlreadyExistsException{
  }
} -> IOException will cause CE
```

```
Both extends: IOException
interface Int1 {
  void m() throws FileNotFoundException;
}
interface Int2 {
  void m() throws UnsupportedEncodingException;
}
class Class1 implements Int1,Int2 {
  public void m()
     throws FileNotFoundException,
            UnsupportedEncodingException -> CE {
          …
       }
}
```

Exceptions in Initializers:
- direct throw is not allowed
- direct of call method:

```
public static void xxx() throws IOException { }

{
  xxx(); -> checked: ok if handled in constructors
         -> unchecked: ok
  throw new IOException();-> CE
}

public Test() throws IOException { }

static {
  xxx(); -> checked: CE
         -> unchecked: ok
  throw new IOException();-> CE
}
```

```
Scanner scanner = new Scanner(System.in);
System.out.println(consoleScanner.nextInt());
       -> InputMismatchException (if invalid input)
```

```
try {} catch (FileNotFoundException e) {}
       → fail, checked exception not thrown

try {} finally {}
```

```
try {
  ...
  return 10;
} catch (...) {
  ...
  return 20;
} finally {
  …executed, if no fatal exception/Syste.exit()…
  return 30; → final return!
}
```

```
int getInt() {
  int x=10;
  try {…}
  } catch (...) {
     return x;
  } finally {
    x += 5; → copy of primitive! StringBuilder:diff
  }
  return x;
}  getInt() == 10
```

Which methods will compile if inserted into Joey? (Choose all that apply)

```
class Kangaroo {
  public void hop() { }
}
class Joey extends Kangaroo {
  // INSERT CODE HERE
}
```

A. `public void hop() {}`

B. `public void hop() throws Exception {}`

C. `public void hop() throws IOException {}`

D. `public void hop() throws RuntimeException {}`

@throws in JavaDoc

```
/**
* ...
* @throws NoSuchElementException if input exhausted
* @throws IllegalStateException if scanner is closed
*/
```

Multi-Catch Blocks

- if "similar" reasons
- if "similar" handling code

```
try {…}
catch( NoSuchElementException
     | IllegalStateException e ) { … }
----------
Can't: combine base/derived exceptions!
   catch(InputMismatchException
       | NoSuchElementException e) -> CE
    Alternative: Use only base class
```

Releasing Resources

- resource leak (opened, but not closed)
- network, file, database, ...
- GC not responsible for close()

```
java.io.Closeable: void close();
```

Option 1:

```
try {…} catch(…) {…} finally { res.close();… }
NOTE: if no System.exit();
```

Option 2:

```
try (…) {…} catch(…) {…}
```

Throws

- method throws checked exceptions (help to understand possible failures)

```
-----
public static void main(String[] args)
   throws FileNotFoundException
     -> Othewise unreported exception ... must be
     caught or declared to be thrown
{
  Scanner scanner = new Scanner(new File("A.txt"));
  scanner.nextInt();
}
```

exception chaining:

```
  catch (Exception e) {
      throw new Exception("tadaa", e);
  }

NOTE: Exception thrown from inside finally block
masks the exception thrown in the catch block! as if
the previous exception was not thrown at all
```

```
try {
  LoginException le = new AccountNotFound();
  throw (Exception) le;
} catch (AccountNotFound anfe) {
   System.out.println("1"); -> executed
}  catch (Exception e) {
   System.out.println("2");
}
```

Try-with-Resources

- compiler transforms to finally
- simplify (don't define finally blocks explicitly)
- MUST implement java.lang.AutoCloseable interface
- Closeabe extends AutoCloseable
- the resource in try-with-resources statement cannot be assigned within body

```
try(Scanner scanner = new Scanner(System.in); … ) {
  System.out.println(scanner.nextInt());
  scanner = … -> CE Can't reassign
  scanner.close(); -> duplicate all, some can RTE
} catch(Exception e) {
  ...
}
try ( … ) { … }          -> OK
try { … }                -> CE diff: try-with-resource
try { … } finally { } -> OK
```

Multiple resources:

```
try (
  ZipOutputStream zipFile = new ZipOutputStream(new FileOutputStream(zipFileName));
  FileInputStream fileIn = new FileInputStream(fileName))
{ … }
```

custom class for try-with-resource

```
class My implements AutoCloseable {
  @Override
  public void close() {
    System.out.println("Closing...");
  }
}
```

suppressed exceptions

- try-with-resources coult have multiple exceptions (try/catch/finally)
- e.getSuppressed()

Custom Exceptions

- first check existing
- Exception/RuntimeException
- Throwable/Error – not recommended

Member	Short description
`Exception()`	Default constructor of the `Exception` class with no additional (or detailed) information on the exception.
`Exception(String)`	Constructor that takes a detailed information string about the constructor as an argument.
`Exception(String, Throwable)`	In addition to a detailed information string as an argument, this exception constructor takes the cause of the exception (which is another exception) as an argument.
`Exception(Throwable)`	Constructor that takes the cause of the exception as an argument.
`String getMessage()`	Returns the detailed message (passed as a string when the exception was created).
`Throwable getCause()`	Returns the cause of the exception (if any, or else returns null).
`Throwable[] getSuppressed()`	Returns the list of suppressed exceptions (typically caused when using a try-with-resources statement) as an array.
`void printStackTrace()`	Prints the stack trace (i.e., the list of method calls with relevant line numbers) to the console (standard error stream). If the cause of an exception (which is another exception object) is available in the exception, then that information will also be printed. Further, if there are any suppressed exceptions, they are also printed.

```
e.initCause(e2); -> RTE IllegalStateException
```

```
class InvalidInputException extends RuntimeException {
public InvalidInputException() {
  super();
}
public InvalidInputException(String str) {
  super(str);
}
public InvalidInputException(Throwable originalException) {
  super(originalException);
}
public InvalidInputException(String str, Throwable
originalException) {
  super(str, originalException);
}
}

throw new InvalidInputException("kuku", nsee);
```

```
class MyException extends RuntimeException { }
throw new MyException(); -> ok (comp adds default)
throw new MyException("kuku"); -> CE
```

Assertions

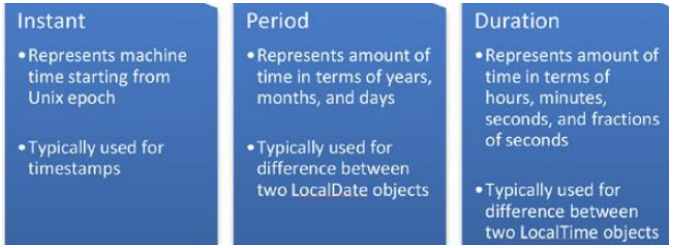
- AssertionError
- check/test assumptions about the program
- terminate if FALSE
- helps find the critical problems early
- Disabled by default! VM arg:
  enable: **-ea** (-enableasserts)
  disable **-da**

```
If false, then throws AssertionError
    assert (i>=0);
    assert (i>=0) : "impossible value!";
    assert (i+k); -> CE
```

Command-Line Argument	Short Description
`-ea`	Enables assertions by default (except system classes).
`-ea:<class name>`	Enables assertions for the given class name.
`-ea:<package name>...`	Enables assertions in all the members of the given package <package name>.
`-ea:...`	Enable assertions in the given unnamed package.
`-esa`	Short for `-enablesystemsassertions`; enables assertions in system classes. This option is rarely used.
`-da`	Disable assertions by default (except system classes).
`-da:<class name>`	Disable assertions for the given class name.
`-da:<package name>...`	Disables assertions in all the members of the given package <package name>.
`-da:...`	Disable assertions in the given unnamed package.
`-dsa`	Short for `-disablesystemsassertions`; disables assertions in system classes. This option is rarely used.

Date/Time API

- **java.time.*** package
  - java.time.**temporal**.* - fields/units
  - java.time.**format**.* - Format/Output
  - java.time.**zone**.* - Time zones
  - java.time.**chrono**.* - Japanese, Thai, …
- replaces old java.util* (Date, Calendar, TimeZone)
  - inconvenient/inconsistent API design
  - days 1..31, but month 0..11 etc
  - not thread-safe
- most classes: immutable/thread-safe
- **fluent** interfaces (code is readable & easy to use)
- classes with many **factory methods**
- ISO 8601 default calendar format (order by: year, month … nanoseconds)



LocalDate (2018-02-09) IMMUTABLE

- LocalDate, LocalTime, and LocalDateTime implements TemporalAccessor

```
LocalDate d =
    LocalDate.of(2015,1,2);
    LocalDate.of(2015,14,2); -> RTE
    LocalDate.of(2015,Month.DECEMBER,27);

    LocalDate.ofYearDay(2016,100);    -> 2016-04-09

    LocalDate.ofEpochDay(10);         -> 1970-01-11

    LocalDate.parse("2025-08-09");
    LocalDate.parse("2025-8-9"); → RTE

    LocalDate.now();
    LocalDate.now(Clock.systemDefaultZone());
    LocalDate.now(ZoneId.of("Asia/Kolkata"));

d.getDayOfMonth();
d.getDayOfWeek(); → SUNDAY
d.getDayOfYear();
d.getMonth(); → JANUARY
d.getMonthValue();
d.getYear();
d.getYears(); → fail

d1.isAfter(d2);
d1.isBefore(d2);
d1.isBefore(d2);

LocalDate d2 = d1.plusDays(10); <..>
LocalDate d2 = d1.minusDays(10);
LocalDate d2 = d1.minusMonths(2);
LocalDate d2 = d1.minusWeeks(30);
LocalDate d2 = d1.minusYears(1);

LocalDate date = LocalDate.of(2018, Month.APRIL, 30);
date.plusDays(2);
date.plusYears(3);
```

IMMUTABLE (no changes)!

```
LocalDate d2 = d.withDayOfMonth(1);
LocalDate d2 = d.withDayOfYear(1);
LocalDate d2 = d.withMonth(7);
LocalDate d2 = d.withYear(1);
```

Conversion

```
LocalDateTime dt = localDate.atTime(16, 30);
LocalDateTime dt = localDate.atTime(16, 61);→ RTE

long t = localDate.toEpochDay(); → 1970.01.01+++
```

LocalTime (23:59:59.999999999) IMMUTABLE

```
LocalTime t =
    LocalTime.of(12, 12);
    LocalTime.of(0, 12, 6);
    LocalTime.of(14, 7, 10, 998654578);
    LocalTime.parse("12:15");
    LocalTime.parse("12:5");→ RTE
    LocalTime.parse("24:00:00");→ RTE
    LocalTime.parse("12:00:23 PM");→ RTE
    LocalTime.ofSecondOfDay(66620); -> 18:30:20

LocalTime.now()
LocalTime.now(Clock.systemDefaultZone())
LocalDate.now(ZoneId.of("Asia/Tokyo");



t.getMinute();
t.getMinutes(); → fail

t2 = t.plusHours(6).plusMinutes(30).minusNanos(100);

LocalTime t = LocalTime.of(14, 10, 0);
LocalDate d = LocalDate.of(2016,02,28);
LocalDateTime dt = t.atDate(d); → 2016-02-28T14:10
```

LocalDateTime (2050-06-18T14:20:30.999999999) IMMUTABLE

```
LocalDateTime dt =
    LocalDateTime.of(2050, 6, 5, 14, 30, 0);
    LocalDateTime.parse("2050-06-05T14:00:00");
    LocalDateTime.now()

dt2 =
    dt.toLocalDate()
    dt.toLocalTime()
    dt.plusHours(2).minusDays(1);
```

Instant (2050-06-18T14:20:30.999999992)

```
- java.time.Instant
- trace the execution duration
- Unix epoch (1970-01-01T00:00:00Z) / Uniz timestamp
- internally:
        a) long for +/- seconds
        b) int for nanosecond in each second
- long/int are not very readable
- Missing: minus/plus days,months,…
- LocalDateTime uses default time zone, Instant NOT?

Instant inst = Instant.now();
    inst.getEpochSecond(); -> duration (sec)
    inst.toEpochMilli();   -> duration (ms)
```

Period (PnYnMnWnD or PnW) IMMUTABLE

```
- amount of years, months, days

Period p1 =
    Period.of(1, 2, 7);
    Period.ofYears(2);
    Period.ofMonths(5);
    Period.ofWeeks(10);
    Period.ofDays(15);
    Period.ofDays(35); → NOT 1Month+5days!
    Period.ofDays(-15); → OK

    Period.parse("p5y");
    Period.parse("+P5Y");
    Period.parse("P+5Y");
    Period.parse("-P-5Y1M"); → 5Y-1M

    Period.parse("P5y1m2d");
  W→D
    Period.parse("P2W"); → 2*7=14D
    "P1Y2M3W4D" → Period.of(1, 2, 25);
    Period.parse("-P5W"); → P-5W = P-35D

p.getMonths();
p.getWeeks(); → fail

Period p = Period.between(d1, d2); → [d1;d2) =d2-d1
```

```
LocalDateTime.parse("2052-01-31T14:18:36");
dateTime.plus(Period.ofMonths(1));
        → 2052-02-29T14:18:36
dateTime.plus(Period.ofMonths(-1));
dateTime.minus(Period.ofMonths(-1));

period.isZero();
period.isNegative(); → at least 1 negative!

period.plus(period2);
period.plusMonths(1);
period.minus(period2);
period.minusHours(2);
period.multipliedBy(2); -- all elements
period.divideBy(2); → fail
```

Period Calculations:

```
Period.of(10,5,40).toTotalMonths(); → 125 (10Y+5M)
P1M - P10D = P1M-10D (individual parts, no calc!)
P1M + P12M = P13M
-P1M1D = P-1M-1D
P1Y2M3W4D = P1Y2M25D
P1Y2M4D3W → fail, order PnYnMnWnD
```

Duration (PT7H13M42.003S)

```
- equivalent of Period
- amount of hours, minutes, seconds
- measuring machine time
- working with Instance objects
- internally: long (sec) + int (ms)

Duration between = Duration.between(start, end);
System.out.println(between); -> PT7H13M42.003S

Duration dur =
    Duration.of(3600, ChronoUnit.MINUTES) -> PT60H
    Duration.ofDays(4)        -> (4*24)=PT96H
    …
    Duration.ofSeconds(30)    -> PT30S
    Duration.ofMillis(120)    -> PT0.12S
    Duration.ofNanos(120)     -> PT0.00000012S

    Duration.parse("P2DT10H30M") -> PT58H30M
    Duration.parse("P1D") -> PT24H
    Duration.parse("P1M") -> RTE
    Duration.parse("P1Y") -> RTE
```

TemporalUnit interface (ChronoUnit)

- java.time.temporal
- enum ChronoUnit implements TemporalUnit (seconds, minutes, hours, days, months, and years)

```
Duration.of(long amount, TemporalUnit unit)

Duration.of(1,ChronoUnit.MINUTES).getSeconds() →60
Duration.of(1,ChronoUnit.HOURS).getSeconds() →3600
Duration.of(1,ChronoUnit.DAYS).getSeconds() →86400


for(ChronoUnit unit : ChronoUnit.values()) {
  System.out.printf("%10s \t %b \t\t %b \t\t %s %n",
unit, unit.isDateBased(), unit.isTimeBased(),
unit.getDuration());
}
```

ChronoUnit	DateBased	TimeBased	Duration
Nanos	false	true	PT0.000000001S
Micros	false	true	PT0.000001S
Millis	false	true	PT0.001S
Seconds	false	true	PT1S
Minutes	false	true	PT1M
Hours	false	true	PT1H
HalfDays	false	true	PT12H
Days	true	false	PT24H
Weeks	true	false	PT168H
Months	true	false	PT730H29M6S
Years	true	false	PT8765H49M12S
Decades	true	false	PT87658H12M
Centuries	true	false	PT876582H
Millennia	true	false	PT8765820H
Eras	true	false	PT87658200000000H
Forever	false	false	PT2562047788015215H30M7.999999999S

Time Zones

- java.time.*
- ZoneId, ZoneOffset, ZonedDateTime

java.time.ZoneId

- time zones (~589)
- offset from Greenwich Mean Time (GMT), also known as UTC/Greenwich

```
System.out.println(ZoneId.systemDefault());
-> Europe/Helsinki

   ZoneId.of("Asia/Kolkata");
   ZoneId.of("Europe/Helsinki");

   ZoneId.systemDefault();
```

ZonedDateTime

- date + time + zone
- 2015-11-05T11:38:40.647+05:30[Asia/Kolkata]

```
ZonedDateTime zdt =
    ZonedDateTime.of(d, t, myZone);
    ZonedDateTime.of(dt, myZone);
    dateTime.atZone(myZone);
------
ZoneId auckland = ZoneId.of("Pacific/Auckland");
zdt2 = zonedDateTime.withZoneSameInstant(auckland);
-------
Duration timeDifference = Duration.between(
  zdt1.toLocalTime(),
  zdt2.toLocalTime() );
```

java.time.ZoneOffset

- time-zone offset from UTC/Greenwich
- extends ZoneId

```
ZoneOffset ofs = zonedDateTime.getOffset(); +03:00
```

Daylight Savings

- due to seasons change
- **+/- hour**: daylight savings time (DST)

```
ZoneId myZone = ZoneId.systemDefault();
Duration dst = myZone.getRules()
   .getDaylightSavings(Instant.now()); -> PT1H

NOTE: dst.isZero() -> then zone has no effect
```

DateTimeFormatter

- **java.time.format.*;**
- reading/printing date and time values in different formats
- case-sensitive format

```
DateTimeFormatter fmt = DateTimeFormatter
 .ofPattern("yyyy-MM-dd")
 .ofPattern("y-M-d")
 .BASIC_ISO_DATE
 .ISO_DATE → YYYY-MM-dd
 .ISO_TIME → HH:mm:ss
 .ISO_DATE_TIME
 .RFC_1123_DATE_TIME
      Thu, 5 Nov 2015 11:27:22 +0530
 .ISO_ZONED_DATE_TIME
      2015-11-05T11:30:33.49+05:30[Asia/Kolkata]
  --- OS Configuration related:
 . ISO_LOCAL_DATE_TIME
 . ofLocalizedDate(FormatStyle.MEDIUM)
 . ofLocalizedTime(FormatStyle.MEDIUM)
 . ofLocalizedDateTime(
        FormatStyle.FULL, FormatStyle.SHORT)
```

Symbol	Meaning	Example
y, Y	year	2057; 57
M	month of year	8; 08; Aug; August
D	day of year	223
d	day of month	11
E	day of week	Sat
e	localized day of week	7; Sat
a	a.m. or p.m. of day	pm
h	clock hour of a.m./p.m.	03
H	hour of day	14
m	minute of hour	30
s	second of minute	15
'	escape for text	

```
M=Month, m=minute
D=Day of Year, d=day
H=24hour, h=12hour
Y=y
        YYYY-MM-dd → 2018.03.27
        YY-MM-dd → 18-03.27
        Y-M-d → 2018-3-27
HH 01, H 1
"'time now: 'HH:mm:ss"

z (time zone: general time-zone format)
k (hour: value range 1-24)
K (hour in a.m./p.m.: value range 0-11)
EEEE - Friday, E=Fri
```

Format:

```
date.format(formatter);
formatter.format(localDate);
formatter.format(localDateTime);
```

OK: Only if all components present:

```
if "yyyy MM dd": formatter.format(localTime); → RTE
if "HH:mm:ss":   formatter.format(localDate); → RTE
```

Parse:

```
LocalDate date = LocalDate.parse("2057", formatter);
LocalDate.parse("2017-01-02"); → ok, static

DateTimeFormatter.parse("2017-01-02");→ fail, inst
formatter.parse("2017-01-02"); → ok, instance
```

```
String d = LocalDate
.format(DateTimeFormatter.ISO_DATE_TIME)
.parse("2057-08-11"); → fail to compile
```

Flight Travel Example

- flight from Singapore: January 1, 2016 at 6:00a.m.
- flight to: Auckland, New Zealand
- flight takes 10 hours

- arrival time in Auckland?

```
public static void main(String[] args) {
    DateTimeFormatter dateTimeFormatter =
            DateTimeFormatter.ofPattern("dd MMM yyyy hh.mm a");

    // Leaving on 1st Jan 2016, 6:00am from "Singapore"
    ZonedDateTime departure = ZonedDateTime.of(
            LocalDateTime.of(2016, Month.JANUARY, 1, 6, 0),
            ZoneId.of("Asia/Singapore"));

    System.out.println("Departure: " + dateTimeFormatter.format(departure));

    // Arrival on the same day in 10 hours in "Auckland"
    ZonedDateTime arrival =
            departure.withZoneSameInstant(ZoneId.of("Pacific/Auckland"))
            .plusHours(10);

    System.out.println("Arrival: " + dateTimeFormatter.format(arrival));
}
```

```
Departure: 01 Jan 2016 06.00 AM
Arrival: 01 Jan 2016 09.00 PM
```

Java I/O Fundamentals (java.io)

Read/Write Console

```
java.lang.System
        in - java.io.InputStream
                -> InputStreamReader
                 -> BufferedReader
        out - java.io.PrintStream
        err - java.io.PrintStream

System.in.read() -> byte 0..255
    ➔  checked IOException
------------
java.io.*
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in)
);
String str = br.readLine();
… br = new BufferedReader(System.in); -> CE
------------
java.util.*
Scanner scanner = new Scanner(System.in);
String str = scanner.next();
```

Reassign

```
PrintStream ps = new PrintStream("log.txt");
System.setOut(ps);
System.out.println("Test output to System.out");
    ➔  checked: FileNotFoundException
```

java.io.Console

```
- helps reading/writing console data
- character input device (keyboard)
- character display device (screen)
- NULL: if JVM is invoked indirectly (IDE)

Console console = System.console();
if (console != null) {
    String txt = console.readLine();
    console.printf(txt);
}

- Reader reader()
- PrintWriter writer()
- String readLine()
    - String readLine(String fmt, Object... args)
- char[] readPassword()
   - char[] readPassword(String fmt, Object... args)
```

```
- Console format(String fmt, Object... args)
  = Console printf(String fmt, Object... args)
    ➔  unchecked: IllegalFormatException
- crux format
- console.printf("",1,x,y); -> nothing printed!
- %% escape for %
- wrong arg: IllegalFormatConversionException

  %
  [argument_index]    argument 1$,2$ , $1
    [flags]             - (left align) or 0 (pads)
    [width]
  [.precision]
  datatype_specifier

console.printf("%2$s %1$s %n", "hello", "world");
console.printf("%d %<x %<o", 10);
     -> relative (previous match): 10 10 12
```

Symbol	Description
%b	Boolean
%c	Character
%d	Decimal integer (signed)
%e	Floating point number in scientific format
%f	Floating point number in decimal format
%g	Floating point number in decimal or scientific format (depending on the value passed as argument)
%h	Hashcode of the passed argument
%n	Line separator (new line character)
%o	Integer formatted as an octal value
%s	String
%t	Date/time
%x	Integer formatted as an hexadecimal value

```
console.printf("%-15s \t %5d \t\t %d \t\t %.1f \n",
player, matches, goals,
((float)goals/(float)matches));
```

```
 %-15s          \t     %5d    \t\t   %d   \t\t     %.1f     \n
-------------------------------------------------------------------
Player                  Matches         Goals       Goals per match
-------------------------------------------------------------------
Demando                   100            122            1.2
Mushi                      80            100            1.3
Peale                     150            180            1.2
```

```
String userName = console.readLine("username: ");
char[] pass = console.readPassword("password: ");

if (new String(pass).equals("nuts").equals("x"))
…
Arrays.fill(pass, ' '); ➔ recommended
```

Read/Write Files

- streams: ordered sequence of data
- character streams:
  - 16bit
  - **Reader ==> Writer**
  - unicode,tab,newline,…
- byte streams:
  - 8bit
  - **InputStream ==> OutputStream**
  - low-level format,exe,asm,jpg…
- use Buffered* for efficiency
  - readLine
  - readInt (byte, short,int,long,double)
  - flush() - make sure that buff persisted
  - **close**()
    - includes flush()
    - wrapper closes underlying streams!
- **Don't mix:** Reader + OutputStream, … -> RTE

character streams

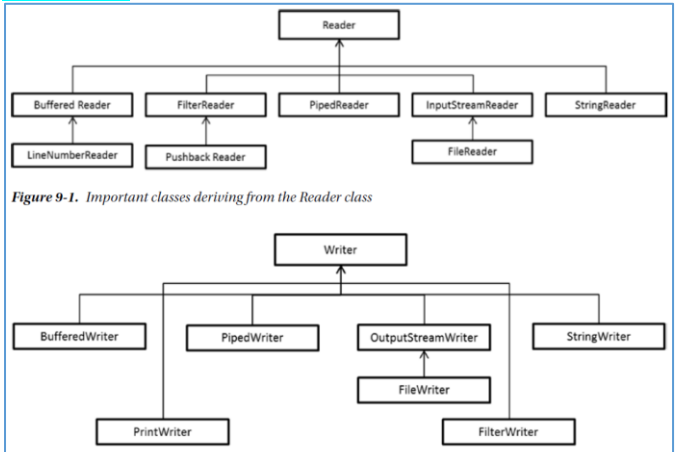


Figure 9-1. Important classes deriving from the Reader class

Figure 9-2. Important classes deriving from the Writer class

```
try (FileReader inputFile = new FileReader("D:\\Temp\\dummy.txt")) {
    int ch = 0;
    while ((ch = inputFile.read()) != -1) {
        System.out.print((char) ch);
    }
} catch (FileNotFoundException fnfe) {
    System.err.printf("Cannot open the given file");
} catch (IOException ioe) {
    System.err.printf("Error when processing file");
}
```

➔ OR inputFile.read()

➔ End-Of-File (EOF) = -1

Reading/Writing Text Files

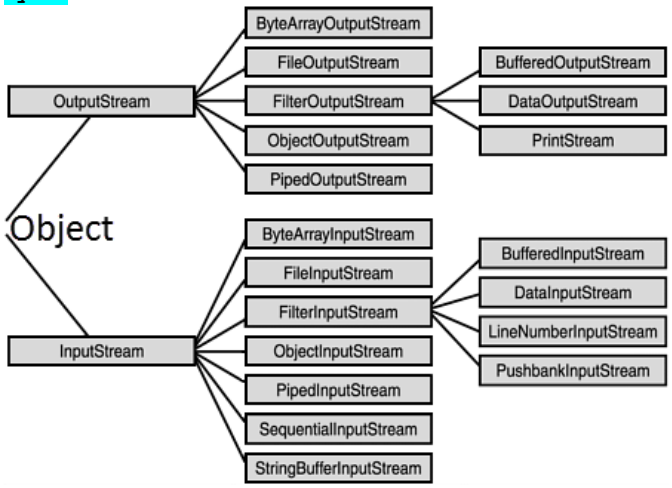
- use Buffered* for efficiency
- close() calls flush()

```
String srcFile = "D:\\Temp\\dummy.txt";
String dstFile = "D:\\Temp\\dummy2.txt";
try (BufferedReader inputFile = new BufferedReader(new FileReader(srcFile));
     BufferedWriter outputFile = new BufferedWriter(new FileWriter(dstFile))) {
    int ch = 0;
    while ((ch = inputFile.read()) != -1) {
        outputFile.write((char) ch);
    }
} catch (FileNotFoundException fnfe) {
    System.err.printf("Cannot open the given file");
} catch (IOException ioe) {
    System.err.printf("Error when processing file");
}
```

"Tokenizing Text"

```
try (Scanner tokenizingScanner = new Scanner(new FileReader("D:\\Temp\\dummy.txt"))) {
    tokenizingScanner.useDelimiter("\\W");
    while (tokenizingScanner.hasNext()) {
        System.out.println(tokenizingScanner.next());
    }
} catch (FileNotFoundException fnfe) {
    System.err.printf("Cannot open the given file");
}
```

byte streams



Class name	Short description
PipedInputStream, PipedOutputStream	PipedInputStream and PipedOutputStream create a communication channel on which data can be sent and received. PipedOutputStream sends the data and PipedInputStream receives the data sent on the channel.
FileInputStream, FileOutputStream	FileInputStream receives a byte stream from a file, FileOutputStream writes a byte stream into a file.
FilterInputStream, FilterOutputStream	These filtered streams are used to add functionalities to plain streams. The output of an InputStream can be filtered using FilterInputStream. The output of an OutputStream can be filtered using FilterOutputStream.
BufferedInputStream, BufferedOutputStream	BufferedInputStream adds buffering capabilities to an input stream. BufferedOutputStream adds buffering capabilities to an output stream.
PushbackInputStream	A subclass of FilterInputStream, it adds "pushback" functionality to an input stream.
DataInputStream, DataOutputStream	DataInputStream can be used to read java primitive data types from an input stream. DataOutputStream can be used to write Java primitive data types to an output stream.

```
String fileName = "d:\\temp\\dummy.jpg";

byte[] magicNumber = { (byte) 0xCA, (byte) 0xFE, (byte) 0xBA, (byte) 0xBE };

try (FileInputStream fis = new FileInputStream(fileName)) {
    byte[] u4buffer = new byte[4];
    if (fis.read(u4buffer) != -1) {
        if (Arrays.equals(magicNumber, u4buffer)) {
            System.out.printf("Matched");
        } else {
            System.out.printf("Not matched");
        }
    }
} catch (FileNotFoundException fnfe) {
    System.err.println("file does not exist with the given file name ");
} catch (IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
}
```

```
BufferedInputStream bis = new
BufferedInputStream(new FileInputStream(fileName));
```

data stream

```
DataOutputStream dos = new DataOutputStream(new
FileOutputStream("temp.data"));
for(int i = 0; i < 10; i++) {
   dos.writeByte(i);
   dos.writeShort(i);
   dos.writeInt(i);
   dos.writeLong(i);
   dos.writeFloat(i);
   dos.writeDouble(i);
}
…
DataInputStream dis = new DataInputStream(new
FileInputStream("temp.data"))
for(int i = 0; i < 10; i++) {
  System.out.printf("%d %d %d %d %g %g %n",
    dis.readByte(),
    dis.readShort(),
    dis.readInt(),
    dis.readLong(),
    dis.readFloat(),
    dis.readDouble());
}
```

Object stream

- ObjectInputStream/ObjectOutputStream
- serialization (metadata + data)
- Map<String, String> lost on serialization (type erasure)
- Map<?,?> wildcards

```
Map<String, String> presidentsOfUS = new HashMap<>();
presidentsOfUS.put("Barack Obama", "2009 to --, Democratic Party, 56th term");
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("object.data"))) {
    oos.writeObject(presidentsOfUS);
} catch (FileNotFoundException fnfe) {
    System.err.println("cannot create a file with the given file name ");
} catch (IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
}

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("object.data"))) {
    Object obj = ois.readObject();
    if (obj != null && obj instanceof Map) {
        Map<?, ?> presidents = (Map<?, ?>) obj;
        System.out.println("President name \t Description");
        for (Map.Entry<?, ?> president : presidents.entrySet()) {
            System.out.printf("%s \t %s %n", president.getKey(), president.getValue());
        }
    }
} catch (FileNotFoundException fnfe) {
    System.err.println("cannot create a file with the given file name ");
} catch (IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
} catch (ClassNotFoundException cnfe) {
    System.err.println("cannot recognize the class of the object - is the file  corrupted?");
}
```

Customize serialization:

```
public class User implements Serializable {

private void readObject(
  ObjectInputStream aInputStream)
throws ClassNotFoundException, IOException  {
  firstName = aInputStream.readUTF();
  lastName = aInputStream.readUTF();
  accountNumber = aInputStream.readInt();
  dateOpened = new Date(aInputStream.readLong());
}

private void writeObject(
   ObjectOutputStream aOutputStream)
throws IOException {
  aOutputStream.writeUTF(firstName);
  aOutputStream.writeUTF(lastName);
  aOutputStream.writeInt(accountNumber);
  aOutputStream.writeLong(dateOpened.getTime());
}

}
```

- JVM checks for the duplicate object
- if already serialized, then reference to it

```
oos.writeObject(usPresident);
usPresident.setAge(70);
oos.writeObject(usPresident);
```

Java File I/O (NIO.2)

- operations related to a file system
- Stream API + NIO.2
- **Path**: operate on file/directory (could be unreal!)
- **Files**: create, move, copy, delete files
- File - old J4

Path Interface

- programming abstraction to represent dir/file
- file system as a tree
- **symbolic links**:
  - link/**reference** to another file (real/unreal)
  - transparent to the user
- absolute path (starts from root): C:\Temp\a.txt
- relative path: ..\a.txt
- **Path**: directories and files
  - no constructor/get instance: Paths.get("f");
  - equals() - true/false (if Path equals)
    NOTE: do normalize/absolute before check
  - compareTo() - 1/0/+1 sorting
- some methods: not require real file

```
java.io.File: toPath() -> java.nio.file.Path
java.nio.Path: toFile() -> java.io.File
```

Method	Description
Path getRoot()	Returns a Path object representing the root of the given path, or null if the path does not have a root.
Path getFileName()	Returns the file name or directory name of the given path. Note that the file/directory name is the last element or name in the given path.
Path getParent()	Returns the Path object representing the parent of the given path, or null if no parent component exists for the path.
int getNameCount()	Returns the number of file/directory names in the given path; returns 0 if the given path represents the root.
Path getName(int index)	Returns the *i*th file/directory name; the index 0 starts from closest name to the root.
Path subpath(int beginIndex, int endIndex)	Returns a Path object that is part of this Path object; the returned Path object has a name that begins at beginIndex and ends with the element at index endIndex - 1. In other words, beginIndex is inclusive of the name in that index and exclusive of the name in endIndex. This method may throw IllegalArgumentException if beginIndex is >= number of elements, or endIndex <= beginIndex, or endIndex > number of elements.
Path normalize()	Removes redundant elements in the path, such as . (dot symbol that indicates the current directory) and .. (double-dot symbol that indicates the parent directory).
Path resolve(Path other) Path resolve(String other)	Resolves a path against the given path. For example, this method can combine the given path with the other path and return the resulting path.
Boolean isAbsolute()	Returns true if the given path is an absolute path; returns false if not (when the given path is a relative path, for example).
Path startsWith(String path) Path startsWith(Path path)	Returns true if this Path object starts with the given path, or false otherwise.
Path toAbsolutePath()	Returns the absolute path.

```
Path testFilePath = Paths.get("D:\\test\\testfile.txt");   could be not real!

System.out.println("Printing file information: ");
System.out.println("\t file name: " + testFilePath.getFileName());
System.out.println("\t root of the path: " + testFilePath.getRoot());
System.out.println("\t parent of the target: " + testFilePath.getParent());

for (Path element : testFilePath) {
    System.out.println("path element: " + element);
}
```

```
Path x = Paths.get("D:\test\testfile.txt");-> RTE
```

```
Path testFilePath = Paths.get(".\\Test");
System.out.println("The file name is: " + testFilePath.getFileName());
System.out.println("Its URI is: " + testFilePath.toUri());
System.out.println("Its absolute path is: " + testFilePath.toAbsolutePath());
System.out.println("Its normalized path is: " + testFilePath.normalize());

Path testPathNormalized = Paths.get(testFilePath.normalize().toString());
System.out.println("Its normalized absolute path is: "
+ testPathNormalized.toAbsolutePath());

try {
    System.out.println("Its normalized real path is: "
+ testFilePath.toRealPath(LinkOption.NOFOLLOW_LINKS));
} catch (IOException e) {
    e.printStackTrace();
}
```

```
The file name is: Test
Its URI is: file:///C:/Users/dm/github/ws_oca/oca/./Test
Its absolute path is: C:\Users\dm\github\ws_oca\oca\.\Test
Its normalized path is: Test
Its normalized absolute path is: C:\Users\dm\github\ws_oca\oca\Test
Exception in thread "main" java.nio.file.NoSuchFileException: C:\Users\dm\github\ws_oca\oca\Test
        at sun.nio.fs.WindowsException.translateToIOException(WindowsException.java:79)
        at sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:90)
        at sun.nio.fs.WindowsLinkSupport.getRealPath(WindowsLinkSupport.java:259)
        at sun.nio.fs.WindowsPath.toRealPath(WindowsPath.java:836)
        at sun.nio.fs.WindowsPath.toRealPath(WindowsPath.java:44)
        at oca.Test.main(Test.java:27)
```

```
Path dirName =
      Paths.get("D:\\OCPJP\\programs\\NIO2\\");

Path resolvedPath = dirName.resolve("Test");
System.out.println(resolvedPath);
    ➔  D:\OCPJP\programs\NIO2\Test
```

```
Path aPath = Paths.get("D:\\A\\xx\\..\\B\\C\\.\\D");
aPath = aPath.normalize();
System.out.println(aPath.subpath(2, 3));  [ ; )
-> D:\A\B\C\D
    0 1 2 3     ➔ C
```

Files Class

- Files utility class J7+
- No constructors, only static methods
- create/copy/delete dir/file/symbolic links
- create streams dir/byte channels
- attributes/metadata of file
- walk tree
- throws checked: **IOException**

Method	Description
Path createDirectory(Path dirPath, FileAttribute<?>... dirAttrs) Path createDirectories(Path dir, FileAttribute<?>... attrs)	Creates a file given by the dirPath, and sets the attributes given by dirAttributes. May throw an exception such as FileAlreadyExistsException or UnsupportedOperationException (for example, when the file attributes cannot be set as given by dirAttrs). The difference between createDirectory and createDirectories is that createDirectories creates intermediate directories given by dirPath if they are not already present.
Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)	Creates a temporary file with the given prefix, suffix, and attributes in the directory given by dir.
Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)	Creates a temporary directory with the given prefix and directory attributes in the path specified by dir.
Path copy(Path source, Path target, CopyOption... options)	Copies the file from source to target. CopyOption can be REPLACE_EXISTING, COPY_ATTRIBUTES, or NOFOLLOW_LINKS. Can throw exceptions such as FileAlreadyExistsException.
Path move(Path source, Path target, CopyOption... options)	Similar to the copy operation, but the source file is removed. If the source and target are in the same directory, it is a file-rename operation.
boolean isSameFile(Path path, Path path2)	Checks whether the two Path objects locate the same file.
boolean exists(Path path, LinkOption... options)	Checks whether a file/directory exists in the given path; can specify LinkOption.NOFOLLOW_LINKS to not to follow symbolic links.
Boolean isRegularFile(Path path, LinkOption...)	Returns true if the file represented by path is a regular file.
Boolean isSymbolicLink(Path path)	Returns true if the file represented by path is a symbolic link.
Boolean isHidden(Path path)	Return true if the file represented by path is a hidden file.
long size(Path path)	Returns the size of the file represented by path in bytes.
UserPrincipal getOwner(Path path, LinkOption...), Path setOwner(Path path, UserPrincipal owner)	Gets/sets the owner of the file.
FileTime getLastModifiedTime(Path path, LinkOption...), Path setLastModifiedTime(Path path, FileTime time)	Gets/sets the last modified time for the specified file.
Object getAttribute(Path path, String attribute, LinkOption...), Path setAttribute(Path path, String attribute, Object value, LinkOption...)	Gets/sets the specified attribute of the specified file.

```
Files.isSameFile(path1, path2);
    ➔  (could: NoSuchFileException)
Files.exists(path);
Files.exists(path, LinkOption.NOFOLLOW_LINKS)
Files.isDirectory(path, LinkOption.NOFOLLOW_LINKS)
```

Check Attributes:

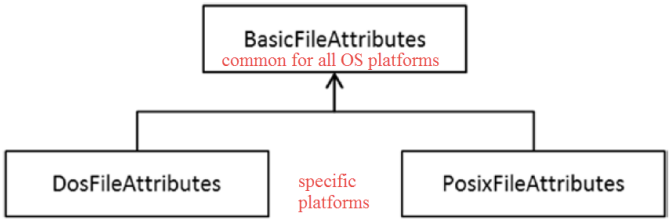
```
Object object
= Files.getAttribute(path, "creationTime",
                     LinkOption.NOFOLLOW_LINKS);
System.out.println("Creation time: " + object);
```

- creationTime
- lastModifiedTime
- size
- dos:hidden -> view:attribute
- isHidden
- isDirectory

➔ throws runtime **IllegalArgumentException**

```
Files.isReadable(path)
Files.isWritable(path)
Files.isExecutable(path)
```

```
Map<String,Object> readAttributes(
    Path path,
    String attributes,          -> "*" or "dos:*"
    LinkOption... options)

<A extends BasicFileAttributes> A readAttributes(
    Path path,
    Class<A> type,
    LinkOption... options)
```




```
import java.nio.file.attribute.BasicFileAttributes;
…
BasicFileAttributes fileAttributes =
Files.readAttributes(path,BasicFileAttributes.class);
```

➔ throws: IOException

```
fileAttributes
        .size();
        .isRegularFile();
        .lastAccessTime();
        .isSymbolicLink();
```

Copy file:

- All path directories must exists
  -> throws NoSuchFileException
- copy dir will **not copy** sub-dirs/files (need explicit copy of file/dir)
- symbolic links: copied as **target files**

```
Path pathSource = Paths.get(args[0]);
Path pathDestination = Paths.get(args[1]);
try {
  Files.copy(pathSource, pathDestination);
} catch (IOException e) {
   e.printStackTrace();
}
```

```
Files.copy(
    pathSource,
    pathDestination,
    StandardCopyOption.REPLACE_EXISTING
);
```

Copy streams from/to file:

```
copy(InputStream, Path, CopyOptions...) and
copy(Path, OutputStream, CopyOptions...)
```

Move file:

- symbolic links: copied as **links**
- ATOMIC_MOVE success all/rollback all

```
Files.move(
    pathSource,
    pathDestination,
    StandardCopyOption.REPLACE_EXISTING
);
```

Delete File:

- **Empty** directory only
- if not empty: FAIL
- if read-only: FAIL
- symbolic links: deleted **target files**
- throws NoSuchFileException/IOException

```
Files.delete(pathSource);
Files.deleteIfExists(pathSource);
```

Stream API with NIO.2

Files.list

- flat list (not recursively)
- throws: IOException
- must close()

```
try(Stream<Path> x = Files.list(Paths.get("."))) {
    x.forEach(System.out::println);
}
```

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

Files.walk

- recursive walk (traversed)
- throws: IOException
- must close()

```
Files.walk(Paths.get("."))
   .forEach(System.out::println);
```

Options:

```
Files.walk(Paths.get("."),FileVisitOption.FOLLOW_LINKS)
   .forEach(System.out::println);
```

Limit depth:

```
Files.walk(Paths.get("."), 4)
   .forEach(System.out::println);
```

Files.find

- recursive walk (traversed)
- must close()

```
BiPredicate<Path, BasicFileAttributes> predicate
    = (path, attrs) -> path.toString().endsWith("class")
                    && attrs.isRegularFile();

try (Stream<Path> entries
        = Files.find(Paths.get("."), 4, predicate)) {
    entries.limit(100).forEach(System.out::println);
}
```

Files.lines

- must close()

```
try (Stream<String> lines =
        Files.lines(Paths.get("d:\\temp\\dummy.txt"))) {
    lines.forEach(System.out::println);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

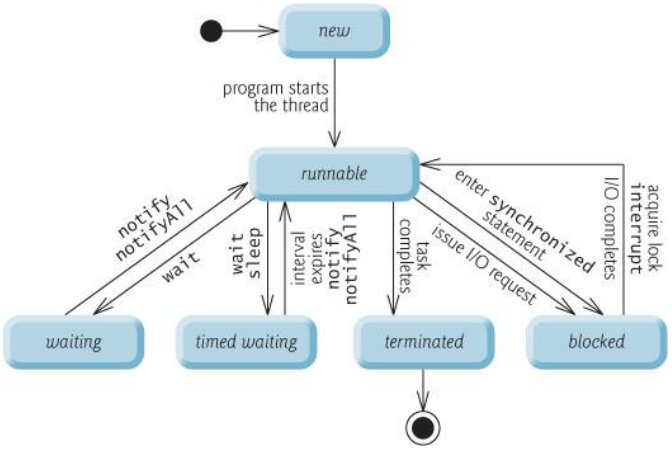
Concurrency

- multi-threading
- running together/parallel
- can improve performance (if properly used)
- Java: low/high-level threads management
- Object class, Thread class, Runnable interface
- main() method is executed as thread
- Runtime.getRuntime().availableProcessors()

Thread Class / Runnable Interface

- Thread.run() – does nothing, should be overriden
- myThread.run() -> will lock current T. Use start()
- Thread.start() -> **JVM Call:** Thread.run()

Method	Method Type	Short Description
Thread currentThread()	Static method	Returns reference to the current thread.
String getName()	Instance method	Returns the name of the current thread.
int getPriority()	Instance method	Returns the priority value of the current thread.
void join(), void join(long), void join(long, int)	Overloaded instance methods	The current thread invoking join on another thread waits until the other thread completes. You can optionally give the timeout in milliseconds (given in long) or timeout in milliseconds as well as nanoseconds (given in long and int).
void run()	Instance method	Once you start a thread (using the start() method), the run() method will be called when the thread is ready to execute.
void setName(String)	Instance method	Changes the name of the thread to the given name in the argument.
void setPriority(int)	Instance method	Sets the priority of the thread to the given argument value.
void sleep(long) void sleep(long, int)	Overloaded static methods	Makes the current thread sleep for given milliseconds (given in long) or for given milliseconds and nanoseconds (given in long and int).
void start()	Instance method	Starts the thread; JVM calls the run() method of the thread.
String toString()	Instance method	Returns the string representation of the thread; the string has the thread's name, priority, and its group.



extend Thread

- Thread implements Runnable

```
public class Test {

    public static void main(String args[]) {
        Thread myThread = new MyThread();
        myThread.start();                          run() called by JVM
        System.out.println("name: " +
        Thread.currentThread().getName()); // main
    }

}

class MyThread extends Thread {
    public void run() {                    if run() is not defined,
        try {                              then Thread does nothing
            sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("name: " + getName()); // Thread-0
    }

}
```

implements Runnable

```
public class Test {
    public static void main(String args[]) throws Exception {
        Thread myThread = new Thread(new RunnableImpl());
        myThread.start();
    }
}

class RunnableImpl implements Runnable {
    public void run() {
        // Implementation
    }
}
```

synchronized

- lock on entire object (not on critical section)!
- **slower** than Atomic*
- employ mutexes
- synchronized blocks
- synchronized methods
- to avoid the problem of **race conditions**:
  - data race/race hazard
  - avoid unintuitive results
  - all variables in memory (+copied locally)
  - multiple threads could update variable at same time
- **critical section**:
  - code commonly accessed by multiple threads
- **atomic**:
  - 1 thread can acquire lock on object at a time, and only that thread can execute critical section
  - thread-safe

synchronized blocks

```
synchronized(this) { }          -> non-static class

public static void main(String args[]) {
    synchronized(this) { }      -> CE static (no this)
}
synchronized() { }              -> CE Object missing
synchronized(myInt) { }         -> CE primitive
synchronized(nullRef) { }       -> RTE null

synchronized(this) {
   throw new Exception(); → Will release lock!
}
```

synchronized methods

```
public synchronized void assign(int i) { }
 = public void assign(int i) {synchronized(this) {}}

public static synchronized void assign(int i) { }
    -> synchronized(MyClass.class) { }
```

synchronized in constructors:

```
public synchronized MyClass(){} -> CE constructor
  - JVM ensures that only one thread can invoke
a constructor call (no need to declare a constructor
synchronized)

public MyClass() {
   synchronized(this) { } -> OK
}
```

lock on entire object (not on critical section):

```
  Integer x = 1;
  synchronized(x) { }
  synchronized(x) { }
```

Threading Problems

Name	Description	Solution
Race conditions	Same variable modified by multiple threads	- use Atomic* - synchronized block - synchronized method
Deadlocks	Wait indefinitely for others to terminate (r1->r2, r2->r1) No progress	- cannot consistently reproduce - avoid acquiring multiple locks - acquire lock in same order (not opposite)
Livelock	Keep executing a task (lots of work is getting done) but no progress is made. Туда-сюда-обратно.	Out of scope OCJP8
Lock Starvation	Low-prio threads are not executed "starved", because of high-prio tasks amount	

java.util.concurrent.atomic.*

- efficient for primitive operations
- **faster** than synchronized
- implements Serializable

extends Number:

- AtomicInteger — int
- AtomicLong — long

extends Object:

- AtomicBoolean — Boolean
- AtomicIntegerArray — int[]
- AtomicLongArray — long[]
- AtomicReference<V> — reference of type V
- AtomicReferenceArray<E> — array of E (baseclass)

Method	Short Description
`AtomicInteger()`	Creates an instance of `AtomicInteger` with initial value 0.
`AtomicInteger(int initVal)`	Creates an instance of `AtomicInteger` with initial value `initVal`.
`int get()`	Returns the integer value held in this object.
`void set(int newVal)`	Resets the integer value held in this object to `newVal`.
`int getAndSet(int newValue)`	Returns the current `int` value held in this object and sets the value held in this object to `newVal`.
`boolean compareAndSet(int expect, int update)`	Compares the `int` value of this object to the `expect` value, and if they are equal, sets the `int` value of this object to the `update` value.
`int getAndIncrement()`	Returns the current value of the integer value in this object and increments the integer value in this object. Similar to the behavior of `i++` where `i` is an `int`.
`int getAndDecrement()`	Returns the current value of the integer value in this object and decrements the integer value in this object. Similar to the behavior of `i--` where `i` is an `int`.
`int getAndAdd(int delta)`	Returns the integer value held in this object and adds given delta value to the integer value.
`int incrementAndGet()`	Increments the current value of the integer value in this object and returns that value. Similar to the behavior of `++i` where `i` is an `int`.
`int decrementAndGet()`	Decrements the current integer value in this object and returns that value. Similar to behavior of `--i` where `i` is an `int`.
`int addAndGet(int delta)`	Adds the `delta` value to the current value of the integer in this object and returns that value.

`int intValue()`
`long longValue()`
`float floatValue()`
`double doubleValue()`
	Casts the current `int` value of the object and returns it as `int`, `long`, `float`, or `double` values.

java.util.concurrent.* high-level synchronizers

- **synchronizers:** high-level abstractions for synchronizing activities:
  - **Semaphore:** controls access to shared resources
  - **CountDownLatch:** N threads wait for countdown
  - **Exchanger:** exchanging data between two threads
  - **Phaser:** work in phases to complete a task
  - **CyclicBarrier:** wait N threads to reach sync point

CyclicBarrier

Method	Short Description
`CyclicBarrier(int numThreads)`	Creates a `CyclicBarrier` object with the number of threads waiting on it specified. Throws `IllegalArgumentException` if `numThreads` is negative or zero.
`CyclicBarrier(int parties, Runnable barrierAction)`	Same as the previous constructor; this constructor additionally takes the thread to call when the barrier is reached.
`int await()` `int await(long timeout, TimeUnit unit)`	Blocks until the specified number of threads have called `await()` on this barrier. The method returns the arrival index of this thread. This method can throw an `InterruptedException` if the thread is interrupted while waiting for other threads or a `BrokenBarrierException` if the barrier was broken for some reason (for example, another thread was timed-out or interrupted).The overloaded method takes a time-out period as an additional option; this overloaded version throws a `TimeoutException` if all other threads aren't reached within the time-out period.
`boolean isBroken()`	Returns true if the barrier is broken. A barrier is broken if at least one thread in that barrier was interrupted or timed-out, or if a barrier action failed throwing an exception.
`void reset()`	Resets the barrier to the initial state. If there are any threads waiting on that barrier, they will throw the `BrokenBarrier` exception.

```
class MixedDoubleTennisGame extends Thread {
    public void run() {
        System.out.println("All players ready!");
    }
}

class Player extends Thread {
    CyclicBarrier waitPoint;

    public Player(CyclicBarrier barrier, String name) {
        this.setName(name);
        waitPoint = barrier;
        this.start();
    }

    public void run() {
        System.out.println("Player " + getName() + " is ready ");
        try {
            waitPoint.await(); // inform that +1 arrived
        } catch (BrokenBarrierException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Test {
    public static void main(String[] args) {
        System.out.println("Waiting for all");
        CyclicBarrier barrier = new CyclicBarrier(4, new MixedDoubleTennisGame());
        new Player(barrier, "G I Joe");
        new Player(barrier, "Dora");
        new Player(barrier, "Tintin");
        new Player(barrier, "Barbie");
    }
}
```

```
Waiting for all
Player G I Joe is ready
Player Dora is ready
Player Tintin is ready
Player Barbie is ready
All players ready!
```

java.util.concurrent.* Collections

Class/Interface	Short Description
BlockingQueue	This interface extends the Queue interface. In BlockingQueue, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
ArrayBlockingQueue	This class provides a fixed-sized array based implementation of the BlockingQueue interface.
LinkedBlockingQueue	This class provides a linked-list-based implementation of the BlockingQueue interface.
DelayQueue	This class implements BlockingQueue and consists of elements that are of type Delayed. An element can be retrieved from this queue only after its delay period.
PriorityBlockingQueue	Equivalent to java.util.PriorityQueue, but implements the BlockingQueue interface.
SynchronousQueue	This class implements BlockingQueue. In this container, each insert() by a thread waits (blocks) for a corresponding remove() by another thread and vice versa.
LinkedBlockingDeque	This class implements BlockingDeque where insert and remove operations could block; uses a linked-list for implementation.
ConcurrentHashMap	Analogous to Hashtable, but with safe concurrent access and updates.
ConcurrentSkipListMap	Analogous to TreeMap, but provides safe concurrent access and updates.
ConcurrentSkipListSet	Analogous to TreeSet, but provides safe concurrent access and updates.
CopyOnWriteArrayList	Similar to ArrayList, but provides safe concurrent access. When the container is modified, it creates a fresh copy of the underlying array.
CopyOnWriteArraySet	A Set implementation, but provides safe concurrent access and is implemented using CopyOnWriteArrayList. When the container is modified, it creates a fresh copy of the underlying array.

RTE ConcurrentModificationException

underlying container has changed when it is iterating over the elements in the container

```
List<String> aList = new ArrayList<>();
aList.add("one");
Iterator listIter = aList.iterator();
while (listIter.hasNext()) {
    System.out.println(listIter.next());
    aList.add("two");
}   java.util.ConcurrentModificationException
```

```
List<String> aList = new CopyOnWriteArrayList<>();
aList.add("one");                                          OK
Iterator listIter = aList.iterator();
while (listIter.hasNext()) {
    System.out.println(listIter.next());
    aList.add("two"); not printed! Old Iterator
}
System.out.println(aList);      [one, two]
```

CopyOnWriteArrayList Class

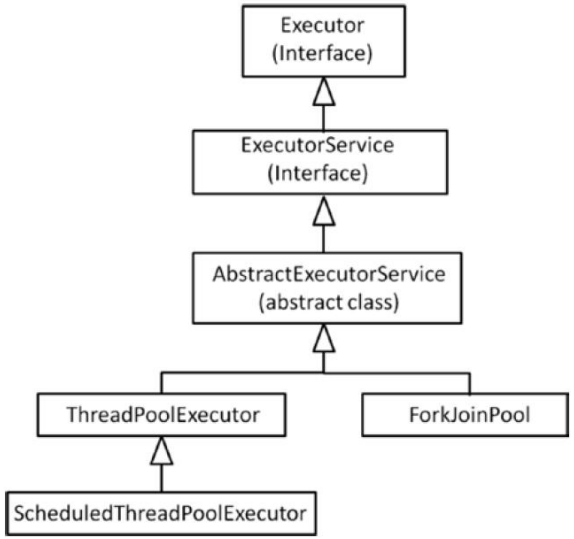
- thread safe ArrayList
- diffs:

	ArrayList	CopyOnWriteArrayList
remove(), add(), set()	May throw ConcurrentModificationException	OK
iteraror(), then remove()		UnsupportedOperationException

```
List<Integer> x= new CopyOnWriteArrayList<>(); -> OK
ArrayList<Integer> x= new CopyOnWriteArrayList<>();
    -> CE Not related classes! Don't have IS-A!
       Implemented Interfaces:
       Iterable<E>, Collection<E>, List<E>
```

Callable and ExecutorService Interfaces

- concurrently execute tasks
- abstract away the low-level details



Executor interface

- one method (functional interface)
- should create Thread and execute
- decide how/when to execute

```
class MyExecutor implements Executor {
  public void execute(Runnable runnable) {
    new Thread(runnable).start();
  }
}
```

Callable and ExecutorService

- Callable<V> interface: V call();
- Future<V>: V get(),isDone()
- Executors.newSingleThreadExecutor()
- Executors.newFixedThreadPool(nThreads)

```
public static void main(String[] args) {
    try {
        ExecutorService es = Executors.newSingleThreadExecutor();
        Future<Integer> x = es.submit(() -> {
            System.out.println("A");
            return 1;
        });
        es.shutdown();
        System.out.println(x.get());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }

}
```

```
public static void main(String[] args) {
    try {
        ScheduledExecutorService es = Executors.newScheduledThreadPool(4);
        // periodic: starts after 1s, then every 3s
        es.scheduleAtFixedRate(() -> System.out.println("Periodic"), 1, 3, TimeUnit.SECONDS);
        // delay: 2 sec
        es.schedule(() -> System.out.println("Once delayed"), 2, TimeUnit.SECONDS);
        Future<Integer> x = es.submit(() -> {
            System.out.println("Now");
            return 1;
        });
        //es.shutdown();
        System.out.println(x.get());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

infinite loop

```
Now
1
Periodic
Once delayed
Periodic
```

```
public static void main(String[] args) {
    try {
        ExecutorService es = Executors.newFixedThreadPool(2);
        Future<Integer> x = es.submit(() -> {
            System.out.println("A");
            return 1;
        });
        es.shutdown();
        System.out.println(x.get());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }

}
```

```java
class Test {
    public static void main(String[] args) {
        Callable<Long> task = new MyTask(1);
        ExecutorService es = Executors.newSingleThreadExecutor();
        Future<Long> future = es.submit(task);

        try {                                   future.isDone()
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        es.shutdown();
    }
}

class MyTask implements Callable<Long> {
    long n;
    public MyTask(long n) {
        this.n = n;
    }
    public Long call() throws Exception {
        return n * 2;
    }
}
```

OR async:

```java
Future<Integer>x = pool.submit(new MyTask(1));
try {
    System.out.println(x.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

Parallel Fork/Join Framework

- high-level implementation (abstracts low-level)
- easier to use
- more effienent than low-level custome impl.
- has a work queue via Deque (always busy). Work-stealing algorithm
- **forking**: divide task into smaller
- **joining**: merging results from smaller tasks
- suitable, when:
  - problem could be sub-divided
  - subdivided results could be combined together
  - independent subdivided tasks
- if recursion:
  - RecursiveTask (with result)
  - RecursiveAction(wo result)
- compute():
  - calculate in one go?
  - or split into sub-tasks?
- invokeAll() / fork():
  - when sub-task return value
- join():
  - get computed results (if fork() used)
- don't flood with parallel tasks
- placement matters!
- performance not guaranteed (check for placements)

- **ForkJoinPool**: executes and manage lifecycle of ForkJoinTask

Method	Short Description
`void execute(ForkJoinTask<?> task)`	Executes a given task asynchronously.
`<T> T invoke(ForkJoinTask<T> task)`	Executes the given task and returns the computed result.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`	Executes all the given tasks and returns a list of future objects when all the tasks are completed.
`boolean isTerminated()`	Returns `true` if all the tasks are completed.

`int getParallelism()`
`int getPoolSize()`
`long getStealCount()`
`int getActiveThreadCount()`
	These are status-checking methods.

`<T> ForkJoinTask<T> submit(Callable<T> task)`
`<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)`
`ForkJoinTask<?> submit(Runnable task)`
`<T> ForkJoinTask<T> submit(Runnable task, T result)`
	These methods are executing a submitted task. Overloaded versions take different types of tasks; returns a Task object or a Future object.

- **ForkJoinTask**: lightweight thread-like entity representing a task. Imlements: fork(), join()

Method	Short Description
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel the execution of the task.
`ForkJoinTask<V> fork()`	Executes the task asynchronously.
`V join()`	Returns the result of the computation when the computation is done.
`V get()`	Returns the result of the computation; waits if the computation is not complete.
`V invoke()` `static <T extends ForkJoinTask<?>> Collection<T> invokeAll(Collection<T> tasks)`	Starts the execution of the submitted tasks; waits until computation complete, and returns results.
`boolean isCancelled()`	Returns `true` if the task is cancelled.
`boolean isDone()`	Returns `true` if the task is completed.

- **RecursiveTask**<V>: V **compute()**
- **RecursiveAction**: void **compute()** - without return

Fork/Join Usage

- divide-and-conquer problems
- work-stealing algorithm
  - if free, take work from other thread)
  - load balancing thread (minimal sync costs)
- Don't flood with parallel tasks! Do:
  a) calculation for the first part in current task
  b) fork()+join() for second part
- Placement matters:

```
first.fork();
resultFirst = first.join();
resultSecond = second.compute();
  -> not progressing, because WAIT for first.join
```

```
class Test {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Integer a = pool.invoke(new MyTask(1));
        System.out.println(a);
    }
}

class MyTask extends RecursiveTask<Integer> {
    final int startValue;

    public MyTask(int startValue) {
        this.startValue = startValue;
    }

    @Override
    protected Integer compute() {
        if (startValue < 5) {
            MyTask a = new MyTask(startValue + 1);
            a.fork();
            return startValue + a.join();
        } else {
            return startValue;
        }
    }

}
```

Parallel Streams

- internally use: fork/join framework
- steps:
  - **split** elements into multiple chunks
  - process chunk in **separate thread**
  - **combine** (if neede)
- **avoid**: global state dependency ("side-effect")
- **not guarantee** performance improvement.
- Use when:
  - stateless and independent tasks
  - data structures are efficiently splittable
  - large number of elements
- by default: threads=CPUs

Make Parallel

```
Collection.stream()              - sequential
Collection.parallelStream()      - parallel

Set s; s.parallelStream() -> OK
Arrays.parallelStream(list) -> CE no such method

LongStream.rangeClosed(2, 100_000)
    .parallel()
    .filter(PrimeNumbers::isPrime)
    .count();
```

Make Parallel <-> Sequential

```
mySequential.parallel()          -> parallel
myParallelStream.sequential()    -> sequential
```

Check Type

```
Arrays.asList(1, 2, 3, 4, 5)
    .parallelStream()
    .filter(i -> (i % 2) == 0)
    .isParallel();          -> true

Arrays.asList(1, 2, 3, 4, 5)
    .parallelStream()
    .filter(i -> (i % 2) == 0)
    .sequential()
    .isParallel();          -> false
```

Avoid: global state dependency ("side-effect")

```
String words[] = "A B C".split(" ");
```

BAD:

```
class StringConcatenator {
  public static String result = "";
  public static void concatStr(String str) {
    result = result + " " + str;
  }
}
…
Arrays.stream(words)
  .parallel()
  .forEach(StringConcatenator::concatStr);
System.out.println(StringConcatenator.result);
        -> B A C
```

GOOD:

```
Optional<String> s =
  Arrays.stream(words)
   .parallel()
   .reduce((a, b) -> a + " " + b));
System.out.println(s.get());          -> A B C
```

Modify parallelism

- by default: threads=CPUs
- Runtime.getRuntime().availableProcessors()

Get default parallelism:

```
ForkJoinPool.commonPool().getParallelism();
```

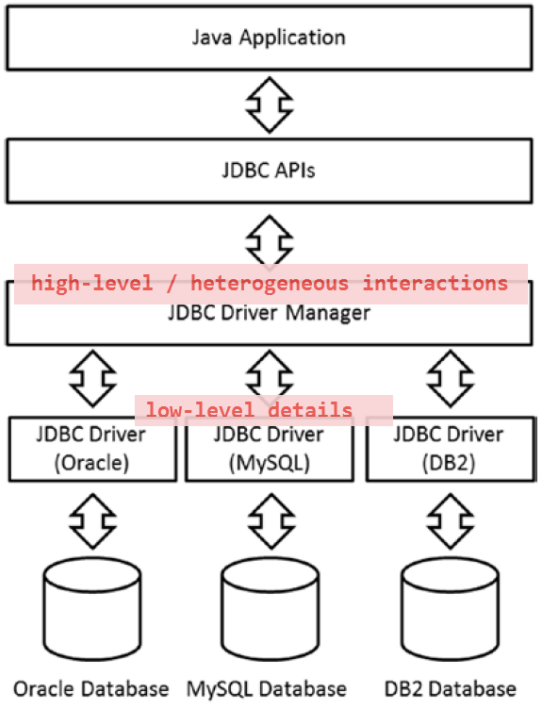
Set detault parallelism:

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "8");
```

```
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=8
GetParallelism
```

JDBC 4.2

- Java Database Connectivity API 4.2
- java.sql.*, javax.sql.*
- seamless integration with DB
- vendor agnostic: not writing program for specific database
- relational DBs only (NoSQL not supported)
- **drivers** not part of JDK!
  - CLASSPATH env variable
  - "-cp" JVM argument
- jdbc: <subprotocol>:<subname>
- throws SQLException
- JDBC driver implementation must provide impl:
  java.sql: Driver, Connection, Statement



java.sql.Connection Interface

- throws SQLException

Method	Description
`Statement createStatement()`	Creates a `Statement` object that can be used to send SQL statements to the database.
`PreparedStatement prepareStatement(String sql)`	Creates a `PreparedStatement` object that can contain SQL statements. The SQL statement can have IN parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.
`CallableStatement prepareCall(String sql)`	Creates a `CallableStatement` object for calling stored procedures in the database. The SQL statement can have IN or OUT parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.
`DatabaseMetaData getMetaData()`	Gets the `DataBaseMetaData` object. This metadata contains database schema information, table information, and so on, which is especially useful when you don't know the underlying database.
`Clob createClob()`	Returns a `Clob` object (`Clob` is the name of the interface). Character Large Object (CLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
`Blob createBlob()`	Returns a `Blob` object (`Blob` is the name of the interface). Binary Large Object (BLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
`void setSchema(String schema)`	When passed the schema name, sets this `Connection` object to the database schema to access.
`String getSchema()`	Returns the schema name of the database associated with this `Connection` object; returns `null` if no schema is associated with it.

```
connection.createStatement();
connection.getStatement(); -> CE no such method
```

java.sql.DriverManager

- helps establish the connection
- don't forget to close();

Method	Description

`static Connection getConnection(String url)`
`static Connection getConnection(String url, Properties info)`
`static Connection getConnection(String url, String user, String password) ()`
	Attempts to establish a connection given the database URL. Additionally, you can provide Information such as a username and password directly as `String` arguments or through a `Properties` file. This method throws an `SQLException` if the connection can't be established.

`static Driver getDriver(String url)`	Searches the list of registered JDBC drivers and, if found, returns the appropriate `Driver` object matching the database URL.
`static void registerDriver(Driver driver)`	Add to the list of registered `Driver` objects in the `DriverManager`.
`static void deregisterDriver(Driver driver)`	Deregisters a driver from the list of registered `Driver` objects in the `DriverManager`

```
try (Connection connection =
    DriverManager.getConnection
      (url + database, userName, password)
){ … }
catch (SQLException e) { … }

URL Samples:
  jdbc:postgresql://localhost/test
  jdbc:oracle://127.0.0.1:44000/test
  jdbc:microsoft:sqlserver://himalaya:1433
```

```
try {
 String url = "jdbc:mysql://localhost:3306/";
 Driver driver = DriverManager.getDriver(url);
 System.out.println(driver.getClass().getName());
 Connection conn=driver.connect(url,/*props=*/null);
 …
 conn.close();
} catch (SQLException e) { … }
```

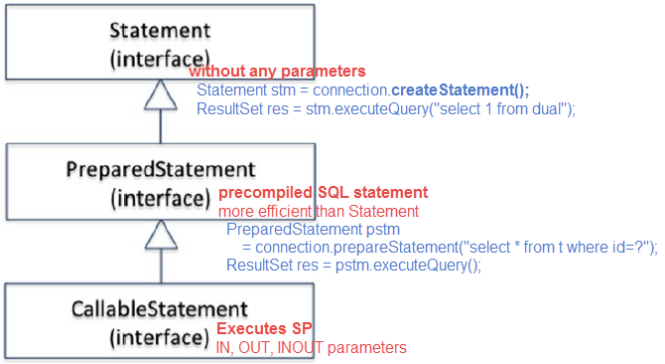
CRUD

- create, read, update, delete

Statement Interface

- communicate a SQL statement to DB
- executeQuery() -> ResultSet
- executeUpdate() -> rowcount
- execute() -> multiple ResultSets/rowcounts

PreparedStatement extends ~~implements~~ Statement interface



Method	Description
`boolean execute(String sql)`	Executes the given SQL query. This method returns `true` if the query resulted in a `ResultSet`. You can retrieve the `ResultSet` object by calling the `getResultSet()` method. This method returns `false` if the SQL query has no results or if there is an update count. You can use the `getUpdateCount()` method to get the update count. In rare situations, this method may return multiple `ResultSets`; in that case, you can call the `getMoreResults()` method.
`ResultSet executeQuery(String sql)`	Executes the query and returns the `ResultSet` object as the result. If there are no results, the method does not return `null`; rather, the returned `ResultSet` object will return false when the `next()` method is called.
`int executeUpdate(String sql)`	Executes `CREATE`, `INSERT`, `UPDATE`, or `DELETE` SQL queries. It returns the number of rows updated (or zero if there is no result, such as with the `CREATE` statement).
`Connection getConnection()`	Returns the `Connection` object with which the `Statement` object was created.
`void close()`	Closes the database and other JDBC resources associated with this `Statement` object. Calling `close()` on an already-closed `Statement` object has no effect.

ResultSet Interface

- tabular data (records, heading:columns/datatype)
- cursor pointing (one record at a time!)
- initially: cursor before 1st row - use next()

- absolute(1) = first()
- absolute(-1) = last()

Method	Description
void beforeFirst()	Sets the cursor just before the first row in the resultset.
void afterLast()	Sets the cursor just after the last row of the resultset.
boolean absolute(int rowNumber) + from begin / - from end	Sets the cursor to the requested row number (absolute position in the table—not relative to the current position).
boolean relative(int rowNumber) - /+ from current row	Sets the cursor to the requested row number relative to the current position. rowNumber can be a positive or negative value: a positive value moves forward, and a negative value moves backward relative to the current position.
boolean next()	Sets the cursor to the next row of the resultset.
boolean previous()	Sets the cursor to the previous row of the resultset.

get:

```
double getDouble(int columnNumber) -> 1..N
double getDouble(String name) case-insensitive
```

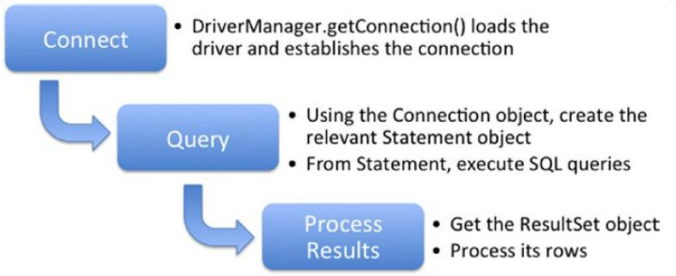
set:

```
void updateXXX(int columnNumber, V  x) -> 1..N
void updateXXX(String columnName, V x)
```

NOTE: 1..N - SQL column order (not DB table)

```
select age, age from persons
resultSet.getInteger("age") -> first "age"
```

Querying the Database




```
public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection("...");
         Statement statement   = connection.createStatement();
         ResultSet resultSet   = statement.executeQuery("SELECT * FROM contact")) {
        while (resultSet.next()) {
            System.out.println(resultSet.getInt("id") + "\t"
            + resultSet.getString("firstName") + "\t"
            + resultSet.getString("lastName") + "\t"
            + resultSet.getString("email") + "\t"
            + resultSet.getString("phoneNo"));
            }
    } catch (SQLException sqle) {
        // TODO:...
    }
}
```

```
int numOfColumns =
resultSet.getMetaData().getColumnCount();
…
for(int i = 1; i <= numOfColumns; i++) {
  System.out.print(resultSet.getObject(i) + "\t");
}
```

Updating the Database

```
make resultset updatable:
Statement stmt = connection.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE
);
DON'T FORGET: to locate to row (rs.next(), …)

insert
  resultSet.moveToInsertRow();
  resultSet.updateString("firstName", "John");
  resultSet.insertRow();

update
  …go to row…
  resultSet.updateString("phoneNo", "+9199210");
  resultSet.updateDouble("salary", 123);
  resultSet.updateRow(); → Important!

delete
  …go to row…
  resultSet.deleteRow();

cancel
  resultSet.cancelRowUpdates()
```

DDL

```
stmt.executeUpdate("
    CREATE TABLE T1(id int);
");
```

Localization

- adapt software to local language/culture/counties
- don't hardcode text (externalize)
- configurable: date/time/currency/formatting
- creating resource bundles for different locales

java.util.Locale

Method	Short Description
`static Locale[] getAvailableLocales()`	Returns a list of available locales (i.e., installed locales) supported by the JVM.
`static Locale getDefault()`	Returns the default locale of the JVM.
`static void setDefault(Locale newLocale)`	Sets the default locale of the JVM.
`String getCountry()`	Returns the country *code* for the locale object.
`String getDisplayCountry()`	Returns the country *name* for the locale object.
`String getLanguage()`	Returns the language *code* for the locale object.
`String getDisplayLanguage()`	Returns the language *name* for the locale object.
`String getVariant()`	Returns the variant *code* for the locale object.
`String getDisplayVariant()`	Returns the *name* of the variant code for the locale object.
`String toString()`	Returns a `String` composed of the codes for the locale's language, country, variant, etc.

```
language + "_" + country
  + "_" + (variant + "_#" | "#") + script
  + "-" + extensions
NOTE: variant any string, depends on needs

locale, locale.getDisplayLanguage()
  en                 English
  en_US              English (USA)
  ar_QA              Arabic (Qatar)
  no_NO_NY           Norwegian (Norway,Nynorsk)
  th_TH_TH_#u-nu-thai Thai (Thailand,TH)
  sr_RS_#Latn        Serbian (Serbia, script:Latin)
```

```
Locale.GERMANY.getDisplayCountry() → Deutschland
Locale.GERMANY.getDisplayCountry(Locale.ENGLISH)
    -> Germany (how British refer country Germany)
```

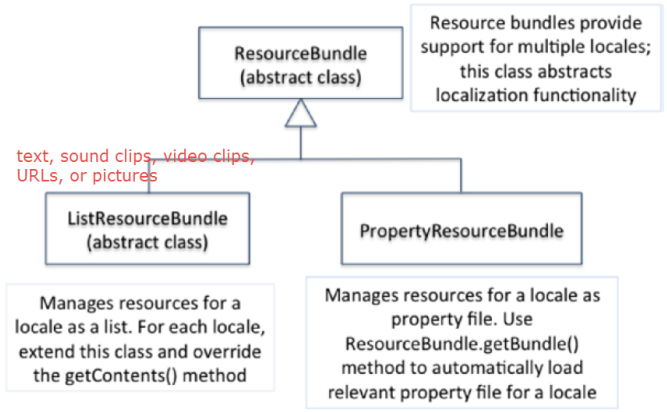
```
Locale.setDefault(new Locale("fr", "CA", ""));
Locale.setDefault(Locale.CANADA_FRENCH);
Locale def = Locale.getDefault();

def                          → fr_CA
def.getLanguage()            → fr      ISO 639
def.getDisplayLanguage()     → français

def.getCountry()             → CA      ISO 3166
def.getDisplayCountry());    → Canada

def.getVariant()             → null
def.getDisplayVariant()      → null
    → Could be: Sun or Oracle / MAC for Macintosh
```

Define Locale:

```
Locale locale1
  = new Locale("it", "", "");
  = Locale.forLanguageTag("it");
  = new Locale.Builder().setLanguageTag("it").build();
  = Locale locale4 = Locale.ITALIAN;

java -Duser.language=it -Duser.region=IT MyClass
```

java.util.ResourceBundle

- customize app to locale-specific needs
- case-sensitive key to local specific value



java.util.ResourceBundle:

Method	Short Description
`Object getObject(String key)`	Returns the value mapped to the given key. Throws a `MissingResourceException` if no object for a given key is found.
`static ResourceBundle getBundle (String baseName), static final ResourceBundle getBundle (String baseName, Locale locale)`	Returns the `ResourceBundle` for the given `baseName`, `locale`, and `control`; throws a `MissingResourceException` if no matching resource bundle is found. The `Control` parameter is meant for controlling or obtaining info about the resource bundle loading process
`final ResourceBundle getBundle (String baseName, Locale targetLocale, Control control)`	
`String getString(String key)`	Returns the value mapped to the given key; equivalent to casting the return value from `getObject()` to `String`. Throws a `MissingResourceException` if no object for a given key is found. Throws `ClassCastException` if the object returned is not a `String`.

java.util.PropertyResourceBundle

- string values only

Create files:

```
  ResourceBundle.properties
  ResourceBundle_ar.properties
  ResourceBundle_it.properties

Locale loc = Locale.getDefault();
ResourceBundle resBundle =
   ResourceBundle.getBundle("ResourceBundle", loc);
System.out.printf(resBundle.getString("Greeting"));
```

```
Locale.setDefault(Locale.ITALY); -> not recommended
java -Duser.language=it -Duser.region=IT MyClass
```

java.util.ListResourceBundle

- string, image, video, object
- **public class!**
- abstract class

```
    protected Object[][] getContents();

public class ResBundle extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }
  static final Object[][] contents = {
    { "MovieName", "Avatar" },
    { "GrossRevenue", (Long) 2782275172L },
    { "Year", (Integer)2009 }
  };
}
…
public class ResBundle_it_IT extends ListResourceBundle{
  public Object[][] getContents() {
    return contents;
  }
  static final Object[][] contents = {
    { "MovieName", "Che Bella Giornata" },
    { "GrossRevenue", (Long) 43000000L },
    { "Year", (Integer)2011 }
  };
}
…
Locale locale = Locale.getDefault();
resBundle =
 ResourceBundle.getBundle("ResBundle",locale));
…
String movieName = resBundle.getString("MovieName");
Long revenue = (Long)resBundle.getObject("GrossRevenue");
Integer year = (Integer)resBundle.getObject("Year");
…
(Integer)(resBundle.getObject("GrossRevenue")); → RTE
(Long)(resBundle.getObject("xxx")); → RTE
    → MissingResourceException
```

Loading Resource Bundles

Rules and Naming Conventions

- public class!
- packagequalifier.bundlename
    + "_" + language
    + "_" + country
    + "_" + (variant + "_#" | "#") + script
    + "-" + extensions

Sample:

localization.examples.AppBundle_en_US_Oracle_exam

Seach sequence for ResourceBundles **in classpath:**

1. exact match (full name)
2. removes details after "_" one-by-one
3. restart serch (default locale)
4. just resource bundle name (without locale)
5. **MissingResourceException**

```
Name + "_" + language + "_" + country + "_" +variant
Name + "_" + language + "_" + country
Name + "_" + language
Name + "_" + defaultLanguage + "_" + defaultCountry
Name + "_" + defaultLanguage
```

```
ResourceBundle.properties
ResourceBundle ar.properties
ResourceBundle_en.properties
ResourceBundle_it.properties
ResourceBundle_it_IT_Rome.properties

default: en_US
loadResourceBundle(
    "ResourceBundle", new Locale("fr", "CA", "")
);
 → ResourceBundle_en.properties
```

---

getBundle() gets ResourceBundle.**Control** class
can change the default resource bundle search process or read from non-standard resource bundle formats (such as XML files).

```
class MyControl extends ResourceBundle.Control {
   …
}
```

```
Locale locale = new Locale("super", "kuku");
System.out.println(locale); -> super KUKU
                NOTE: exception not thrown, but
                resource values will be null
```

TODO: Later

NOTES:
- Implicitly (неявно) / Explicitly (явно)
- automatic variable: allocated/de-allocated aumatically. Scoped { {int a=1;} int a=2; }
- **Callable.call()** allows you to declare **checked exceptions** while Runnable.run() does not. So if your task throws a checked exception, it would be more appropriate to use a Callable.

Patterns

Builder pattern:
- new Locale.Builder().setLanguageTag("it").build();
- Stream.builder().add(1).add(2).add(3).build()

Factory pattern:
- NumberFormat.getInstance(Locale.GERMANY);

```
while(aFilePath.iterator().hasNext()) {
        System.out.println("path element: " + aFilePath.
        iterator().next());
}
```

Infinite loop!

ArrayStoreException

```
class Base {}
class DeriOne extends Base {}
class DeriTwo extends Base {}
class ArrayStore {
  public static void main(String []args) {
    Base [] b = new DeriOne[3];
    b[0] = new DeriOne();
    b[2] = new DeriTwo(); -> RE: ArrayStoreException
    System.out.println(b.length);
  }
}
```