→ Many systems in the economy are software-controlled
→ Expenditure is significant 60% = dev, 40% = testing
→ Costs more to maintain than to develop
→ SWE ⇔ Cost-effective software dev
→ SWE: Eng discipline for SW production → project management & tool dev
→ Fundamental SWE activities: Specification, Development, Validation, Evolution
→ Challenges against SWE: Increasing Diversity, demand for delivery, and developing trustworthy software
→ Software Products:
→ Generic Products: Stand-alone products marketed & sold to any customer (Ex: CAD, Graphics, etc) ↗ specification of functionality & change = DEV
→ Customized Products: Tailored to target audience (Ex: Embedded Controls) → specification on functionality & change = customer
→ SAP ERP: Enterprise Resource Management
→ Attributes of Good Software: Maintainability, Dependability & Security, Efficiency, Acceptability
→ Issues affecting software: Heterogeneity, Business & Social change, Security & Trust, scale of development

Failure of Project comes from
1. Increasing system complexity — less reliable
2. Failure to use SWE methods → more expensive

→ Software: Computer programs and documentation developed for customer or market
↳ Good software has: functionality, performance, maintainability, dependability and reusability

→ SWE vs Systems Eng:
Systems Eng = Hardware + Software
SWE ⊆ Systems Eng

Application Types:
1. Stand-alone apps: Run on local PC
2. Transaction Apps: Run on remote computers or accessed by PCs ↗ Ex: Ecommerce
3. Embedded Controls system: control/manage hardware devices
4. Entertainment systems
5. Scientific systems for modelling & simulation
6. Data Collection Systems
7. System of systems
8. Batch processing: Business to process data in large batches.

→ Software Processes: Structured set of activities → production of SW systems
1. Plan in advance (waterfall)
2. Incremental/Agile Planning: easier to adapt project to feedback

→ Planning: Professional SW Dev is a planned & managed activity
→ Waterfall Model: Plan-driven (distinct phases)
→ Integration & Config: system assembled from existing components
→ Agile Dev: Specification, Development & Validation interleaved

Waterfall: Used for embedded systems, critical systems & large systems
Requirements Analysis → Systems & Software design
↓
Implementation & Unit testing
Operation & Maintain ←

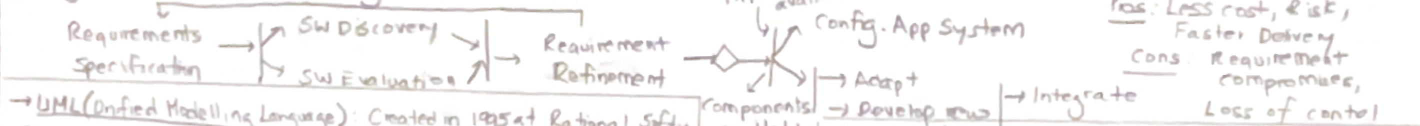
Problems of Waterfall: Inflexible partitioning, Assembly-like, difficult to adapt to change

Agile Dev: Interleaved activities with rapid feedback
outline → Specification ⇄ Initial Vers
Dev ⇄ Intermediate Vers
Validation → Final Vers.

Pros:
↳ Easier for feedback
↳ Cost is reduced
↳ Rapid delivery

Cons:
↳ Process not visible
↳ System structure degrade
↳ Larger Team

Integration & Configuration: Reuse existing systems or components ↗ Need to be integrated into new environment based on requirements

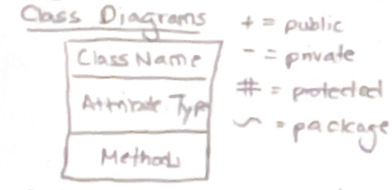



Pros: Less cost, & risk, Faster Delivery
Cons: Requirement compromises, Loss of control

→ UML (Unified Modelling Language): Created in 1995 at Rational Software as graphical notation to visualize design of system (language independent)

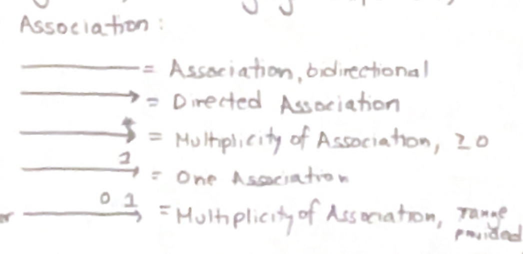
Benefits: Quick sketch, Helps OOP & Communication, document existing SW

Class Diagrams

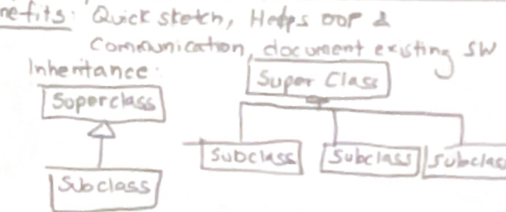



+ = public
− = private
# = protected
~ = package

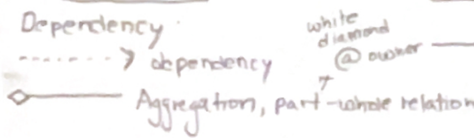
Association:

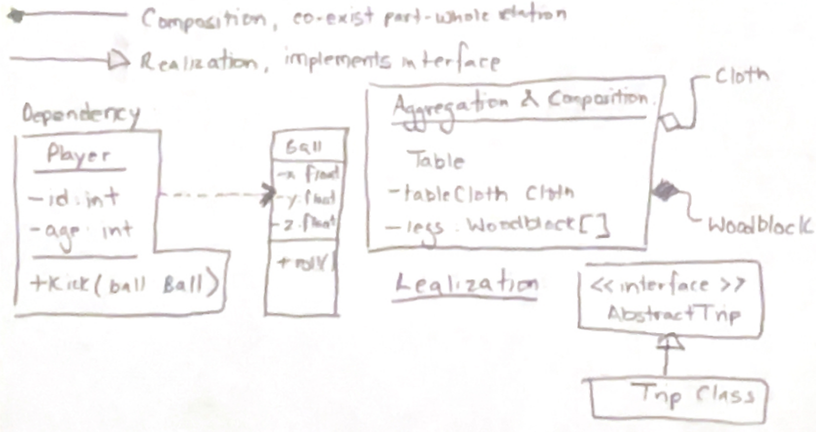



Inheritance:

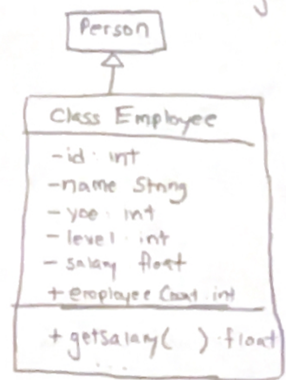



Dependency




Ex: UML Class Diagram



## Primitive Data Types

int long short byte
float double char boolean
individual character storing
int ASCII/unicode val
\b → Backspace → \u0008
\t → tab \u0009
\n → line feed → \u000A
\r → carriage return → \u000D
\" → Double Quote → \u0022

Hex int hex = 0x10
Oct int oct = 010
Bin int bin = 0b1
→ use final → constant
→ static → class scope
\' → single quote → \u0027
\\ → Backslash → \u005C
\s → space → \u0020
\newline

enum Enum = { VAL1, ... }
Enum e = Enum.VAL1

==, !=, >=, >, <=, <
&&, !, ||

## Arithmetic Operators:

+, -, *, /, %

Cast: newType val = (newType) val

byte → short → int → long
char → int; int → float → double

Conditional operator: cond1 ? exp1 : expr2
Ex. int x = 4
int y = x+=4
y = 8 x = 8
++x, x++, --x, x--
int x = 1
x += 3.5 ≡ x = (int)(x+3.5)
x = 4

## Reading Input

import java.util.*
Scanner in = new Scanner(System.in)
String a = in.nextLine()
.nextInt()
.next()

## Format output

System.out.printf(format, var)
d = decimal int
x or X = Hex
o = oct
f or F → float
g or G = general float
e or E = exp float
h or H = hashcode
a or A = hex float
c or C = character
s or S = String
b or B = Boolean

## Operator precedence

[] . () method call (l → r)
! ~ ++ -- + - () cast new (r → l)
* / % (l → r)
+ - (l → r)
<< >> >>> (l → r)
< <= > >= instanceof() (l → r)
== != (l → r)
& (l → r)
^ (l → r)
| (l → r)
&& (l → r)
|| (l → r)
?: (r → l)
= += -= *= /= %= |= ^= <<= >>= >>>= (r → l)

## Strings:

String ex = "hello";
Concat using "+" operator
• substring(first, end)
• length()
• equals(anotherString)
or
str1 == str2
↳ refers if refers to same str

can automatically type cast
→ Are immutable
~~"String"[2] =~~

Empty String → String str = "",
Test Empty: if str.isEmpty()
if str.length() == 0
NULL if str == null

Use StringBuilder to make Strings mutable
StringBuilder strBd1 = new StringBuilder(value);
strBd.append(value)
.setCharAt(index, val)
supports no parameters

## Switch cases

switch(program) {
case "undergrad":
→ ...
}
✓ fallthrough

switch(program) {
case "undergrad":
→ ...
break;
↳ no fallthrough

## Block Scope:

{ } code block
Vars defined in { } are only valid there.

Function Overloading: same f^n name diff parameters

## Big Numbers:

BigInteger bi = BigInteger.valueOf(100);
BigInteger bi = new BigInteger(valueAsStr);
BigDecimal bf = new BigDecimal(valueAsStr);
bi = bi.add(anotherBiobject),
.subtract()
.multiply()

Watch out for initialization blocks!!

## Arrays:

type[] name = new type[size]
type[] existing = { ... }
type[] newArr = new type[] { ... }
newArr = oldArr → copies array

Arrays.copyOf(array, size); copies values
Arrays.toString()
arr.length

## Fields & Methods:

public → accessible by all
private → class
protected → subclass & package

## Initialization Order

1. All fields set to default
2. All initializers & initialization blocks executed in order of pos. in class
3. Body of constructor executed

## Encapsulation

→ Private fields with getters & setters

## Constructors:

→ no return type
→ can be overloaded
→ called with new
→ If you make a constructor w/ parameters, default constructor is disabled
→ calling this() in constructor helps to invoke another constructor

Static Field: data belonging to class
↳ To access: in class → Class field
outside class: object field, Class.field

instance method → invokes instance method
→ access instance field
→ invoke static field
→ access static field

Static Methods: only linked to static fields, don't have access to this

static method ✗ — invoke instance method
✗ — access instance field
invoke static method
access static field

## Inheritance

→ Use "extends"
→ Subclass inherits public or protected fields, getters & setters
→ To initialize superclass, call super(argument)
→ If you call super(), Superclass needs no-argument constructor

Implicit cast: Sub → Super
Class A obj = new Class B();
Explicit cast: Super → Sub
Class A obj1 = new ClassB();
Class B obj2 = (ClassB) obj1;

## Dynamic Vs Static Binding

DB → At run time, calls right function (method overriding)
SB → At compile time, knows which method to call (method overloading)

Final classes cannot be extended → methods are final but not fields
Final methods cannot be overrided

## Interface

→ Methods are implicitly public abstract
→ Fields are implicitly static final
→ Classes implement interfaces

## Abstract Classes

→ Classes that cannot be instantiated
→ Methods only have declaration
→ Class can extend ONLY 1 Class but implement many interfaces

class → class = extends
class → interface = implements
interface → interface = extends

## Enum Classes

```
public enum Size {
    SMALL("S"), MEDIUM("M");
    private String abbreviation;
    Size(String abbrev) {
        this.abbreviation = abbrev;
    }
}
```

Enum.valueOf(Enum.class, String); static
Enum Names[] = EnumName.values(); static
toString() return name of enum constant
ordinal: pos of enum constant ↖ 0 index

## Sealed Class:

→ controls which classes may inherit from it
→ sealed classes can ONLY be extended to final, unsealed or sealed classes that it has permitted

## Static Inner class

```
public class O {
    static class I {
        ...
    }
}
O.I i = new O.I()
```

## Inner Class

```
class O {
    class I {
        ...
    }
}
O o = new O();
O.I i = o.new I();
```

## Local Inner Class

LocalClass in a Method used the same generic way

```
Person p = new Person() {
    void func() { ... }
}
p.func()
```
← Person is abstract/interface

## Functional Interface

```
@FunctionalInterface
public interface Earner {
    void oneFunctionOnly();
}
```

Lambda can only use functional interfaces!

```
Earner earner = () -> {
    Sysout("...");
};
earner.oneFunctionOnly();
```