Software Engineering 301:
Software Analysis and Design

Structural modelling

Structure diagrams

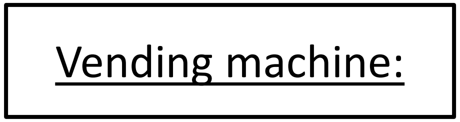
- Used to represent:
  - classes (and other kinds of types)
    - attributes
    - operations
  - relationships between types
  - objects
  - packages
- They do not represent run-time behaviour

Agenda

- Structural models
  - Syntax
    - Objects and their relationships
    - Classes
    - Class relationships
    - Interfaces
    - Packages
  - Practice
  - Purpose

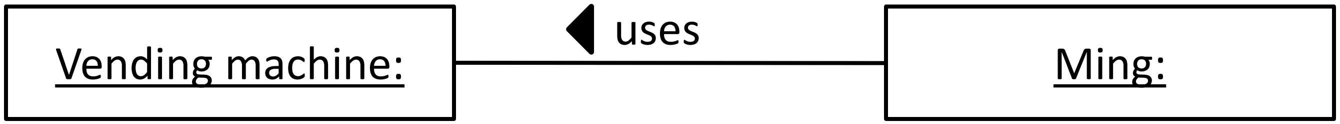
Objects

- Objects: rectangle with label underlined
  - general form [<name>] : [<classifier>]



Vending machine:

- Relationships between objects (called *links*): lines between the boxes
  - the line can have a label (and arrow if the direction is unclear)



Vending machine:

uses

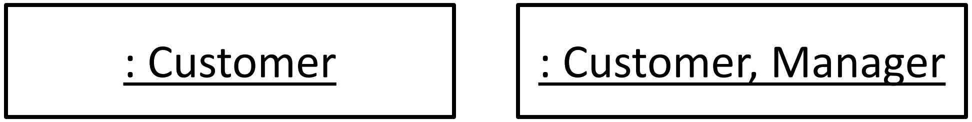
Ming:

Objects

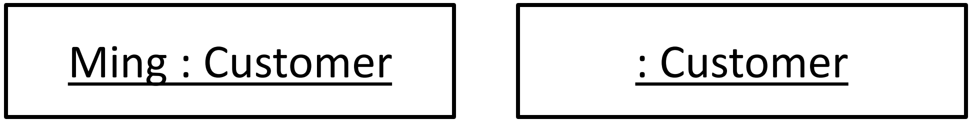
- A link always connects exactly two objects
- Links can have direction:
  - unidirectional
  - bidirectional
  - not specified
- Direction indicates which object can send messages (other object can only respond)

Objects

- Objects can have types
  - Which types we choose to denote for an object is a modelling choice!

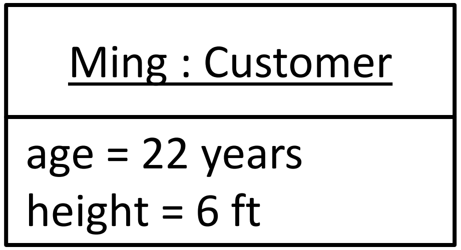


  - We can notice that a specific object is also a known kind of object



Objects

- Objects can have properties that contain values (called **slots**)



Ming : Customer

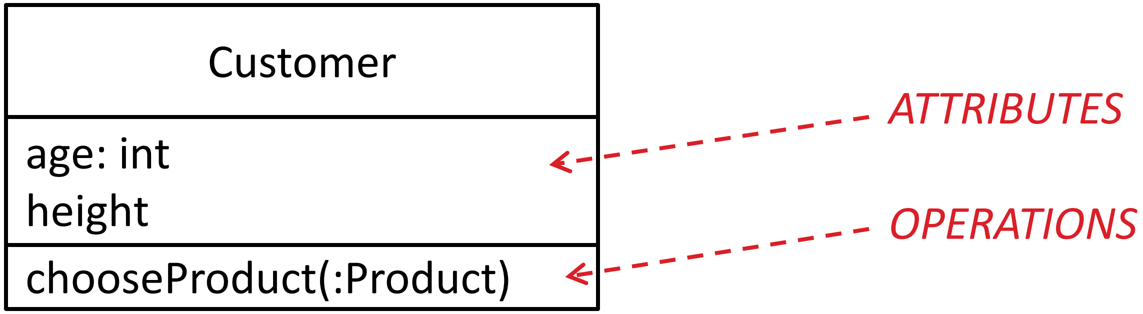
age = 22 years
height = 6 ft

Agenda

- Structural models
  - Syntax
    - ~~Objects and their relationships~~
    - Classes
    - Class relationships
    - Interfaces
    - Packages
  - Practice
  - Purpose

Classes

- Rectangles are also used for classes
- The label for a class has **no colon** (":"), and will (usually) not be underlined



Customer

age: int
height

chooseProduct(:Product)

ATTRIBUTES

OPERATIONS

- Attributes are properties of instances of the class
- Operations are actions that instances of the class can be told to perform
- Attributes and/or operations can be suppressed

Classes

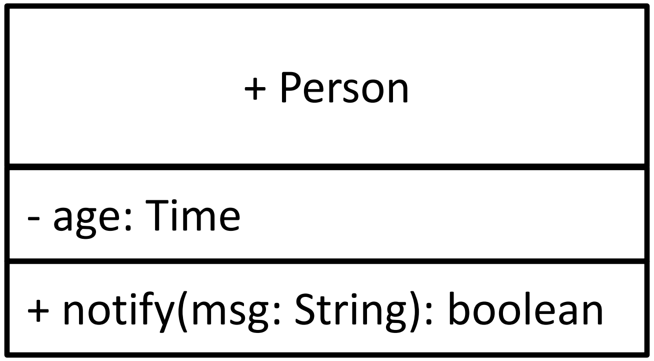
- Attributes and operations details
  - Visibility

    + public

    - private

    # protected

    ~ package protected
  - Scope
    - by default, instance scope
    - class scope (“static”) is shown by underlining the operation or attribute
  - (Result) type
    - Shown after a colon at the end of the name, parameter list
  - Parameters
    - Each parameter can have a name and/or type, e.g., foo: Bar



+ Person

- age: Time

+ notify(msg: String): boolean

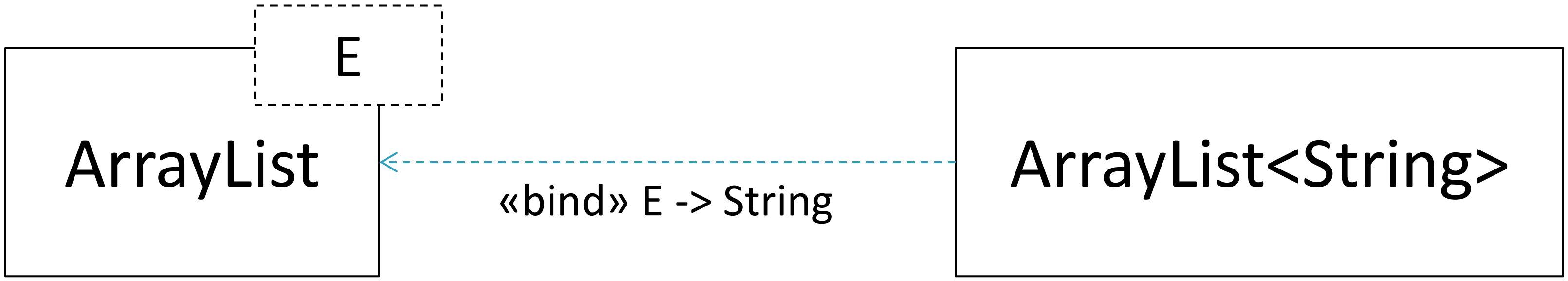
Classes

- Attributes and operations
  - Abstractness
    - No implementation is provided by the class
    - Shown by *italicizing* the operation or class
    - (“Pure virtual” functions in C++ are abstract)
  - Other modifiers
    - No built-in notation: use stereotypes, e.g., + «final» String
    - [We will see stereotypes many times, for many uses]
  - Additional details can be shown as **properties** which go at the end of the line, shown in braces as key=value pairs

    e.g., + notify() {exceptions=UnknownPersonException}
- Note that any or all of these details can be suppressed, as APPROPRIATE to the purpose of the particular model

Generic classes

- Shown with a dashed box overlapping the top right corner that contains the type parameters



- The concrete type (ArrayList<String>) can be shown like this (*a bit complicated*)

Agenda

- Structural models
  - Syntax
    - ~~Objects and their relationships~~
    - ~~Classes~~
    - Class relationships
    - Interfaces
    - Packages
  - Practice
  - Purpose

Class relationships

- There are six kinds of class relationship
  - Inheritance (generalization)
  - Dependency
  - Association
  - Aggregation
  - Composition
  - Containment

Class relationships

- Dependency
  - A depends on B if B is required for A's implementation
  - Formal parameters, local variables, static operation invocation are all kinds of dependency relationships

Class relationships

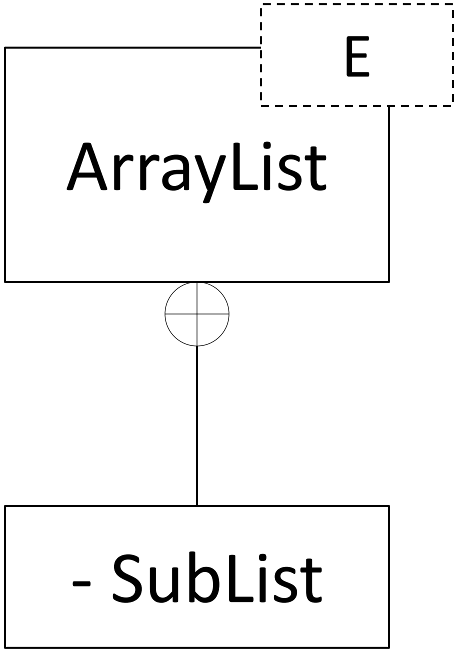
- Association
  - Individual objects of one class are connected with individual objects of another class
  - An association generally indicates that two objects are loosely but specially related to each other
  - e.g., Dept. of Computer Science is associated with SENG 301, Rob Walker is associated with SENG 301
  - Practically, an association is indication of a field in the corresponding implementation

Class relationships

- Aggregation & composition
  - Used to model situations involve a “whole” and its “parts”
  - HAS-A
  - Note: Rob Walker does not HAVE-AN instance of SENG 301, so aggregation or composition are not appropriate
  - In composition:
    - The part has no independent existence
    - When the whole is destroyed, so are its parts

Nested classes (containment)

SubList is a (private) nested class contained within ArrayList



Class relationships

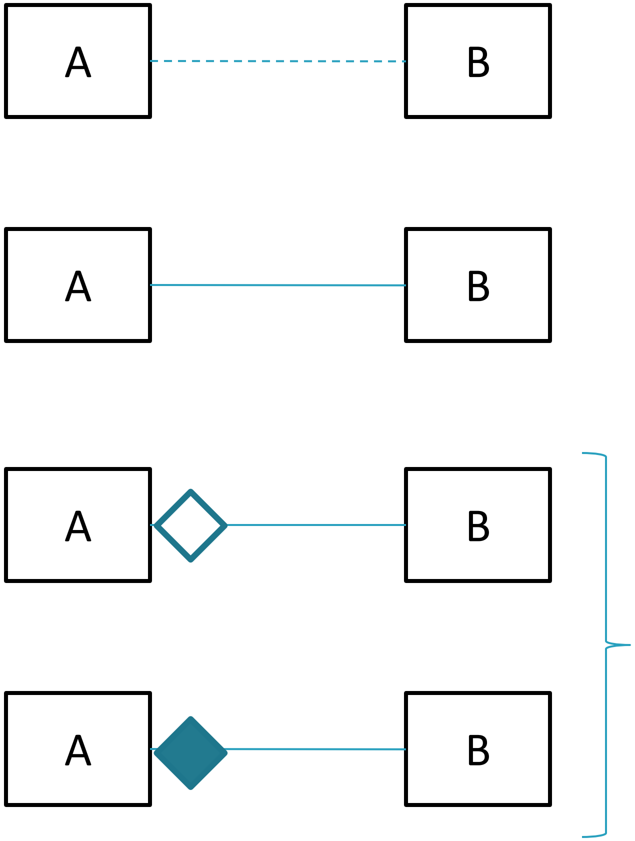
- Some of the relationships are stronger than the others
  - Composition implies aggregation implies association implies dependency (and not vice versa)
  - Generalization implies dependency (and not vice versa)
- Use the strongest applicable relationship
  - Composition and generalization are the strongest in those sequences

Class relationships

- Usually, do not use more than one relationship between two classes, when one implies the other
  - There are special cases where this is not true
- Note that aggregation and composition are relatively uncommon, so if they are all over your diagrams, you've likely made a mistake

Class relationships

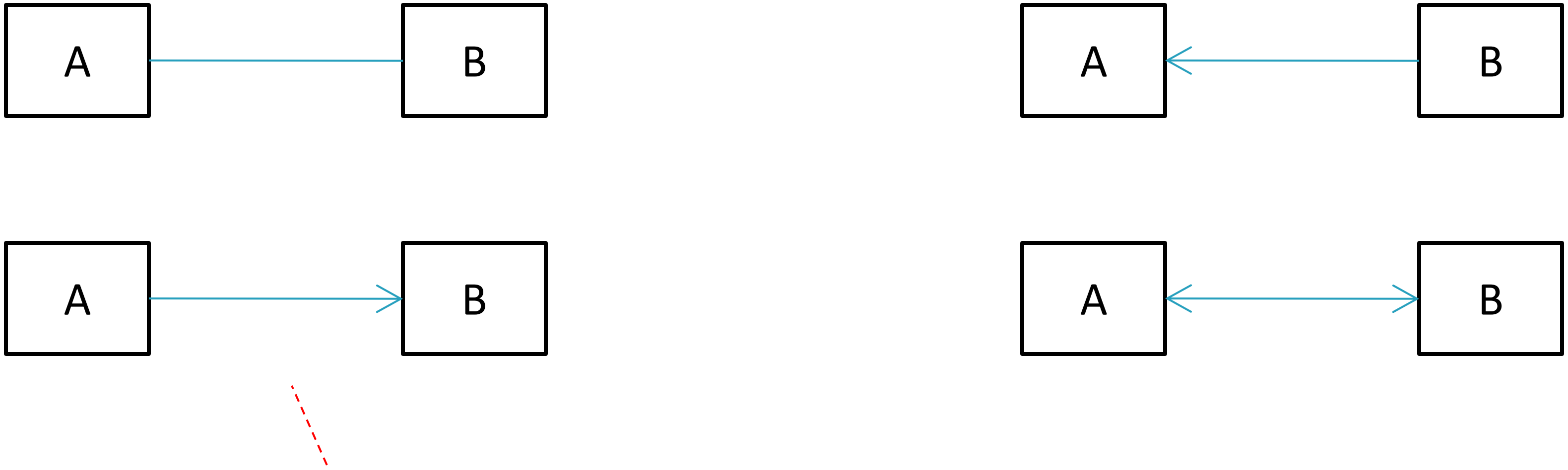
- Dependency
- Association
- Aggregation
- Composition



A is whole; B is part

Class relationships

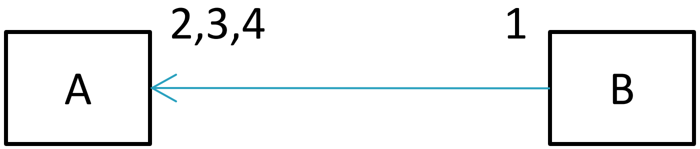
- Navigability
  - A directional relationship implies the flow of control between instances



For example, here, A can make calls to B

Class relationships

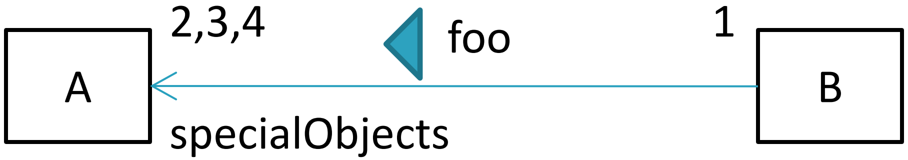
- Multiplicity
  - How many objects can be associated with each other?



  - Each B object is associated with 2, 3, or 4 A objects; each A object is associated with 1 B object
  - Other forms:
    - Ranges: m..n or 5..* (from 5 to arbitrarily many) or 0..1
    - * by itself means “from 0 to arbitrarily many”
    - You can combine these however you want, but odd combinations are uncommon

Class relationships

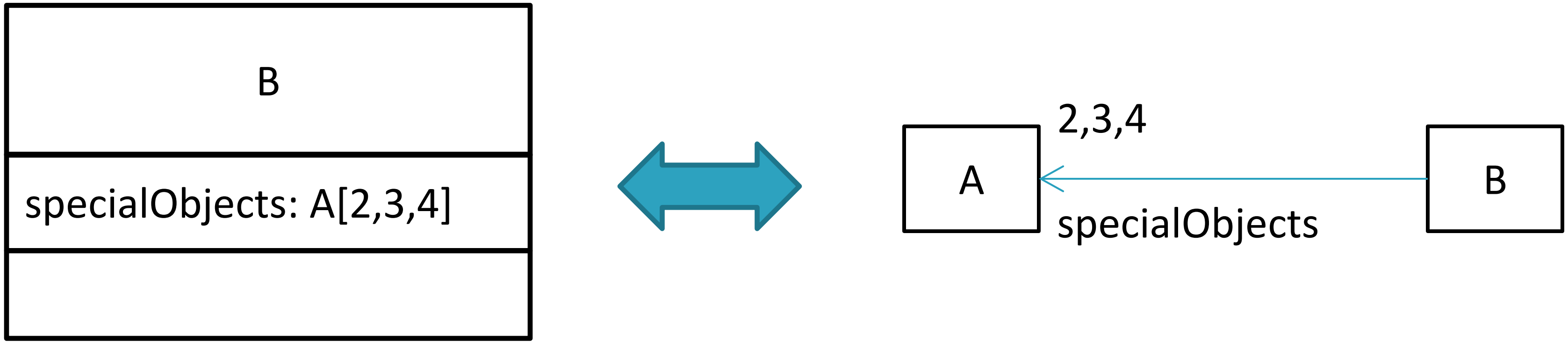
- Association labels



  - The A objects associated with each B object are labelled "specialObjects" (called the association end)
  - The association itself is labelled "foo", with an arrowhead to imply the order of reading
  - The "foo" label kind is more generic than the "specialObjects" label kind

Class relationships

- Attributes can be shown alternatively as labels on association arrows (like “specialObjects”)
- This makes most sense when you want to show the attribute’s type as an explicit class



B

specialObjects: A[2,3,4]

A

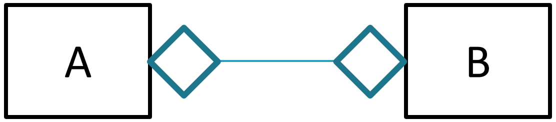
2,3,4

specialObjects

B

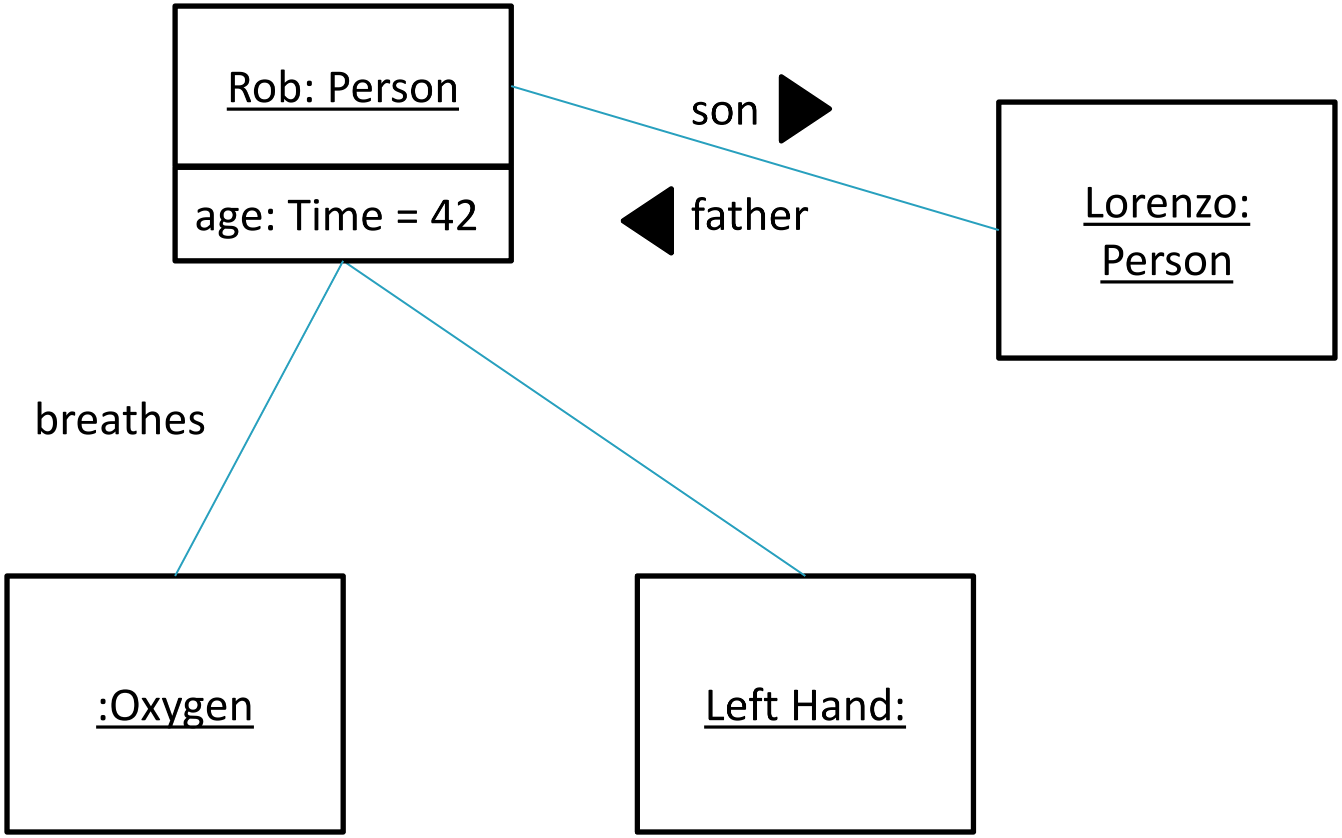
Class relationships

- Aggregation & composition are necessarily asymmetrical relationships
  - If A is the whole and B is the part, B cannot simultaneously be the part and A the whole
  - Meaning, this is **ALWAYS** wrong:

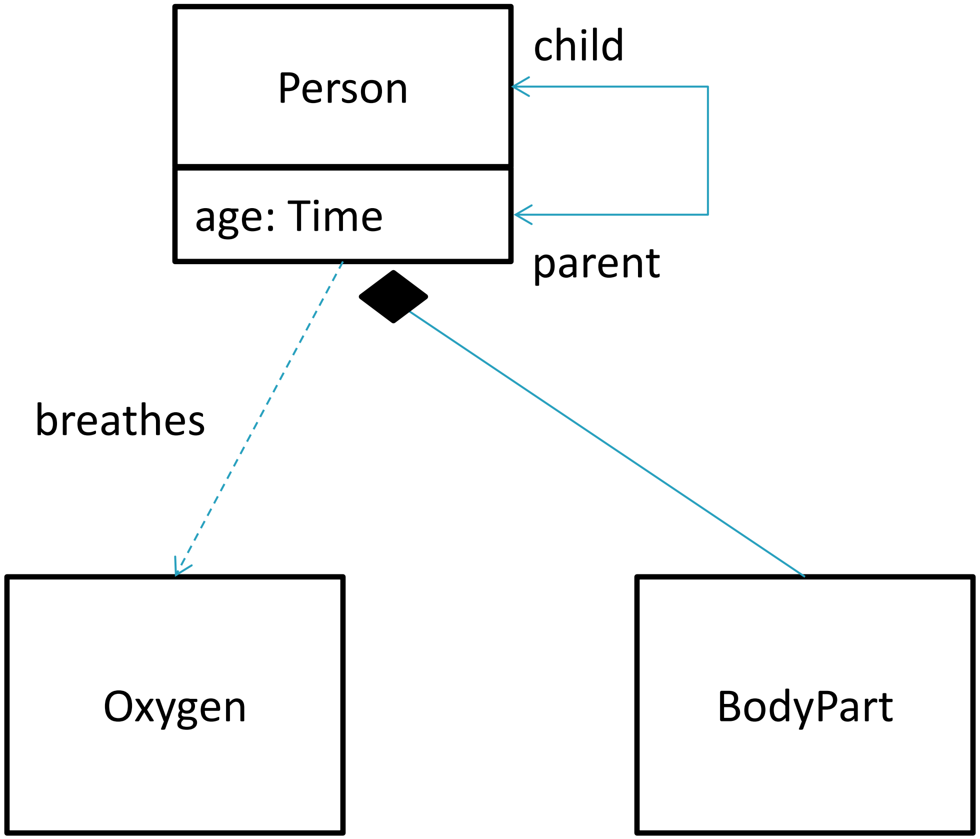


WRONG!

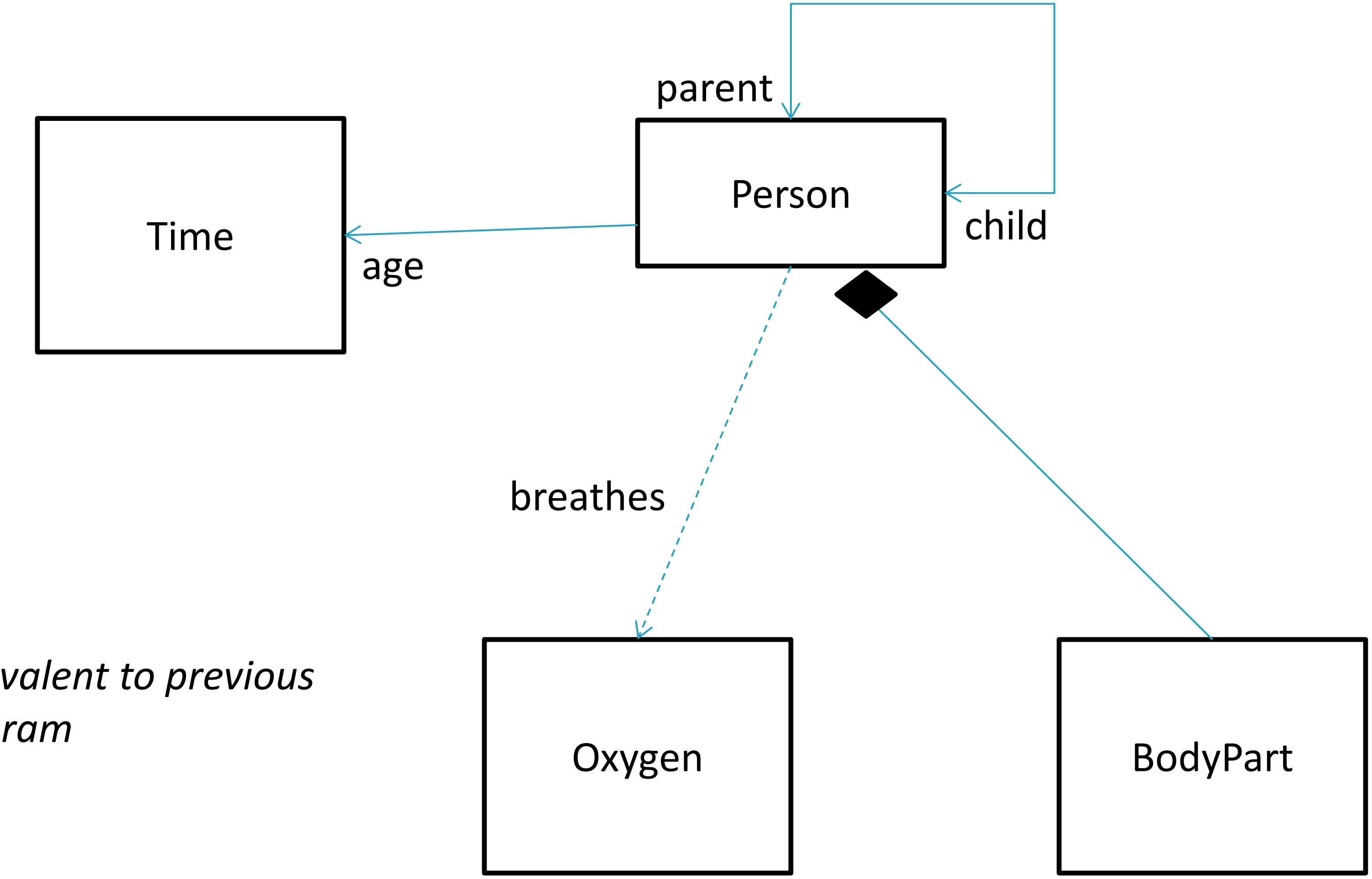
An example


Rob: Person
age: Time = 42
son ▶
◀ father
Lorenzo: Person
breathes
:Oxygen
Left Hand:

An example



An example



Equivalent to previous diagram

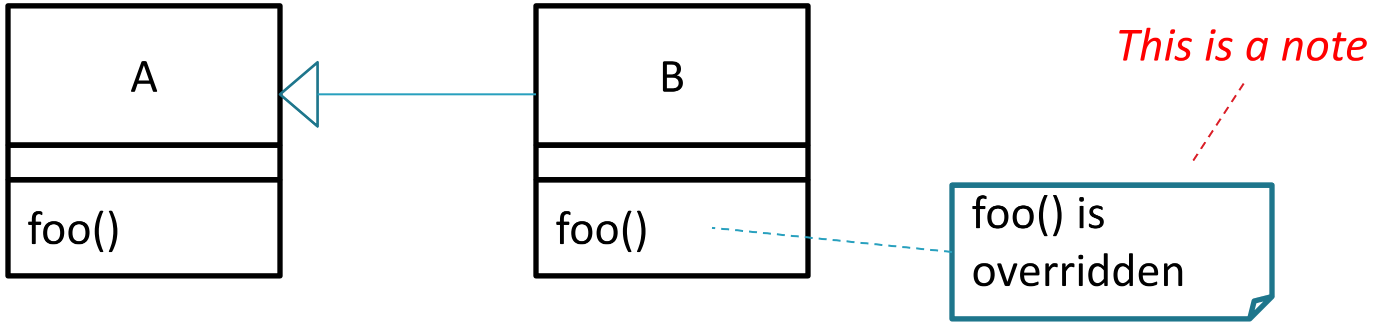
Exercise

- Consider the following cases; which relationship is appropriate for each? Can you also choose navigability?
  - Course and instructor
  - Course and student
  - University and people
  - University and government
  - Government and citizens
  - Car and wheel
  - Team, player, and coach

Class generalization

- Generalization is the inverse of specialization
- A generalizes B implies that A is the superclass of B and that B is the subclass of A
- Specialization implies inheritance: all the attributes and operations of the supertype appear within the subtype, unless marked as private (-)
- A generalizes B means that B is a special kind of A
  - Thus, generalization should be used only when this interpretation makes sense, and not for "convenience"

Class generalization



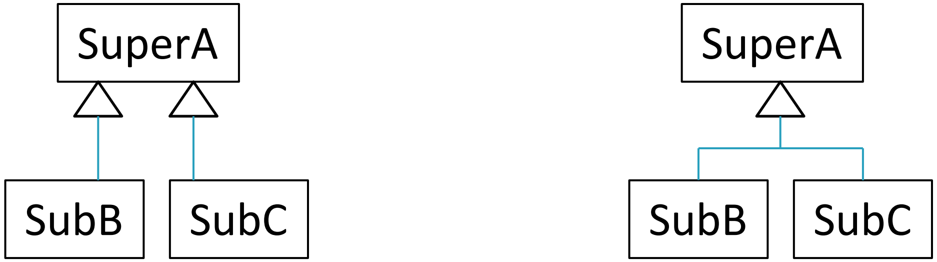
- Here, A generalizes B (so B is the subclass)
- B inherits A.foo() but it overrides it with its own implementation
- Given the following Java code, which foo() is executed?

  A a = new B(); a.foo();

  – B.foo() is executed because the object in a is an instance of B (this is **polymorphism**)

Class generalization

- A notational detail:
  - These are equivalent, in common practice



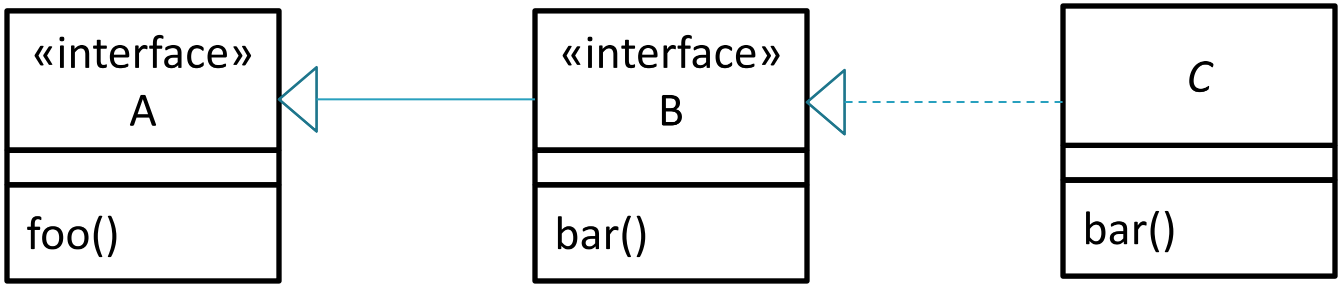
Agenda

- Structural models
    - Syntax
        - ~~Objects and their relationships~~
        - ~~Classes~~
        - ~~Class relationships~~
        - Interfaces
        - Packages
    - Practice
    - Purpose

Interface types

- It is useful to have types that provide no implementations at all, but just specify “contracts” that other types conform to
  - These are **interface** types
- Denote as with class, but place «interface» above name of type
- Operations in an interface are necessarily abstract, and so interfaces themselves are always abstract
  - Don’t bother italicizing everything though!

Interface generalization



- Interface B is a specialization of interface A
- Class C **realizes** the interface B, meaning that it conforms to the interface
  - Since it does not provide an implementation for foo(), it has to be abstract (meaning that it cannot be directly instantiated)
  - Alternatively, perhaps it does declare foo() but this is not shown
  - Alternatively, perhaps the semantics of the Java language are not appropriate here

Agenda

- Structural models
  - Syntax
    - ~~Objects and their relationships~~
    - ~~Classes~~
    - ~~Class relationships~~
    - ~~Interfaces~~
    - Packages
  - Practice
  - Purpose

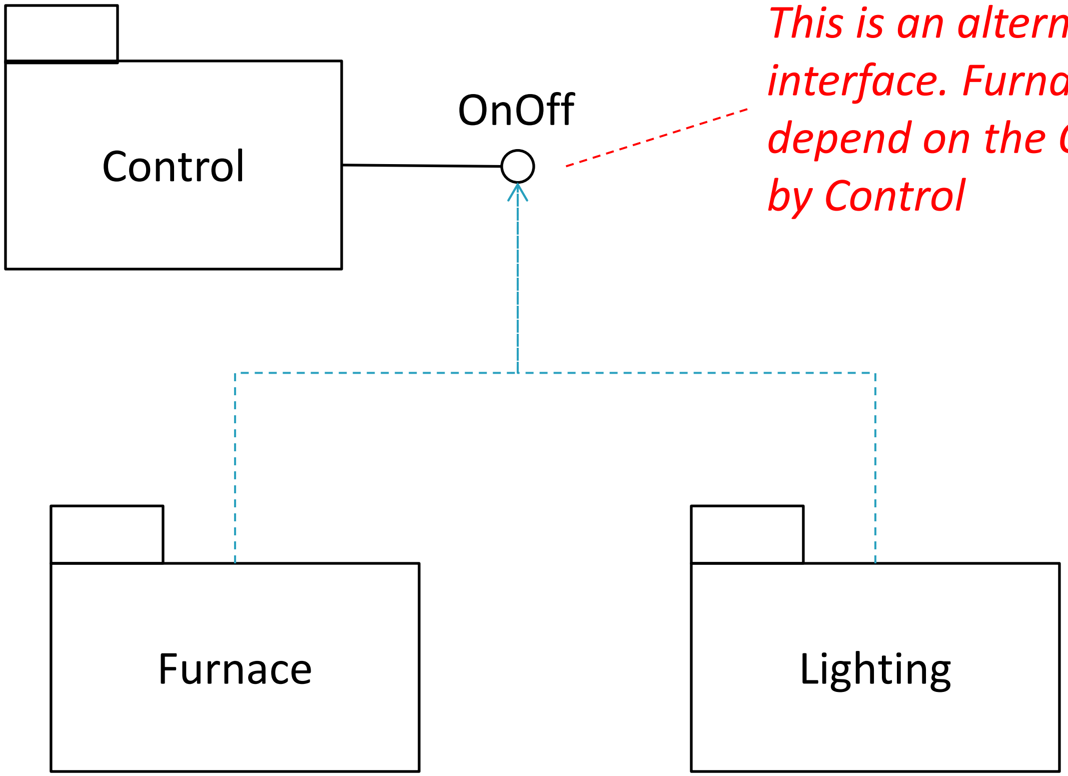
Packages

- Packages are used to show grouping of classes
  - “physical”: the class implementations are arranged in certain folders
  - “conceptual”: the purposes of the classes allows them to be organized in certain ways
- Most useful when you have lots of classes
- Packages can **<u>nest</u>** but they cannot otherwise overlap

Packages

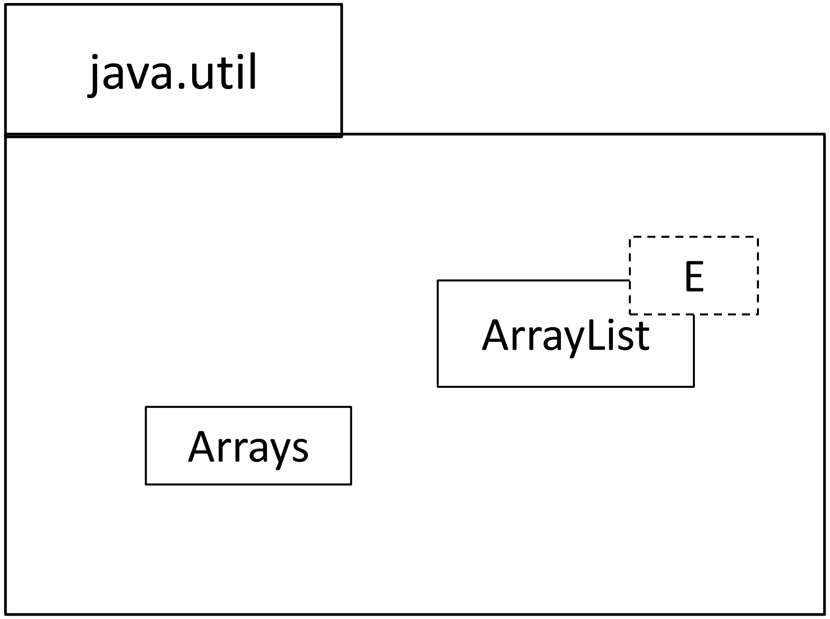
- Packages are denoted with “folder” symbols
  - When you want to show something inside the package (like classes), the package name goes in the tab
  - Otherwise, you can fill up the main rectangle with the name
- Packages can have relationships:
  - Dependency: one or more classes in package A depend on one or more classes in package B
  - Other kinds are possible, but we will ignore these

Package diagram



This is an alternative notation for interface. Furnace and Lighting both depend on the OnOff interface provided by Control

Mixing packages and classes



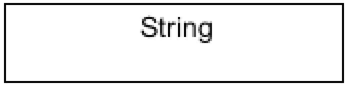
Agenda

- Structural models
  - ~~Syntax~~
    - ~~Objects and their relationships~~
    - ~~Classes~~
    - ~~Class relationships~~
    - ~~Interfaces~~
    - ~~Packages~~
  - Practice
  - Purpose

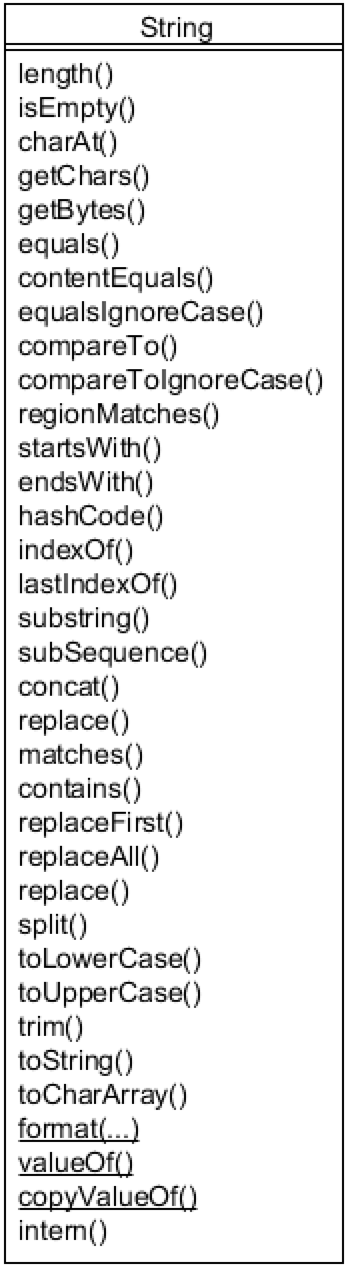
Practice

- Let’s see how some software can be modelled statically
  - java.lang.String (http://www.docjar.com/html/api/java/lang/String.java.html)
  - Java Collections Framework (JSE 5.0) (http://docs.oracle.com/javase/1.5.0/docs/api/java/util/package-summary.html)

java.lang.String

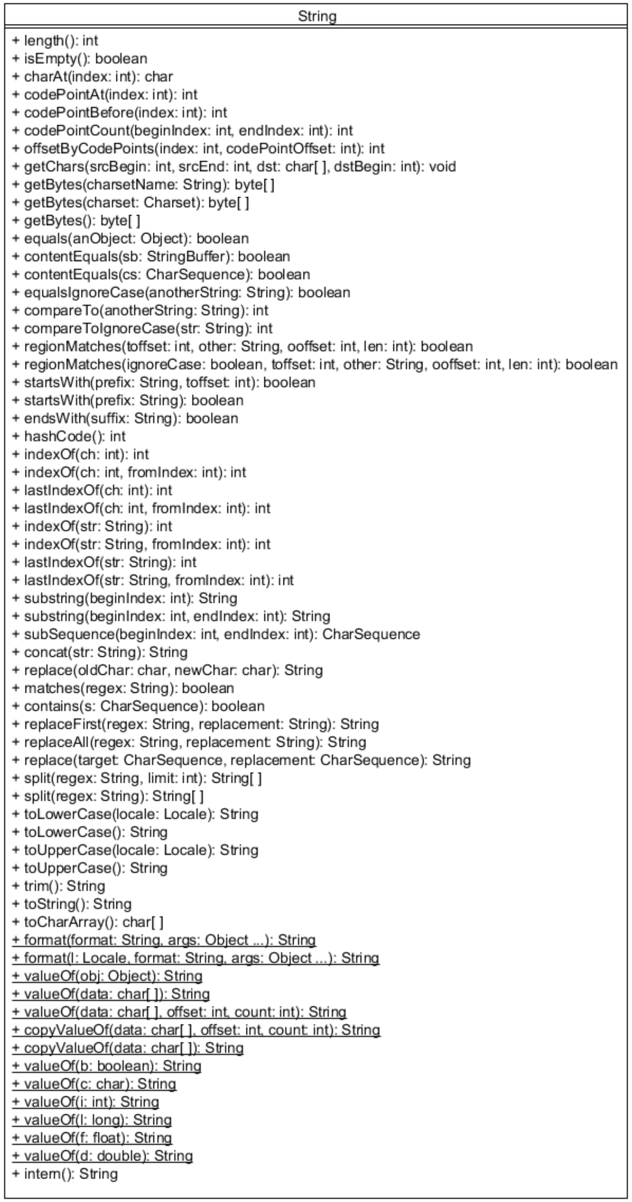

String

Which one is correct and why?


String

length()
isEmpty()
charAt()
getChars()
getBytes()
equals()
contentEquals()
equalsIgnoreCase()
compareTo()
compareToIgnoreCase()
regionMatches()
startsWith()
endsWith()
hashCode()
indexOf()
lastIndexOf()
substring()
subSequence()
concat()
replace()
matches()
contains()
replaceFirst()
replaceAll()
replace()
split()
toLowerCase()
toUpperCase()
trim()
toString()
toCharArray()
format(...)
valueOf()
copyValueOf()
intern()

java.lang.String

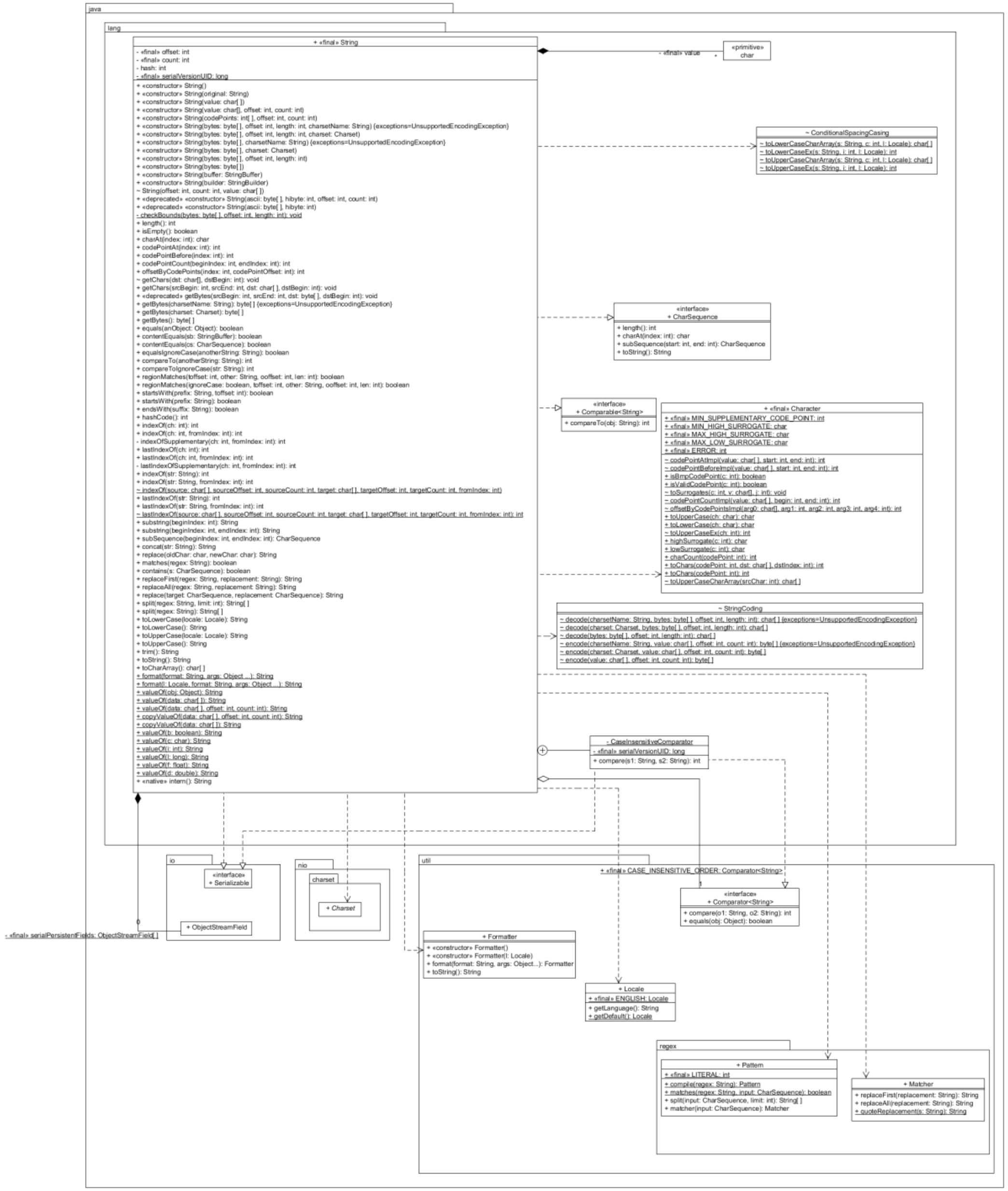


```
String
+ length(): int
+ isEmpty(): boolean
+ charAt(index: int): char
+ codePointAt(index: int): int
+ codePointBefore(index: int): int
+ codePointCount(beginIndex: int, endIndex: int): int
+ offsetByCodePoints(index: int, codePointOffset: int): int
+ getChars(srcBegin: int, srcEnd: int, dst: char[ ], dstBegin: int): void
+ getBytes(charsetName: String): byte[ ]
+ getBytes(charset: Charset): byte[ ]
+ getBytes(): byte[ ]
+ equals(anObject: Object): boolean
+ contentEquals(sb: StringBuffer): boolean
+ contentEquals(cs: CharSequence): boolean
+ equalsIgnoreCase(anotherString: String): boolean
+ compareTo(anotherString: String): int
+ compareToIgnoreCase(str: String): int
+ regionMatches(toffset: int, other: String, ooffset: int, len: int): boolean
+ regionMatches(ignoreCase: boolean, toffset: int, other: String, ooffset: int, len: int): boolean
+ startsWith(prefix: String, toffset: int): boolean
+ startsWith(prefix: String): boolean
+ endsWith(suffix: String): boolean
+ hashCode(): int
+ indexOf(ch: int): int
+ indexOf(ch: int, fromIndex: int): int
+ lastIndexOf(ch: int): int
+ lastIndexOf(ch: int, fromIndex: int): int
+ indexOf(str: String): int
+ indexOf(str: String, fromIndex: int): int
+ lastIndexOf(str: String): int
+ lastIndexOf(str: String, fromIndex: int): int
+ substring(beginIndex: int): String
+ substring(beginIndex: int, endIndex: int): String
+ subSequence(beginIndex: int, endIndex: int): CharSequence
+ concat(str: String): String
+ replace(oldChar: char, newChar: char): String
+ matches(regex: String): boolean
+ contains(s: CharSequence): boolean
+ replaceFirst(regex: String, replacement: String): String
+ replaceAll(regex: String, replacement: String): String
+ replace(target: CharSequence, replacement: CharSequence): String
+ split(regex: String, limit: int): String[ ]
+ split(regex: String): String[ ]
+ toLowerCase(locale: Locale): String
+ toLowerCase(): String
+ toUpperCase(locale: Locale): String
+ toUpperCase(): String
+ trim(): String
+ toString(): String
+ toCharArray(): char[ ]
+ format(format: String, args: Object ...): String
+ format(l: Locale, format: String, args: Object ...): String
+ valueOf(obj: Object): String
+ valueOf(data: char[ ]): String
+ valueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ]): String
+ valueOf(b: boolean): String
+ valueOf(c: char): String
+ valueOf(i: int): String
+ valueOf(l: long): String
+ valueOf(f: float): String
+ valueOf(d: double): String
+ intern(): String
```

Showing formal parameter names and types, return types

java.lang.String

Relationships with other types plus packages



java

lang

+ «final» String
- «final» offset: int
- «final» count: int
- hash: int
- «final» serialVersionUID: long
+ «constructor» String()
+ «constructor» String(original: String)
+ «constructor» String(value: char[ ])
+ «constructor» String(value: char[], offset: int, count: int)
+ «constructor» String(codePoints: int[ ], offset: int, count: int)
+ «constructor» String(bytes: byte[ ], offset: int, length: int, charsetName: String) {exceptions=UnsupportedEncodingException}
+ «constructor» String(bytes: byte[ ], offset: int, length: int, charset: Charset)
+ «constructor» String(bytes: byte[ ], charsetName: String) {exceptions=UnsupportedEncodingException}
+ «constructor» String(bytes: byte[ ], charset: Charset)
+ «constructor» String(bytes: byte[ ], offset: int, length: int)
+ «constructor» String(bytes: byte[ ])
+ «constructor» String(buffer: StringBuffer)
+ «constructor» String(builder: StringBuilder)
~ String(offset: int, count: int, value: char[ ])
+ «deprecated» «constructor» String(ascii: byte[ ], hibyte: int, offset: int, count: int)
+ «deprecated» «constructor» String(ascii: byte[ ], hibyte: int)
- checkBounds(bytes: byte[ ], offset: int, length: int): void
+ length(): int
+ isEmpty(): boolean
+ charAt(index: int): char
+ codePointAt(index: int): int
+ codePointBefore(index: int): int
+ codePointCount(beginIndex: int, endIndex: int): int
+ offsetByCodePoints(index: int, codePointOffset: int): int
~ getChars(dst: char[], dstBegin: int): void
+ getChars(srcBegin: int, srcEnd: int, dst: char[ ], dstBegin: int): void
+ «deprecated» getBytes(srcBegin: int, srcEnd: int, dst: byte[ ], dstBegin: int): void
+ getBytes(charsetName: String): byte[ ] {exceptions=UnsupportedEncodingException}
+ getBytes(charset: Charset): byte[ ]
+ getBytes(): byte[ ]
+ equals(anObject: Object): boolean
+ contentEquals(sb: StringBuffer): boolean
+ contentEquals(cs: CharSequence): boolean
+ equalsIgnoreCase(anotherString: String): boolean
+ compareTo(anotherString: String): int
+ compareToIgnoreCase(str: String): int
+ regionMatches(toffset: int, other: String, ooffset: int, len: int): boolean
+ regionMatches(ignoreCase: boolean, toffset: int, other: String, ooffset: int, len: int): boolean
+ startsWith(prefix: String, toffset: int): boolean
+ startsWith(prefix: String): boolean
+ endsWith(suffix: String): boolean
+ hashCode(): int
+ indexOf(ch: int): int
+ indexOf(ch: int, fromIndex: int): int
- indexOfSupplementary(ch: int, fromIndex: int): int
+ lastIndexOf(ch: int): int
+ lastIndexOf(ch: int, fromIndex: int): int
- lastIndexOfSupplementary(ch: int, fromIndex: int): int
+ indexOf(str: String): int
+ indexOf(str: String, fromIndex: int): int
~ indexOf(source: char[ ], sourceOffset: int, sourceCount: int, target: char[ ], targetOffset: int, targetCount: int, fromIndex: int)
+ lastIndexOf(str: String): int
+ lastIndexOf(str: String, fromIndex: int): int
~ lastIndexOf(source: char[ ], sourceOffset: int, sourceCount: int, target: char[ ], targetOffset: int, targetCount: int, fromIndex: int): int
+ substring(beginIndex: int): String
+ substring(beginIndex: int, endIndex: int): String
+ subSequence(beginIndex: int, endIndex: int): CharSequence
+ concat(str: String): String
+ replace(oldChar: char, newChar: char): String
+ matches(regex: String): boolean
+ contains(s: CharSequence): boolean
+ replaceFirst(regex: String, replacement: String): String
+ replaceAll(regex: String, replacement: String): String
+ replace(target: CharSequence, replacement: CharSequence): String
+ split(regex: String, limit: int): String[ ]
+ split(regex: String): String[ ]
+ toLowerCase(locale: Locale): String
+ toLowerCase(): String
+ toUpperCase(locale: Locale): String
+ toUpperCase(): String
+ trim(): String
+ toString(): String
+ toCharArray(): char[ ]
+ format(format: String, args: Object ...): String
+ format(l: Locale, format: String, args: Object ...): String
+ valueOf(obj: Object): String
+ valueOf(data: char[ ]): String
+ valueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ]): String
+ valueOf(b: boolean): String
+ valueOf(c: char): String
+ valueOf(i: int): String
+ valueOf(l: long): String
+ valueOf(f: float): String
+ valueOf(d: double): String
+ «native» intern(): String

- «final» value
*
«primitive»
char

~ ConditionalSpacingCasing
~ toLowerCaseCharArray(s: String, c: int, l: Locale): char[ ]
~ toLowerCaseEx(s: String, i: int, l: Locale): int
~ toUpperCaseCharArray(s: String, c: int, l: Locale): char[ ]
~ toUpperCaseEx(s: String, i: int, l: Locale): int

«interface»
+ CharSequence
+ length(): int
+ charAt(index: int): char
+ subSequence(start: int, end: int): CharSequence
+ toString(): String

«interface»
+ Comparable<String>
+ compareTo(obj: String): int

+ «final» Character
+ «final» MIN_SUPPLEMENTARY_CODE_POINT: int
+ «final» MIN_HIGH_SURROGATE: char
+ «final» MAX_HIGH_SURROGATE: char
+ «final» MAX_LOW_SURROGATE: char
+ «final» ERROR: int
~ codePointAtImpl(value: char[ ], start: int, end: int): int
~ codePointBeforeImpl(value: char[ ], start: int, end: int): int
+ isBmpCodePoint(c: int): boolean
+ isValidCodePoint(c: int): boolean
~ toSurrogates(c: int, v: char[], j: int): void
~ codePointCountImpl(value: char[ ], begin: int, end: int): int
~ offsetByCodePointsImpl(arg0: char[], arg1: int, arg2: int, arg3: int, arg4: int): int
+ toUpperCase(ch: char): char
+ toLowerCase(ch: char): char
~ toUpperCaseEx(ch: int): int
+ highSurrogate(c: int): char
+ lowSurrogate(c: int): char
+ charCount(codePoint: int): int
+ toChars(codePoint: int, dst: char[ ], dstIndex: int): int
+ toChars(codePoint: int): int
~ toUpperCaseCharArray(srcChar: int): char[ ]

~ StringCoding
~ decode(charsetName: String, bytes: byte[ ], offset: int, length: int): char[ ] {exceptions=UnsupportedEncodingException}
~ decode(charset: Charset, bytes: byte[ ], offset: int, length: int): char[ ]
~ decode(bytes: byte[ ], offset: int, length: int): char[ ]
~ encode(charsetName: String, value: char[ ], offset: int, count: int): byte[ ] {exceptions=UnsupportedEncodingException}
~ encode(charset: Charset, value: char[ ], offset: int, count: int): byte[ ]
~ encode(value: char[ ], offset: int, count: int): byte[ ]

- CaseInsensitiveComparator
- «final» serialVersionUID: long
+ compare(s1: String, s2: String): int

io

«interface»
+ Serializable

+ ObjectStreamField

0

- «final» serialPersistentFields: ObjectStreamField[ ]

nio

charset

+ *Charset*

util

+ «final» CASE_INSENSITIVE_ORDER: Comparator<String>

1

«interface»
+ Comparator<String>
+ compare(o1: String, o2: String): int
+ equals(obj: Object): boolean

+ Formatter
+ «constructor» Formatter()
+ «constructor» Formatter(l: Locale)
+ format(format: String, args: Object...): Formatter
+ toString(): String

+ Locale
+ «final» ENGLISH: Locale
+ getLanguage(): String
+ getDefault(): Locale

regex

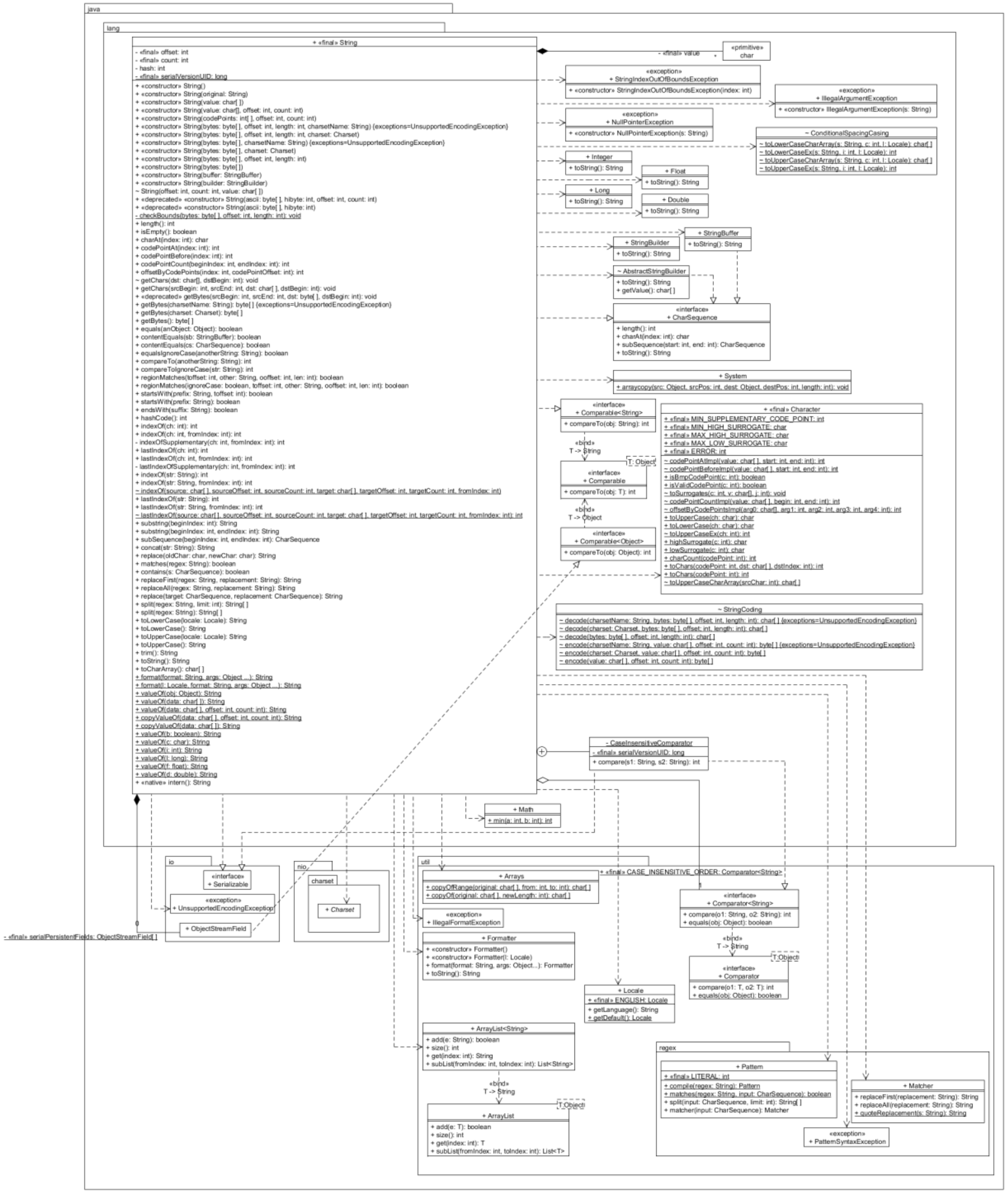
+ Pattern
+ «final» LITERAL: int
+ compile(regex: String): Pattern
+ matches(regex: String, input: CharSequence): boolean
+ split(input: CharSequence, limit: int): String[ ]
+ matcher(input: CharSequence): Matcher

+ Matcher
+ replaceFirst(replacement: String): String
+ replaceAll(replacement: String): String
+ quoteReplacement(s: String): String

java.lang.String

Private and protected types.

A lot of detail



java

lang

+ «final» String
- «final» offset: int
- «final» count: int
- hash: int
- «final» serialVersionUID: long
+ «constructor» String()
+ «constructor» String(original: String)
+ «constructor» String(value: char[ ])
+ «constructor» String(value: char[], offset: int, count: int)
+ «constructor» String(codePoints: int[ ], offset: int, count: int)
+ «constructor» String(bytes: byte[ ], offset: int, length: int, charsetName: String) {exceptions=UnsupportedEncodingException}
+ «constructor» String(bytes: byte[ ], offset: int, length: int, charset: Charset)
+ «constructor» String(bytes: byte[ ], charsetName: String) {exceptions=UnsupportedEncodingException}
+ «constructor» String(bytes: byte[ ], charset: Charset)
+ «constructor» String(bytes: byte[ ], offset: int, length: int)
+ «constructor» String(bytes: byte[ ])
+ «constructor» String(buffer: StringBuffer)
+ «constructor» String(builder: StringBuilder)
~ String(offset: int, count: int, value: char[ ])
+ «deprecated» «constructor» String(ascii: byte[ ], hibyte: int, offset: int, count: int)
+ «deprecated» «constructor» String(ascii: byte[ ], hibyte: int)
- checkBounds(bytes: byte[ ], offset: int, length: int): void
+ length(): int
+ isEmpty(): boolean
+ charAt(index: int): char
+ codePointAt(index: int): int
+ codePointBefore(index: int): int
+ codePointCount(beginIndex: int, endIndex: int): int
+ offsetByCodePoints(index: int, codePointOffset: int): int
~ getChars(dst: char[], dstBegin: int): void
+ getChars(srcBegin: int, srcEnd: int, dst: char[ ], dstBegin: int): void
+ «deprecated» getBytes(srcBegin: int, srcEnd: int, dst: byte[ ], dstBegin: int): void
+ getBytes(charsetName: String): byte[ ] {exceptions=UnsupportedEncodingException}
+ getBytes(charset: Charset): byte[ ]
+ getBytes(): byte[ ]
+ equals(anObject: Object): boolean
+ contentEquals(sb: StringBuffer): boolean
+ contentEquals(cs: CharSequence): boolean
+ equalsIgnoreCase(anotherString: String): boolean
+ compareTo(anotherString: String): int
+ compareToIgnoreCase(str: String): int
+ regionMatches(toffset: int, other: String, ooffset: int, len: int): boolean
+ regionMatches(ignoreCase: boolean, toffset: int, other: String, ooffset: int, len: int): boolean
+ startsWith(prefix: String, toffset: int): boolean
+ startsWith(prefix: String): boolean
+ endsWith(suffix: String): boolean
+ hashCode(): int
+ indexOf(ch: int): int
+ indexOf(ch: int, fromIndex: int): int
- indexOfSupplementary(ch: int, fromIndex: int): int
+ lastIndexOf(ch: int): int
+ lastIndexOf(ch: int, fromIndex: int): int
- lastIndexOfSupplementary(ch: int, fromIndex: int): int
+ indexOf(str: String): int
+ indexOf(str: String, fromIndex: int): int
~ indexOf(source: char[ ], sourceOffset: int, sourceCount: int, target: char[ ], targetOffset: int, targetCount: int, fromIndex: int)
+ lastIndexOf(str: String): int
+ lastIndexOf(str: String, fromIndex: int): int
~ lastIndexOf(source: char[ ], sourceOffset: int, sourceCount: int, target: char[ ], targetOffset: int, targetCount: int, fromIndex: int): int
+ substring(beginIndex: int): String
+ substring(beginIndex: int, endIndex: int): String
+ subSequence(beginIndex: int, endIndex: int): CharSequence
+ concat(str: String): String
+ replace(oldChar: char, newChar: char): String
+ matches(regex: String): boolean
+ contains(s: CharSequence): boolean
+ replaceFirst(regex: String, replacement: String): String
+ replaceAll(regex: String, replacement: String): String
+ replace(target: CharSequence, replacement: CharSequence): String
+ split(regex: String, limit: int): String[ ]
+ split(regex: String): String[ ]
+ toLowerCase(locale: Locale): String
+ toLowerCase(): String
+ toUpperCase(locale: Locale): String
+ toUpperCase(): String
+ trim(): String
+ toString(): String
+ toCharArray(): char[ ]
+ format(format: String, args: Object ...): String
+ format(l: Locale, format: String, args: Object ...): String
+ valueOf(obj: Object): String
+ valueOf(data: char[ ]): String
+ valueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ], offset: int, count: int): String
+ copyValueOf(data: char[ ]): String
+ valueOf(b: boolean): String
+ valueOf(c: char): String
+ valueOf(i: int): String
+ valueOf(l: long): String
+ valueOf(f: float): String
+ valueOf(d: double): String
+ «native» intern(): String

- «final» value

«primitive»
char

«exception»
+ StringIndexOutOfBoundsException
+ «constructor» StringIndexOutOfBoundsException(index: int)

«exception»
+ IllegalArgumentException
+ «constructor» IllegalArgumentException(s: String)

«exception»
+ NullPointerException
+ «constructor» NullPointerException(s: String)

~ ConditionalSpacingCasing
~ toLowerCaseCharArray(s: String, c: int, l: Locale): char[ ]
~ toLowerCaseEx(s: String, i: int, l: Locale): int
~ toUpperCaseCharArray(s: String, c: int, l: Locale): char[ ]
~ toUpperCaseEx(s: String, i: int, l: Locale): int

+ Integer
+ toString(): String

+ Float
+ toString(): String

+ Long
+ toString(): String

+ Double
+ toString(): String

+ StringBuffer
+ toString(): String

+ StringBuilder
+ toString(): String

~ AbstractStringBuilder
+ toString(): String
+ getValue(): char[ ]

«interface»
+ CharSequence
+ length(): int
+ charAt(index: int): char
+ subSequence(start: int, end: int): CharSequence
+ toString(): String

+ System
+ arraycopy(src: Object, srcPos: int, dest: Object, destPos: int, length: int): void

«interface»
+ Comparable<String>
+ compareTo(obj: String): int

«bind»
T -> String

T: Object

«interface»
+ Comparable
+ compareTo(obj: T): int

«bind»
T -> Object

«interface»
+ Comparable<Object>
+ compareTo(obj: Object): int

+ «final» Character
+ «final» MIN_SUPPLEMENTARY_CODE_POINT: int
+ «final» MIN_HIGH_SURROGATE: char
+ «final» MAX_HIGH_SURROGATE: char
+ «final» MAX_LOW_SURROGATE: char
+ «final» ERROR: int
~ codePointAtImpl(value: char[ ], start: int, end: int): int
~ codePointBeforeImpl(value: char[ ], start: int, end: int): int
+ isBmpCodePoint(c: int): boolean
+ isValidCodePoint(c: int): boolean
~ toSurrogates(c: int, v: char[], j: int): void
~ codePointCountImpl(value: char[ ], begin: int, end: int): int
~ offsetByCodePointsImpl(arg0: char[], arg1: int, arg2: int, arg3: int, arg4: int): int
+ toUpperCase(ch: char): char
+ toLowerCase(ch: char): char
~ toUpperCaseEx(ch: int): int
+ highSurrogate(c: int): char
+ lowSurrogate(c: int): char
+ charCount(codePoint: int): int
+ toChars(codePoint: int, dst: char[ ], dstIndex: int): int
+ toChars(codePoint: int): int
~ toUpperCaseCharArray(srcChar: int): char[ ]

~ StringCoding
~ decode(charsetName: String, bytes: byte[ ], offset: int, length: int): char[ ] {exceptions=UnsupportedEncodingException}
~ decode(charset: Charset, bytes: byte[ ], offset: int, length: int): char[ ]
~ decode(bytes: byte[ ], offset: int, length: int): char[ ]
~ encode(charsetName: String, value: char[ ], offset: int, count: int): byte[ ] {exceptions=UnsupportedEncodingException}
~ encode(charset: Charset, value: char[ ], offset: int, count: int): byte[ ]
~ encode(value: char[ ], offset: int, count: int): byte[ ]

- CaseInsensitiveComparator
- «final» serialVersionUID: long
+ compare(s1: String, s2: String): int

+ Math
+ min(a: int, b: int): int

io

«interface»
+ Serializable

«exception»
+ UnsupportedEncodingExceptio

+ ObjectStreamField

- «final» serialPersistentFields: ObjectStreamField[ ]

0

nio

charset

+ *Charset*

util

+ «final» CASE_INSENSITIVE_ORDER: Comparator<String>

1

+ Arrays
+ copyOfRange(original: char[ ], from: int, to: int): char[ ]
+ copyOf(original: char[ ], newLength: int): char[ ]

«exception»
+ IllegalFormatException

+ Formatter
+ «constructor» Formatter()
+ «constructor» Formatter(l: Locale)
+ format(format: String, args: Object...): Formatter
+ toString(): String

+ Locale
+ «final» ENGLISH: Locale
+ getLanguage(): String
+ getDefault(): Locale

«interface»
+ Comparator<String>
+ compare(o1: String, o2: String): int
+ equals(obj: Object): boolean

«bind»
T -> String

T:Object

«interface»
+ Comparator
+ compare(o1: T, o2: T): int
+ equals(obj: Object): boolean

+ ArrayList<String>
+ add(e: String): boolean
+ size(): int
+ get(index: int): String
+ subList(fromIndex: int, toIndex: int): List<String>

«bind»
T -> String

T:Object

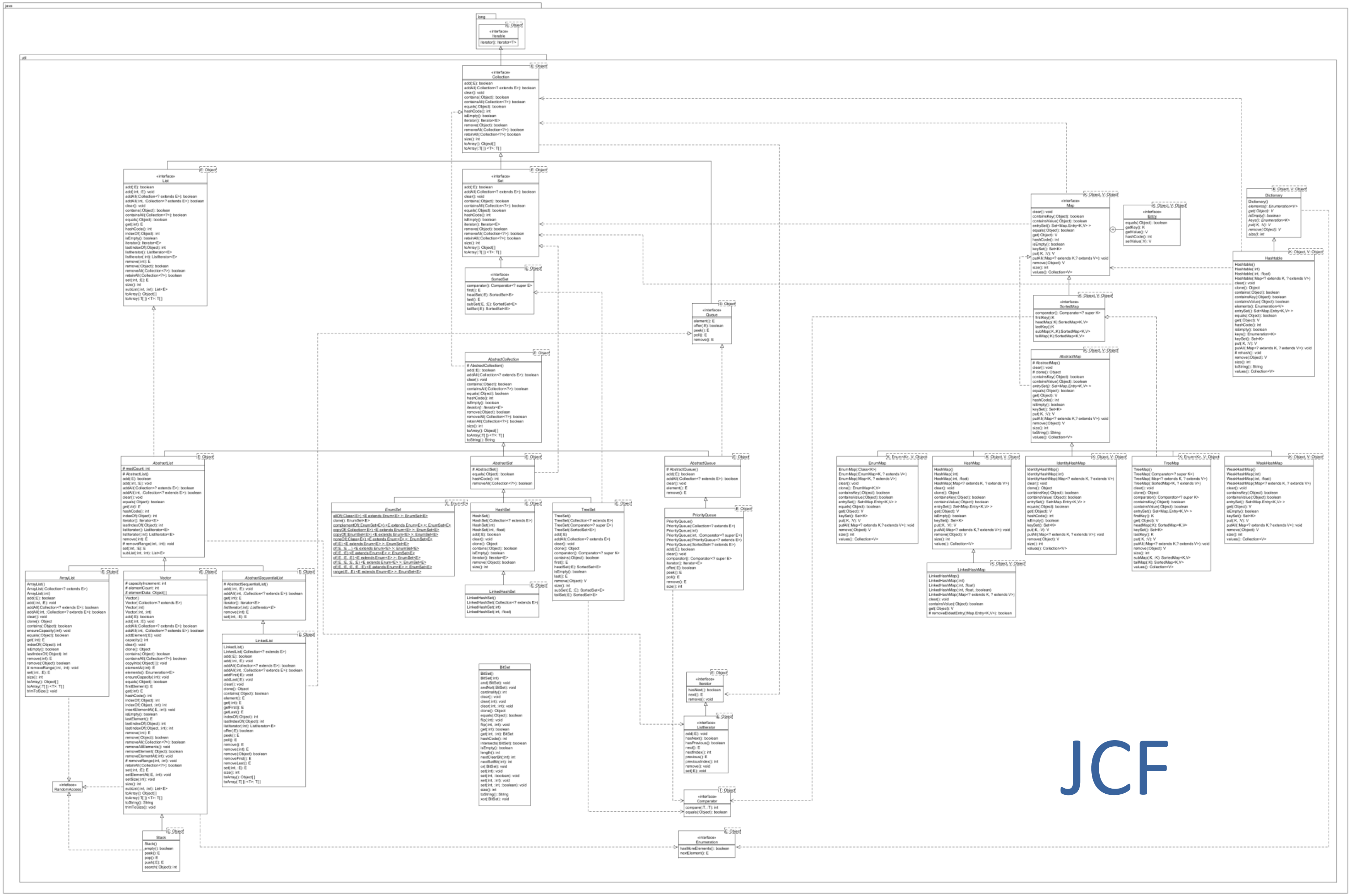
+ ArrayList
+ add(e: T): boolean
+ size(): int
+ get(index: int): T
+ subList(fromIndex: int, toIndex: int): List<T>

regex

+ Pattern
+ «final» LITERAL: int
+ compile(regex: String): Pattern
+ matches(regex: String, input: CharSequence): boolean
+ split(input: CharSequence, limit: int): String[ ]
+ matcher(input: CharSequence): Matcher

+ Matcher
+ replaceFirst(replacement: String): String
+ replaceAll(replacement: String): String
+ quoteReplacement(s: String): String

«exception»
+ PatternSyntaxException


java
lang
«interface» Iterable
util
«interface» Collection
«interface» List
«interface» Set
«interface» SortedSet
«interface» Queue
AbstractCollection
AbstractList
AbstractSet
AbstractQueue
ArrayList
Vector
AbstractSequentialList
LinkedList
Stack
«interface» RandomAccess
EnumSet
HashSet
TreeSet
LinkedHashSet
PriorityQueue
BitSet
«interface» Iterator
«interface» ListIterator
«interface» Comparator
«interface» Enumeration
«interface» Map
«interface» Entry
Dictionary
Hashtable
«interface» SortedMap
AbstractMap
EnumMap
HashMap
IdentityHashMap
TreeMap
WeakHashMap
LinkedHashMap

JCF

Agenda

- Structural models
  - ~~Syntax~~
    - ~~Objects and their relationships~~
    - ~~Classes~~
    - ~~Class relationships~~
    - ~~Interfaces~~
    - ~~Packages~~
  - ~~Practice~~
  - Purpose

Purpose

- What was the purpose of each of those diagrams? What was I trying to model?
- In part, I was trying to show you the importance of **abstraction**
  - I imagine that those diagrams are rather overwhelming … and that is the point: is someone going to be able to interpret something from them?
    - “JCF is big”, “String is a lot more complicated than it should be” … that’s about it
- This is why using an automated diagram-generation tool is typically a **BAD** idea

Purpose

- Common mistake:
  - Turn off brain, attempt to represent every detail
- Why is that a problem?
  - Too much detail is overwhelming
  - Too much detail will obscure what is important
- “What is important” depends on your purpose
  - Demonstrate API of String
  - Illustrate internal complexity of String
  - Illustrate coupling to other types of String

All are reasonable goals, but very different ones demanding different diagrams

(I do not mean “different KINDS of diagram”!)

Purpose

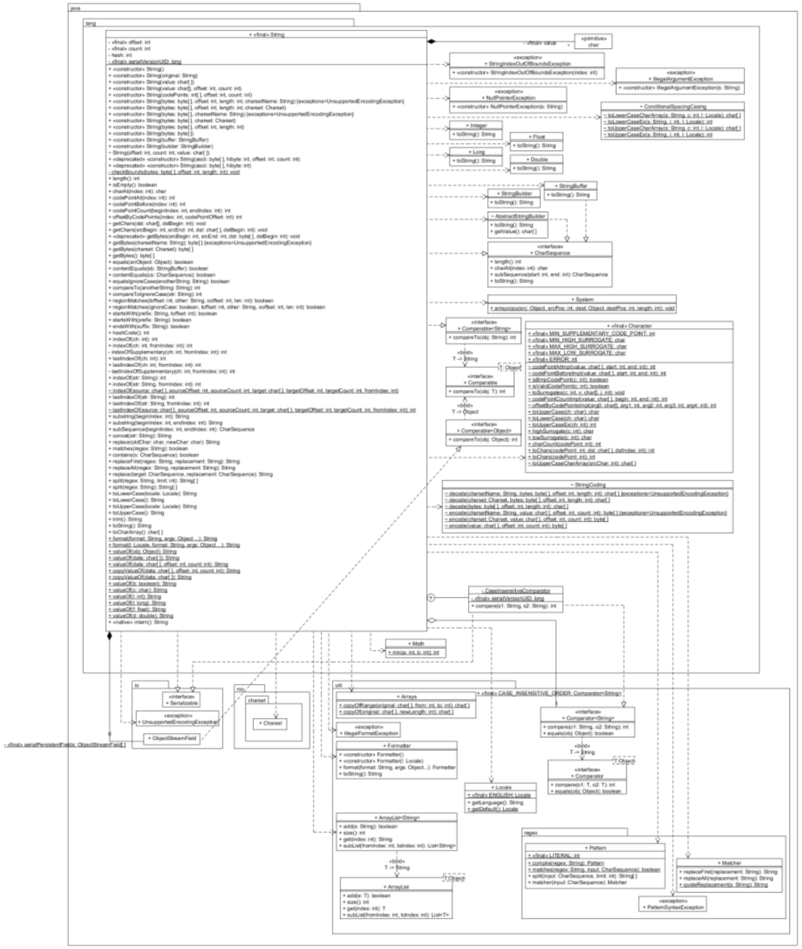
- “Illustrate type hierarchy of JCF”
  - Suppress methods
  - Suppress dependencies
  - Suppress visibility modifiers (and only show public types)
- If that is still too much…
  - eliminate “unimportant types”
  - RandomAccess, BitSet, Iterable …

Reasoning

- Models are only useful if they allow us to reason about (i.e., analyze) their implications
- Some typical questions one wants answered:
  - How does it work?
  - Is there anything missing?
  - Why was it done like this?
  - Are there other alternatives?
- Many such questions are very hard to answer reliably
  - We will consider some of them within the course

Reasoning

- Consider this diagram again:
  1. Does String depend on Comparator?
  2. Can String legally call subSequence(:int, :int) on an instance of StringBuffer?
  3. Does String conform to the CharSequence interface?
  4. If Character.lowSurrogate() has a bug, which methods of String will be affected?
  5. Can String call the compare method on CaseInsensitiveComparator?
- In all cases, why or why not?



Next time

- Behavioural modelling: Sequence diagrams