Object Oriented Systems

CSM-203

Module-I

Introduction

What is Functional Programing?

- Functional programming is a programming style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- It is a declarative programming paradigm in that programming is done with expressions or declarations instead of statements.
- The output value of a function depends only on its arguments, so calling a function with the same value for an argument always produces the same result.
- E.g – Haskel etc.

Contd.

- Functional programming supports higher-order functions
- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.
- The efficiency of a programming code is directly proportional to the algorithmic efficiency and the execution speed. Good efficiency ensures higher performance.

Comparison between OOP & FP

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior"	Supports only "Abstraction over Data".

Example

```
#include <stdio.h>

int addNum(int a, int b);     // function prototype

int main() {
   int sum;
   sum = addNum(5,6);         // function call
   printf("sum = %d",sum);
   return 0;
}
int addNum (int a,int b) {    // function definition
   int result;
   result = a + b;
   return result;             // return statement
}
```

Algorithms using Conditions

- An algorithm is a step-by-step solution to a given problem. An algorithm has the following properties:
  - finiteness - the process terminates, the number of steps are finite
  - definiteness - each step is precisely stated
  - effective computability - each step can be carried out by a computer

Contd.

- In a conditional statement , The result of the test is a Boolean - either True or False. If the result of the test is True a certain course of action is taken and if the result of the test is False another course of action is taken
- IF condition THEN
- sequence 1
- ELSE
- sequence 2
- ENDIF
- If the condition is True sequence 1 is executed, otherwise sequence 2 is executed. The ELSE sequence is optional.

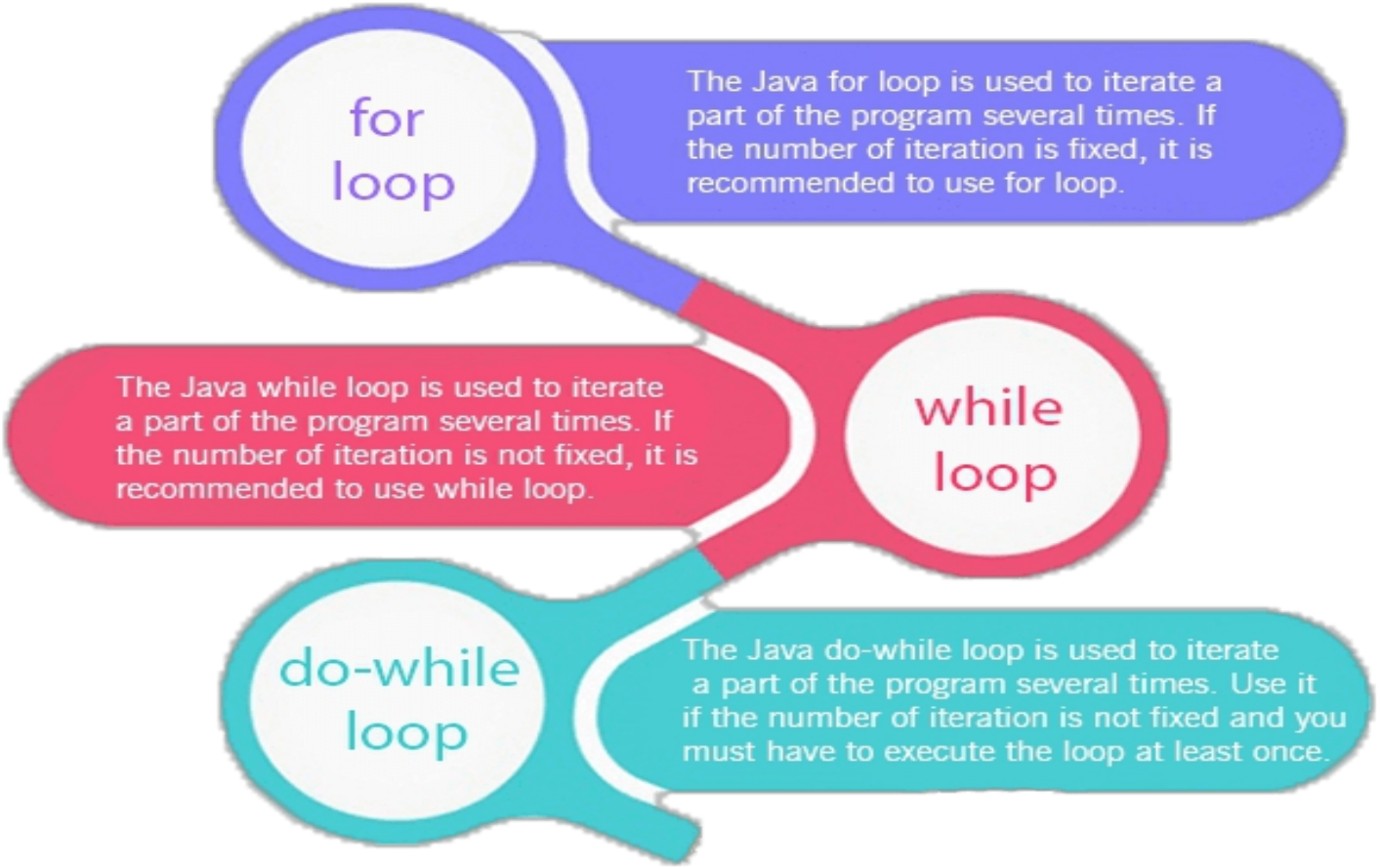
Example

```
//Java Program to demonstate the use of if –else statement.
public class LeapYearExample {
public static void main(String[] args) {
    int year=2020;
    if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
        System.out.println("LEAP YEAR");
    }
    else{
        System.out.println("COMMON YEAR");
    }
}
}
```

Algorithms using Loops

- Loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.
  - for loop
  - while loop
  - do-while loop

Contd.


for loop
The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
while loop
The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.
do-while loop
The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	for(init;condition;incr/decr){ // code to be executed }	while(condition){ //code to be executed }	do{ //code to be executed }while(condition);
Example	//for loop for(int i=1;i<=10;i++){ System.out.println(i); }	//while loop int i=1; while(i<=10){ System.out.println(i); i++; }	//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);
Syntax for infinitive loop	for(;;){ //code to be executed }	while(true){ //code to be executed }	do{ //code to be executed }while(true);

Basic IO

- Java IO is an API that comes with Java which is targeted at reading and writing data (input and output).

Example

- system.in - the standard input stream
- system.out - the standard output stream
  - print
  - println
  - printf
- system.err - the standard error stream
  - print
  - println
  - printf
- etc.

Contd.

- The two keywords cout and cin in C++ are used for printing outputs and taking inputs respectively.
- Header Files in C++ for I/O operations are,
  - **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.
  - **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
  - **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

Object-Oriented Design and Analysis using UML

- Object-Oriented Design is a software development approach to design and implement software system as a collection of interacting stateful objects with specified structure and behaviour.

Analysis & Design

- Analysis emphasizes an investigation of the problem and requirements
- requirement analysis is an investigation of the requirements and object analysis is an investigation of domain objects
- Design emphasizes a conceptual solution that fulfils requirements, ultimately the design may be implemented

OO analysis & Design

- During object oriented analysis emphasizes on finding and describing the objects in the problem domain
- Examples: in case of library information system, some of the objects include Book, Library etc.
- During Object Oriented Design emphasizes on defining software objects and how they collaborate to fulfill the requirements
- Example: in the library system, a Book software object may have a title attribute and a getChapter () method
- Finally, during implementation the objects are implemented, such as a Book class in C++

Contd.

Domain object

Book

title

Visualization of Domain object

```
class Book{
    String title;
    public:
    chapter getChapter(int);
};
```

Implementation

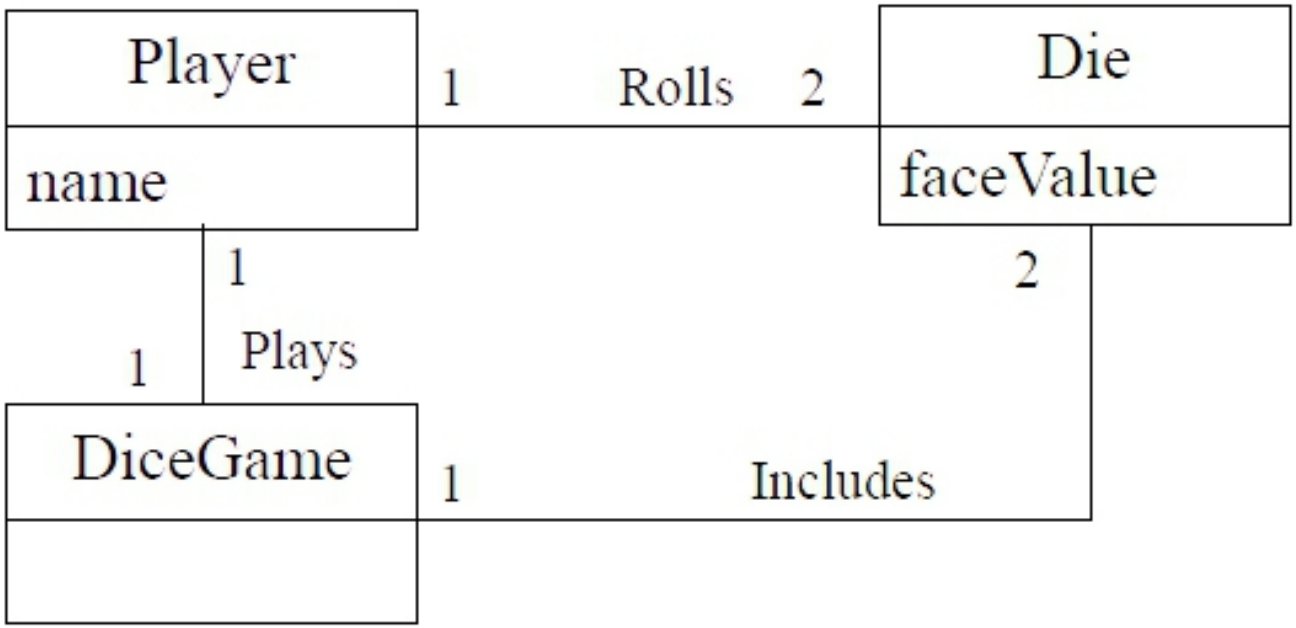
OOAD

- The steps for object oriented analysis and design may be as follows
  - Define Use cases
  - Define domain model
  - Define interaction diagram
  - Define design class diagram

Example

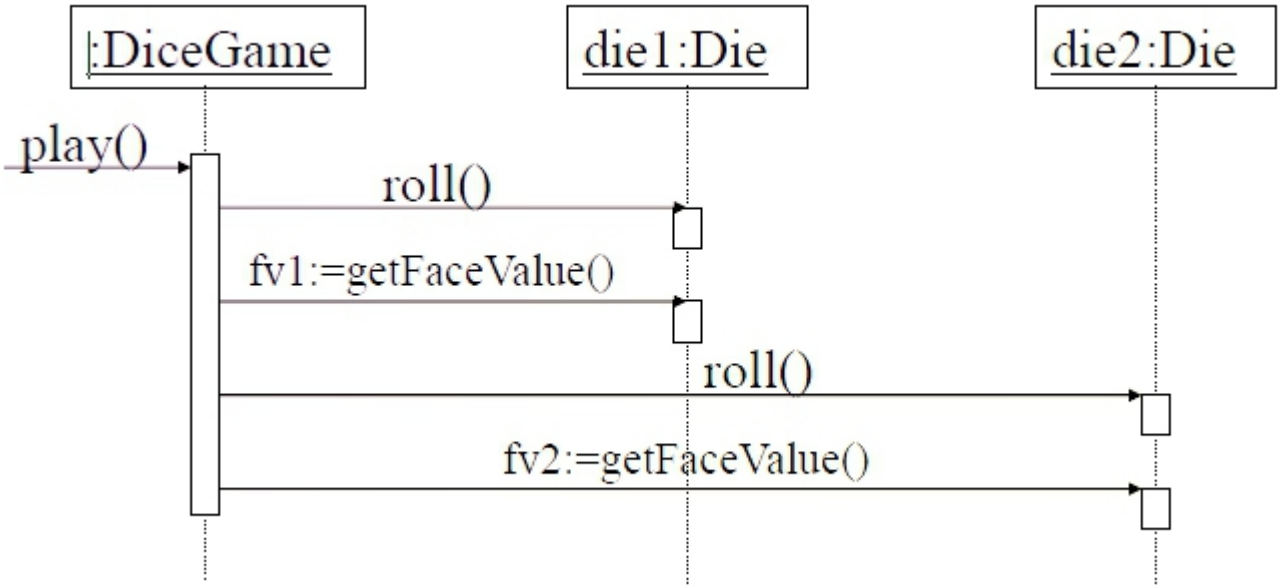
- A “Dice Game” : A player rolls two die
- Usecase: description of related domain process
  - Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose
- Define domain model: Decomposition of the domain involves identification of objects, attributes and associations.
  - Represented in a diagram

Defining Domain Model



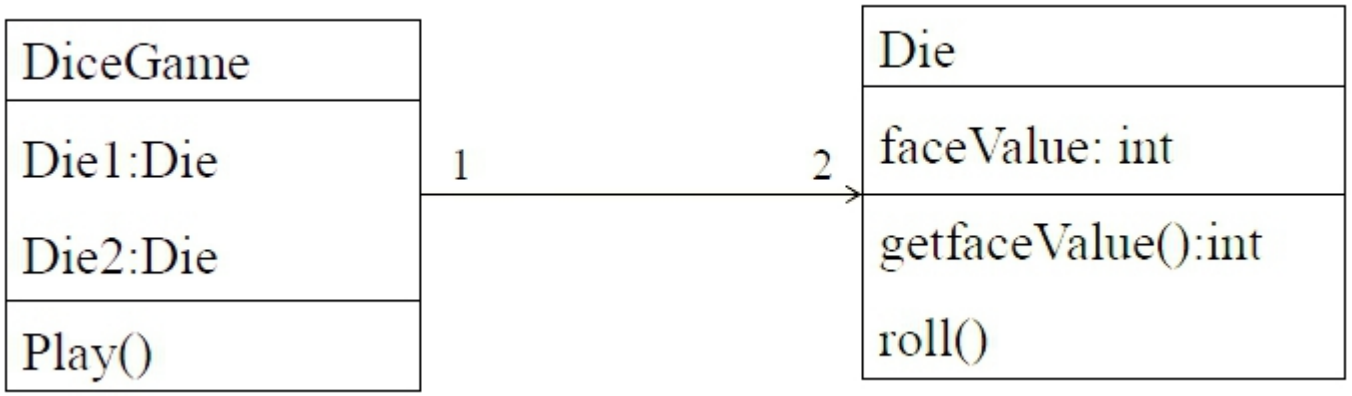
Define Interaction diagram

- Defining software objects and their collaborations.
- This diagram shows the flow of message between software objects and thus invocation of methods



Define design class diagrams

- To create a static view of the class definition with a design class diagram.
- This illustrates the attributes and methods of the classes.



Unified Modelling Language (UML)

- The Unified Modelling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- It simplifies the complex process of software design, making a "blueprint" for construction.

Purpose of Modelling?

- Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building.
- Good models are essential for communication among project teams and to assure architectural soundness.
- As the complexity of systems increase, so does the importance of good modelling techniques. There are many additional factors of a project's success, but having a rigorous modelling language standard is one essential factor.

Building blocks of UML

- Building blocks of UML constitutes
  - Things
  - Relationships
  - Diagrams

Things

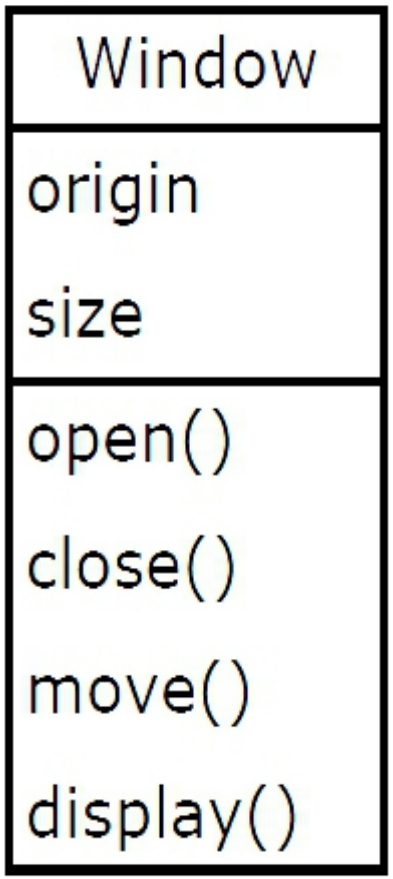
- There are four kind of things defined
  - Structural things
  - Behavioral things
  - Grouping things
  - Annotational things

Structural Things

- These are nouns of UML model
- These things represents elements that are either conceptual or physical.
- There are seven kinds of structural things
  - Class
  - Interface
  - Collaboration
  - Use Case
  - Active Class
  - Component
  - Node

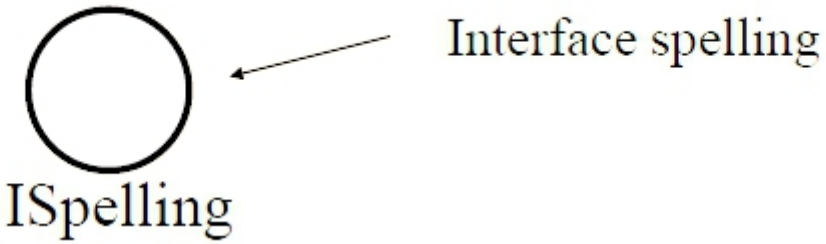
Class

- Is a description of a set of objects that share common attributes, operations, relationships, and semantics.
- A class implements one or more instances
- It is represented by a rectangular box



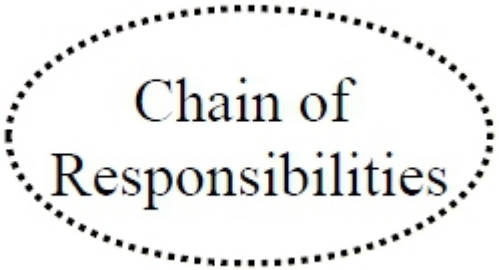
Interface

- Is a collection of operations that specify a service of a class
- An interface might represent the complete behaviour of a class or only a part of that behaviour
- It is represented by a circle, with its name



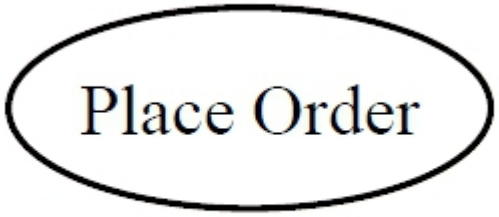
Collaboration

- Defines an interaction and is a society of roles and other elements that work together to provide some cooperative behaviour
- It is represented by an dashed ellipse with its name

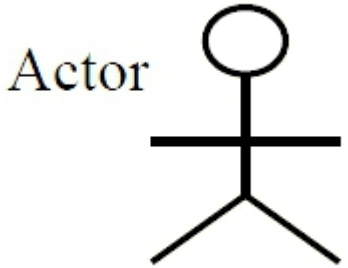


Use Case

- It is a description of sequence of actions that a system performs that yields an observable result of a value to a particular actor
- It is represented by a solid ellipse with its name

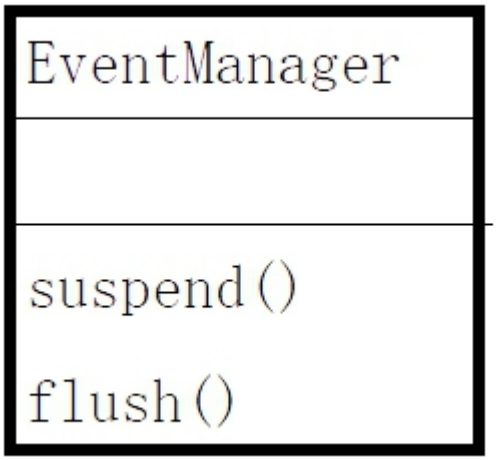


Use case



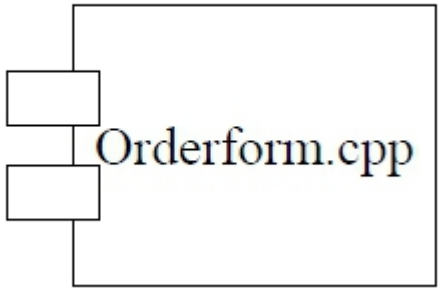
Active Class

- It is a class, whose objects own one or more processes or threads and therefore can initiate control activity
- It is represented by a heavy lined box



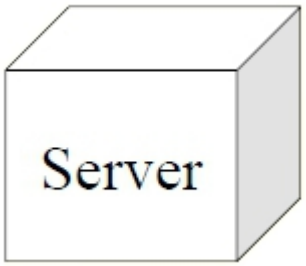
Component

- It is a physical and replaceable part of a system that confirms to and realization of a set of interfaces.
- i.e. physical packaging of classes, interfaces and collaborations



Node

- It is a physical element that exists in run time and represents a computational resource, generally having some memory and processing capability
- A set of components may reside as a node and may also migrate from node to node
- It is represented as a cube



Behavioural Things

- These are dynamic parts of UML model
- These are verbs of a model, representing behaviour over time and space
- There are two types of Behavioural things
  - Interaction
  - State Machine

Interaction

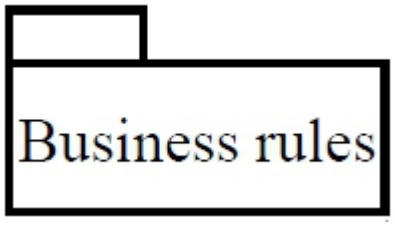
- It is a behaviour that comprises a set of messages exchanged among a set of objects to accomplish a specific purpose.
- An interaction involves messages, action sequences and links. (connection between objects)
- It is represented by a solid arrow

State Machine

- It is a behaviour that specifies the sequence of states of an object or an interaction goes through during its life time in response to events, to gather with its responses to these events.
- A state machine involves
  - States
  - Transitions (flow from one state to another)
  - Events (things that trigger a transition)
  - Activities (response to transitions)
  - It is represented with a rectangular box having rounded corners

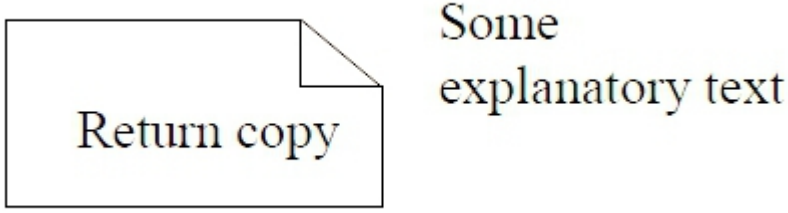
Grouping Things

- These are organizational part of UML models
- There is one primary kind of grouping thing called packages
- Package
  - It is a general purpose mechanism for organizing elements in to groups
  - It is rendered as a tabbed folder



Annotational Things

- These are explanation parts of UML
- Note is a annotational thing called
- Note
  - It is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

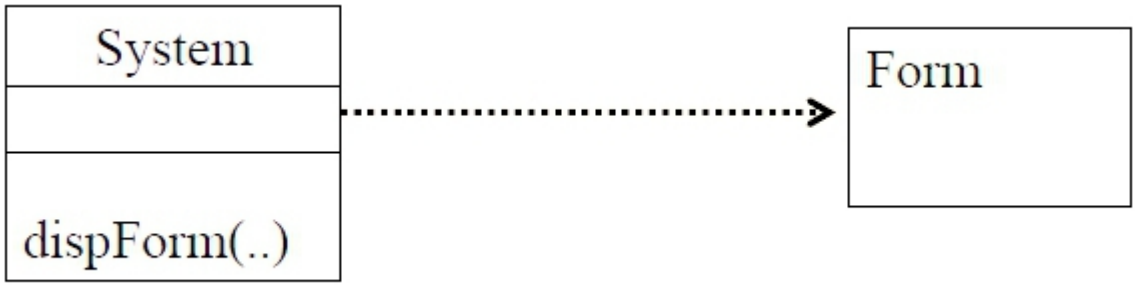


Relationships

- A relationship is a connection among things
- It is rendered as a path, with different kind of lines used to distinguish the kind of relationships
- There are 4 kinds of relationships
  - Dependency
  - Association
  - Generalization
  - Realization

Dependency

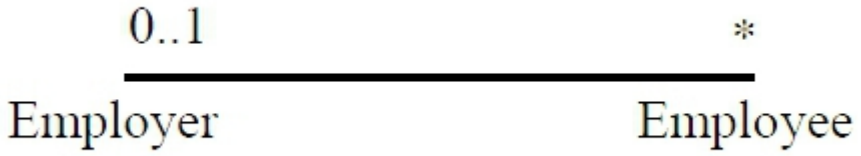
- It is a semantic relationship between two things in which a change to independent thing may affect the semantic of the other thing
- It is rendered as a directed dashed line



The form the system displays obviously depends on which form the user selects

Association

- It describes a set of links
  - Aggregation is a special kind of association, representing a structural relationship between a whole and its parts
  - It is rendered as a solid line, occasionally including a label and other specifications

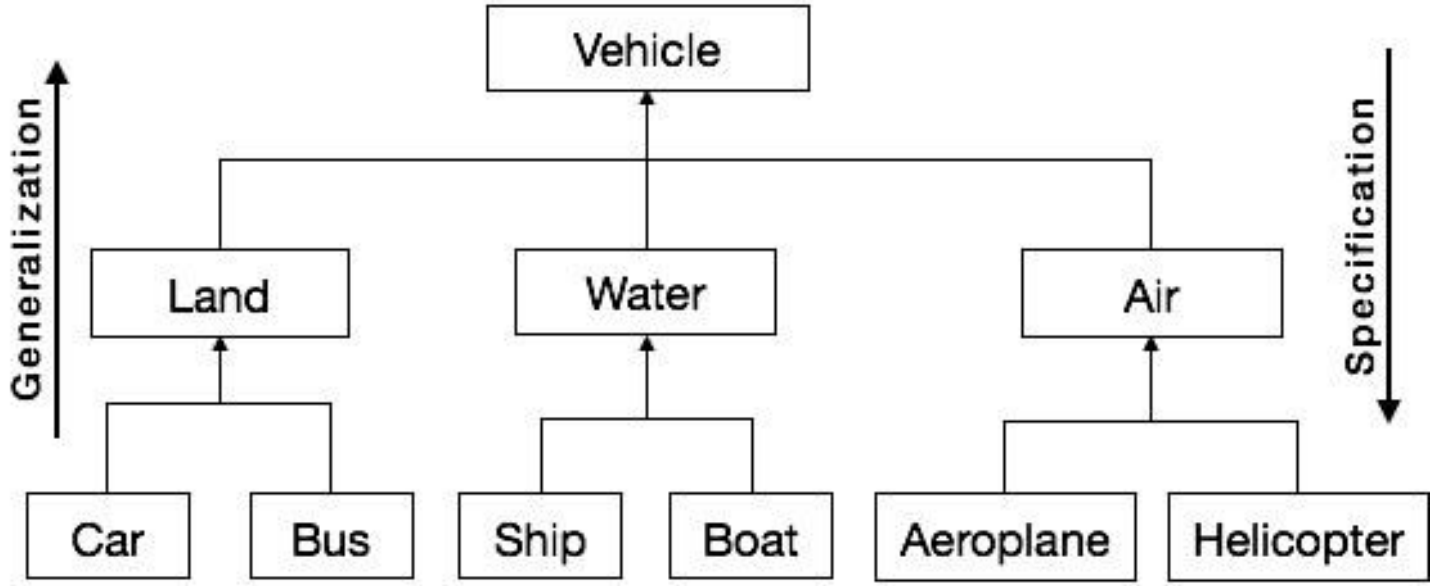


Generalization/Specialization

- Child shares the structure and behaviour of the parent
  - Objects of specialized elements are substitutable for the objects of generalized elements
  - It is rendered as a solid line with a hollow arrow head pointing to the parent.

Contd.

- Specialization is the reverse process of generalization



Realization

- It is a relationship between classifiers. (polymorphism)
  - One classifier specifies a contract that another classifier guarantees to carry out
  - It is rendered as a dashed line with a hollow arrow head.
  - The base class provides an interface for which the derived class writes an interface

Diagrams

- A Graphical representation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

Contd.

- There are 9 types of Diagrams
  - Class Diagram
    - Shows a set of classes, interfaces, and collaborations and their relationships.
  - Object Diagram
    - Shows a set of objects and their relationships
  - Use Case Diagram
    - Shows a set of use cases and actors
  - Sequence Diagram & Collaboration Diagram
    - These two are interaction kind of diagram. They show a set of objects and their relationships

Contd.

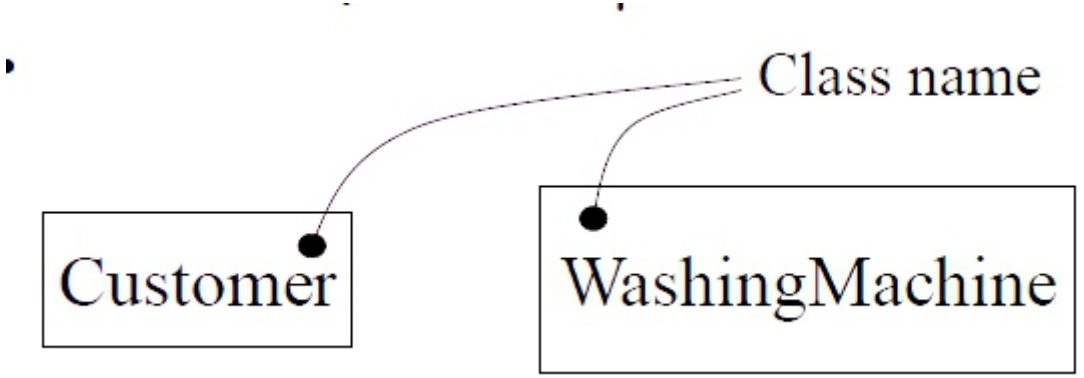
- State chart Diagram
  - Shows a state machine, consisting of states, transitions, events and activities
- Activity Diagram
  - It is a kind of state chart diagram that shows the flow from activity to activity with a system
- Component Diagram
  - Shows the organisation and dependencies among a set of components. It addresses the static implementation views of a system
- Deployment Diagram
  - Shows the configuration of run-time processing nodes and the components that live on them.

Class

- A class is a descriptions of a set of objects, that share attributes, relationships and semantics.
- Graphically a class is rendered as a rectangle
- Name of a class
  - In practice class names are short nouns or noun phrases drawn from the vocabulary of the system to be modelled
  - Every class has a name that distinguishes it from other classes

Name of a class

- Name is a textual string, with first letter of every word capitalized

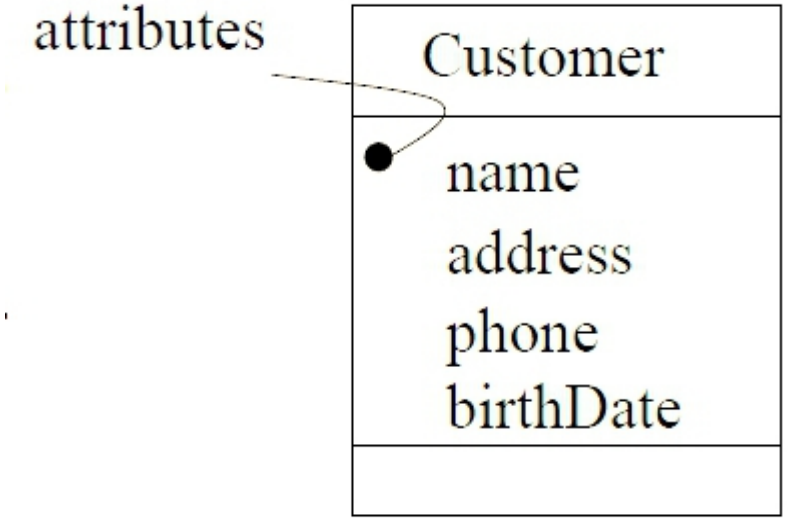


- A path name is the class name prefixed by the name of the package in which that class lives

HouseholdAppliances : : WashingMachine

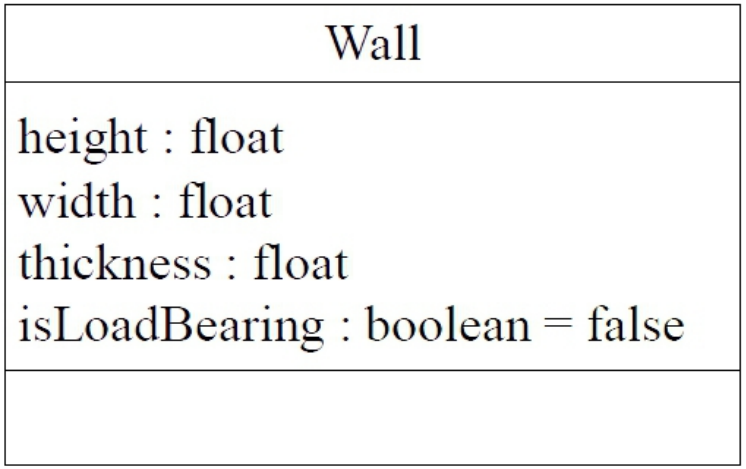
Attributes of a Class

- An attribute is an abstraction of the kind of data or state an object of the class might encompass
- An attribute is a named property of a class
- A class may have any number of attributes or no attributes at all.



Contd.

- Attributes are listed in a compartment just below the class name
- First letter of every word is capital in an attribute name expect the first word
- An attribute may be specified by stating its class and possibly default initial value
- In practice an attribute name is a short noun or noun Phrase that represents some property of its enclosing class
- Example-

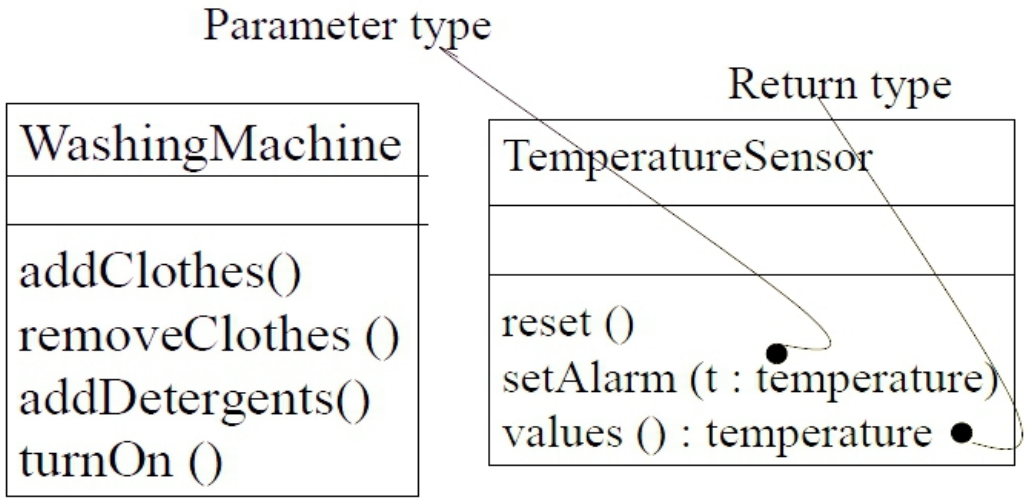


Operations in a Class

- An operation is an abstraction of something, one can do to at an object and that is shared by all objects of that class
- A class may have any no. of operations or no operations at all
- Operation are listed in a Compartment just below The class attribute

Contd.

- The first letter of every word in an operations name is capital, except the first letter of first word
- An operations can be specified By stating its signature, covering the name, type and default values of all parameters and a return type
- In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class

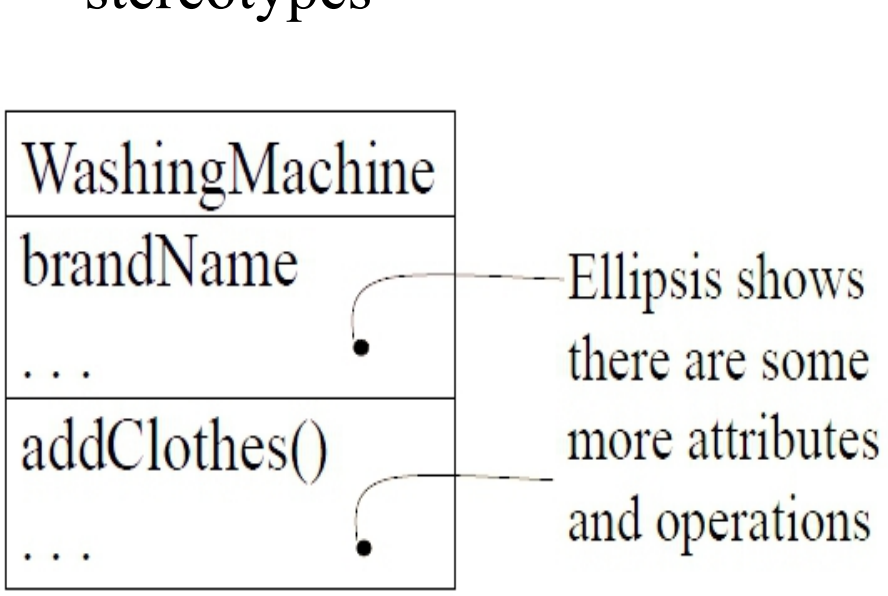


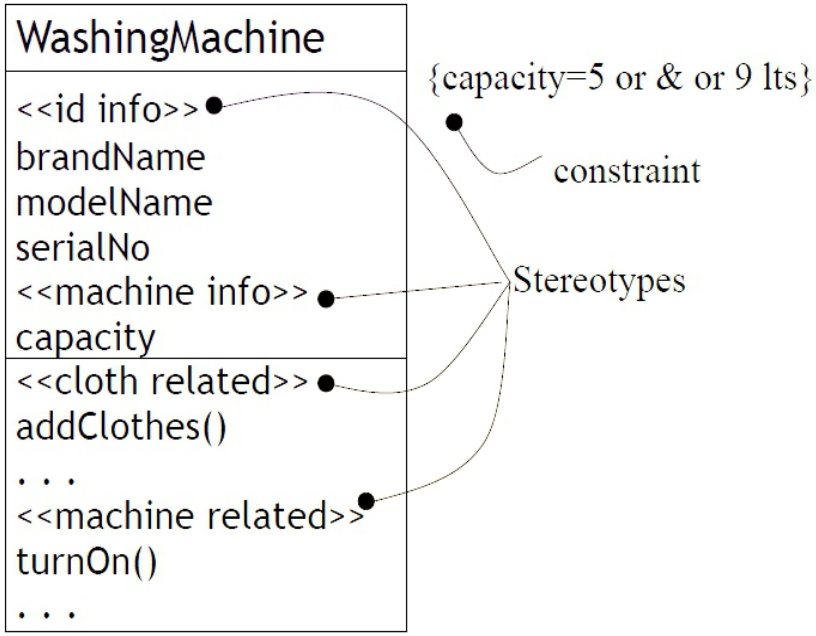
Organising Attributes and Operations

- It may not be possible to list all attributes and operations at once
- Therefore, a class can be elided, i.e. it is possible to show only some or none of attributes and operations
- An empty compartment does not necessarily mean that there are no attributes or operations
- It can be explicitly specified that there are more attributes or operations than shown, by ending each list with an ellipsis.

Attributes & Operations

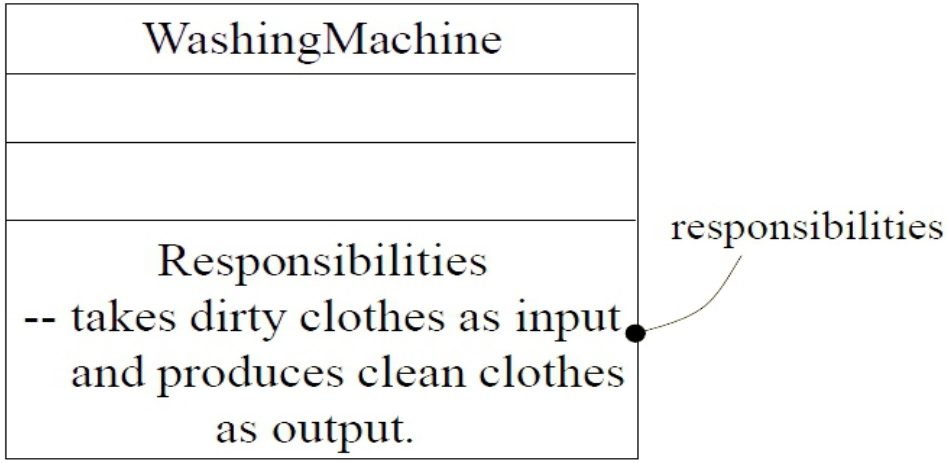
- To better organise long lists of attributes and operations, those can be grouped.
- Each group is prefixed with a descriptive category by using stereotypes





Responsibilities

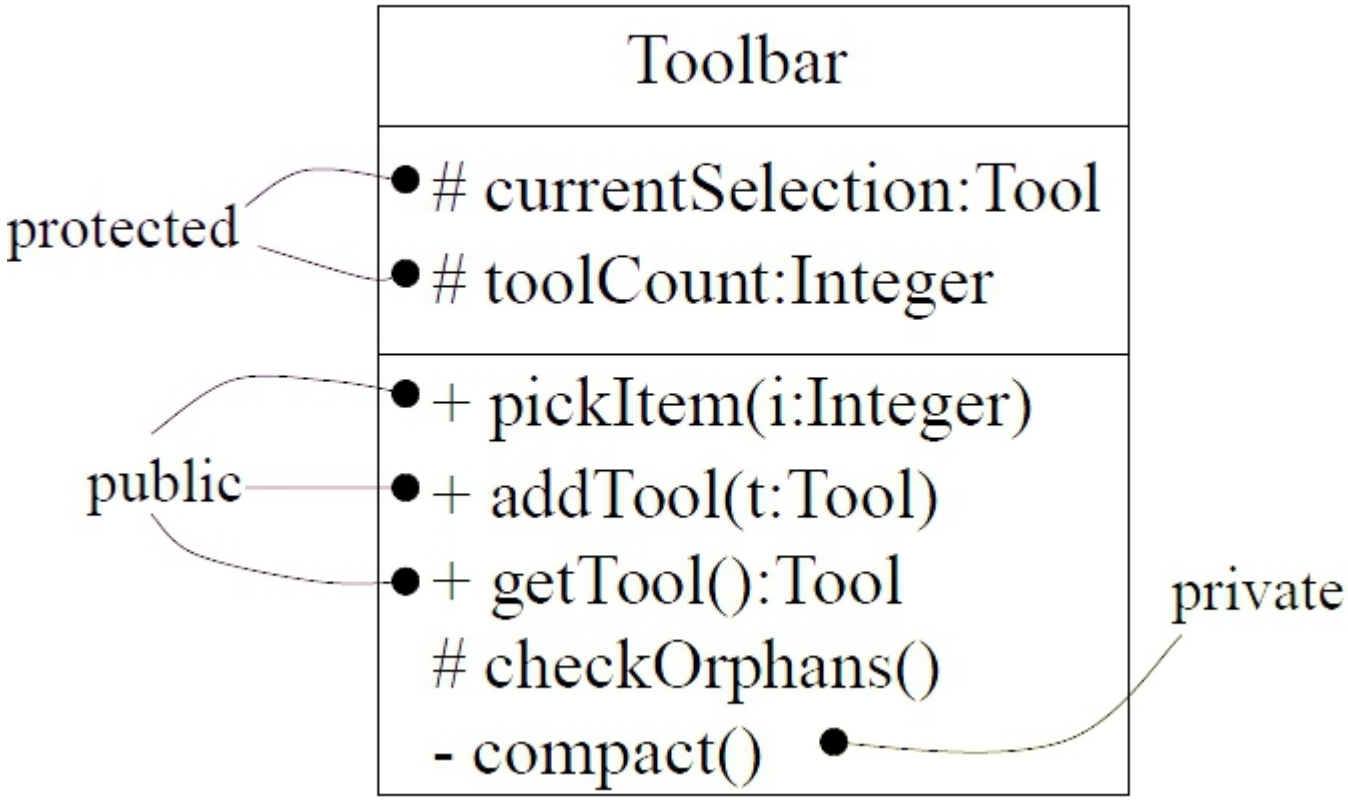
- It is a contract or an obligation of a class. i.e it is a description of what the class has to do
- Responsibilities can be drawn in a separate compartment at the bottom of the class icon
- A single responsibility is written as a phrase, a sentence, or (at most) a short paragraph (free-form text)



Visibility of a feature

- It allows to specify the visibility for a classifier's features.
- Three levels of visibility can be specified using UML
  - Public : any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
  - Protected : Any descendant of the classifier can use the feature; specified by prepending the symbol #
  - Private : only the classifier itself can use the feature; specified by prepending the symbol -

Contd.

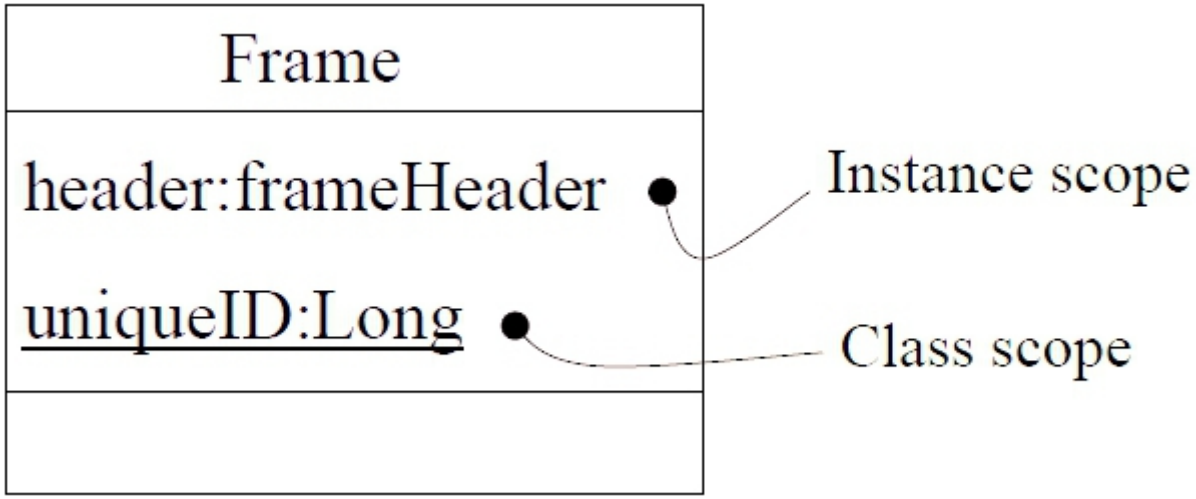

Toolbar
currentSelection:Tool
toolCount:Integer
+ pickItem(i:Integer)
+ addTool(t:Tool)
+ getTool():Tool
# checkOrphans()
- compact()
protected
public
private

Scope of a feature

- The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance for all instances of the classifier
- UML provides two kinds of scope
  - Instances : Each instance of the classifier holds its own value for the feature
  - Classifier : There is just one value of the feature for all instances of the classifier

Contd.

- Classifier scope is rendered by underlining the feature
- In C++ we call the class scoped as static members of a class



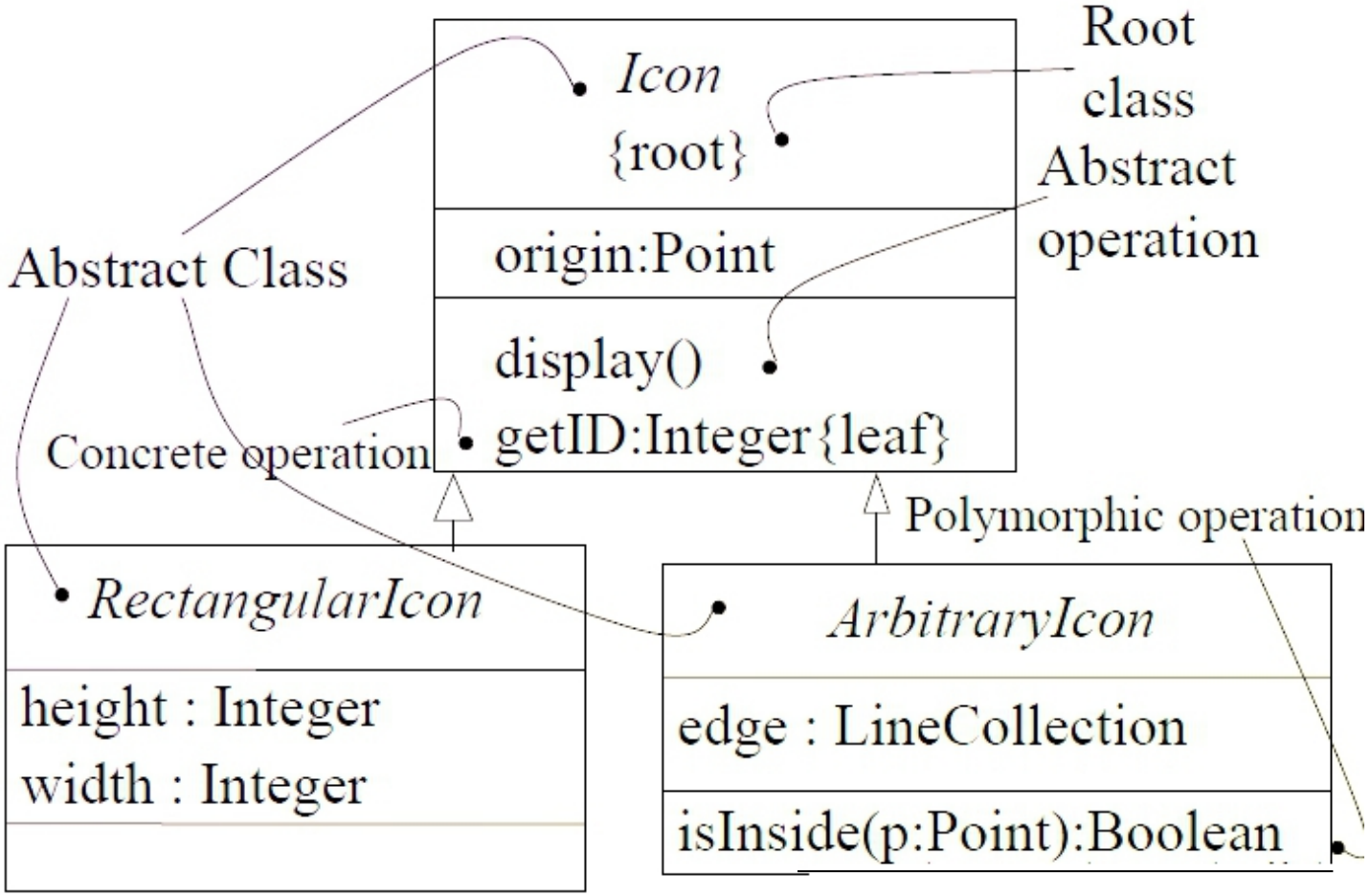
Polymorphic Elements

- Abstract Class
  - Class having no direct instances (italics)
- Concrete Class
  - Class that may have direct instances
- Root Class
  - Class having no parent but may have children {root}
- Leaf Class
  - Class having parent but no children {leaf}

Contd.

- Polymorphic Operations
  - An operation is polymorphic, if it can be specified with the same signature at different points of hierarchy
  - When a message is dispatched at run time, the operation in the hierarchy that is invoked is chosen polymorphycally. i.e. a match is determined at run time according to the type of the object
- Abstract Operation
  - This operation is incomplete and requires a child to supply an implementation. In UML it is specified by writing its name in italics.
- Leaf operation
  - Operation is not polymorphic and may not be overridden

Example

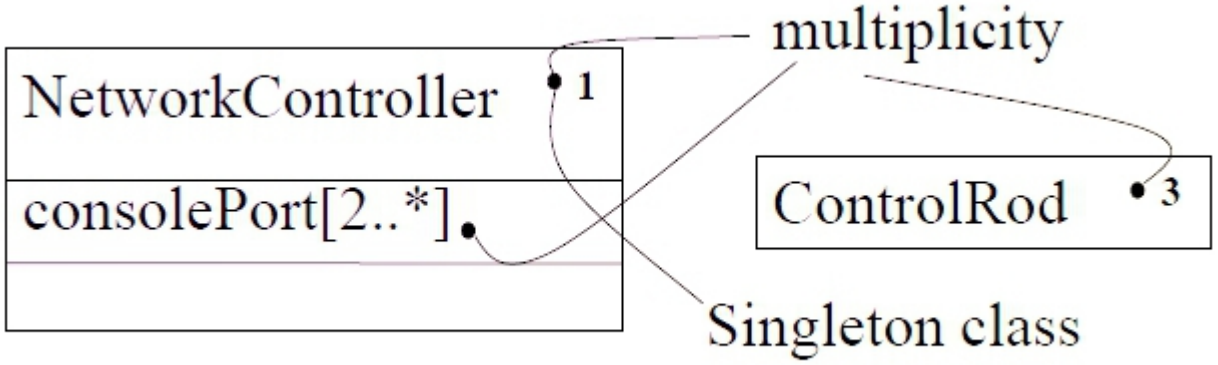

Icon
{root}
origin:Point
display()
getID:Integer{leaf}
Root class
Abstract operation
Abstract Class
Concrete operation
Polymorphic operation
RectangularIcon
height : Integer
width : Integer
ArbitraryIcon
edge : LineCollection
isInside(p:Point):Boolean

Multiplicity

- It is a specification of the range of allowable cardinalities an entity may assume
- Multiplicity of a class
  - It is used to restrict the no of instances of a class
  - i.e. It tells the no of instances a class may have
- Multiplicity of an attribute
  - It is used to represent a set of elements for an attribute
  - i.e. It maps to representing an array variable in C++

Example

- Abstract operations maps to pure virtual functions in C++
- Leaf operations maps to non-virtual functions in C++



Attributes

- In its full form, the syntax of an attribute in UML is
- [visibility] name [multiplicity] [: type]
- [= initial-value] [{property-string}]
- Examples :

Origin	Name Only
+origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplycity, and type

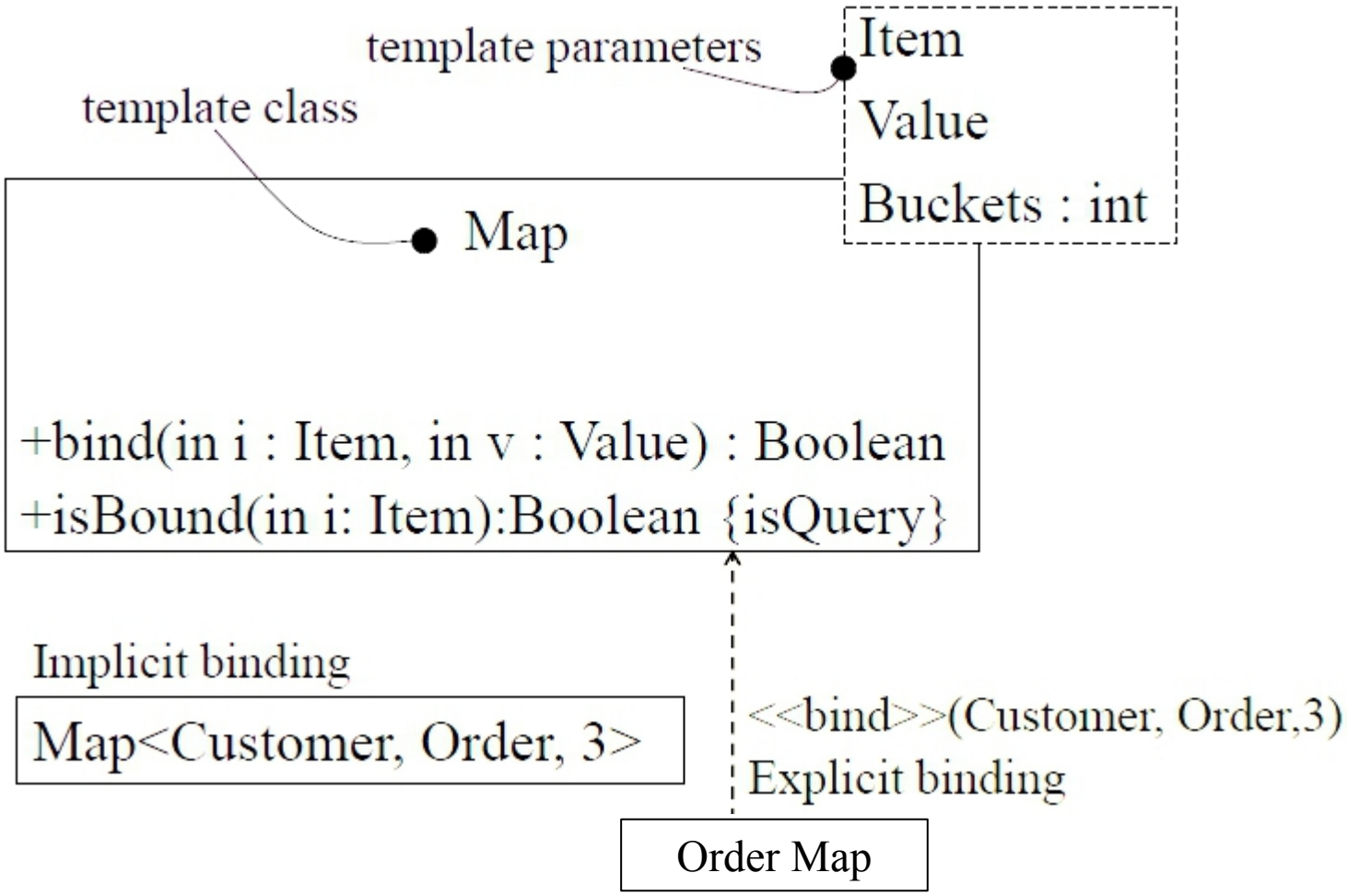
Contd.

- There are three defined properties, that can be used with attributes
  - changeable
    - There are no restrictions on modifying the attribute value
  - addonly
    - For attributes with a multiplicity greater than one, but once created, a value may not be removed or altered
  - frozen
    - The attribute's value may not be changed after the object is initialized(constants)

Template Classes

- Template is a parameterized element
- Template class define a family of classes
- A Template includes slots for classes, objects and values, and these slots serve as the template parameter
- Instance of a template class is a concrete class that can be used just like any ordinary class
- The most use of template class is to specify containers that can be instantiated for specific elements

Example


template parameters
Item
Value
Buckets : int
template class
Map
+bind(in i : Item, in v : Value) : Boolean
+isBound(in i: Item):Boolean {isQuery}
Implicit binding
Map<Customer, Order, 3>
<<bind>>(Customer, Order,3)
Explicit binding
Order Map

Stereotypes for Class

- The UML defines four standard sterotypes that apply to classes
  - Metaclass : specifies a classifier whose objects are all classes
  - Powertype : specifies a classifier whose objects are the children of a given parent
  - Stereotype : classifier is a stereotype that may be applied to other elements
  - Utility : specifies a class whose attributes and operations are all class scoped

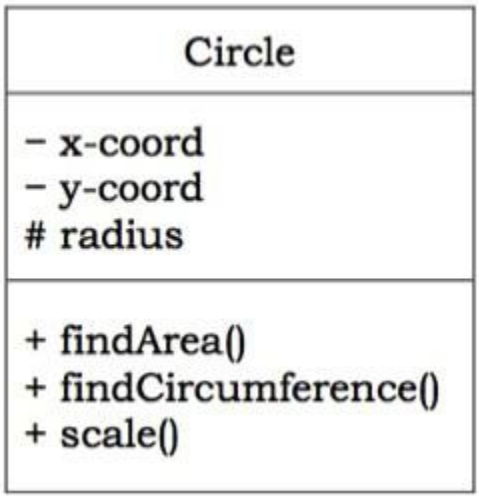
OOAD - Implementation Strategies

Implementing an object-oriented design generally involves using a standard object oriented programming language (OOPL) or mapping object designs to databases. In most cases, it involves both.

Implementation using Programming Languages

Usually, the task of transforming an object design into code is a straightforward process. Any object-oriented programming language like C++, Java, Smalltalk, C# and Python, includes provision for representing classes. In this chapter, we exemplify the concept using C++.

The following figure shows the representation of the class Circle using C++.




```
class Circle
{
   private:
     double x_coord;
     double y_coord;
   protected:
     double radius;
   public:
     double findArea();
     double findCircumference();
     void scale();
};
```

Implementing Associations

Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

Associations may be either unidirectional or bidirectional. Besides, each association may be either one–to–one, one–to–many, or many–to–many.

Unidirectional Associations

For implementing unidirectional associations, care should be taken so that unidirectionality is maintained. The implementations for different multiplicity are as follows −

- **Optional Associations** − Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++ −

```
class Customer {
   private:
   // attributes
   Current_Account c; //an object of Current_Account as attribute

   public:

   Customer() {
      c = NULL;
   } // assign c as NULL

   Current_Account getCurrAc() {
      return c;
   }

   void setCurrAc( Current_Account myacc) {
      c = myacc;
   }

   void removeAcc() {
      c = NULL;
   }
};
```

- **One–to–one Associations** − Here, one instance of a class is related to exactly one instance of the associated class. For example, Department and Manager have one–to–one association as shown in the figure below.

This is implemented by including in Department, an object of Manager that should not be NULL. Implementation using C++ −

```
class Department {
   private:
   // attributes
   Manager mgr; //an object of Manager as attribute

   public:
   Department (/*parameters*/, Manager m) { //m is not NULL
      // assign parameters to variables
      mgr = m;
   }

   Manager getMgr() {
      return mgr;
   }
};
```

- **One–to–many Associations** − Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.

This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container −

```
class Employee {
   private:
   char * deptName;
   list <Dependent> dep; //a list of Dependents as attribute

   public:
   void addDependent ( Dependent d) {
      dep.push_back(d);
   } // adds an employee to the department

   void removeDeoendent( Dependent d) {
      int index = find ( d, dep );
      // find() function returns the index of d in list dep
      dep.erase(index);
   }
};
```

Bi-directional Associations

To implement bi-directional association, links in both directions require to be maintained.

- **Optional or one–to–one Associations** − Consider the relationship between Project and Project Manager having one–to–one bidirectional association as shown in the figure below.

Implementation using C++ −

```
Class Project {
   private:
   // attributes
   Project_Manager pmgr;
   public:
   void setManager ( Project_Manager pm);
   Project_Manager changeManager();
};

class Project_Manager {
   private:
   // attributes
   Project pj;

   public:
```

```
   void setProject(Project p);
   Project removeProject();
};
```

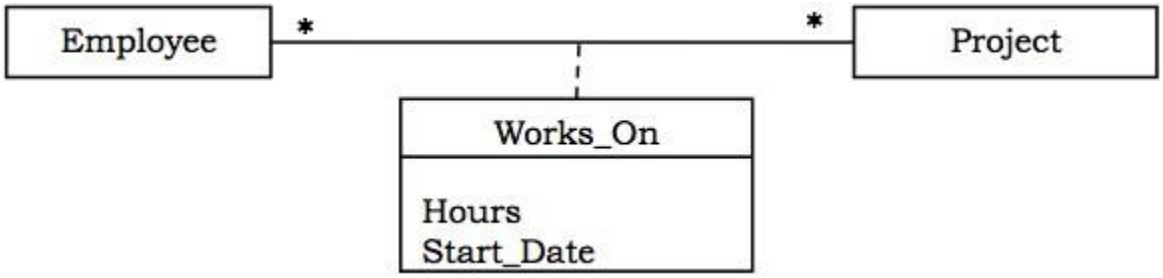
- **One–to–many Associations** − Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

Implementation using C++ STL list container

```
class Department {
   private:
   char * deptName;
   list <Employee> emp; //a list of Employees as attribute

   public:
   void addEmployee ( Employee e) {
      emp.push_back(e);
   } // adds an employee to the department

   void removeEmployee( Employee e) {
      int index = find ( e, emp );
      // find function returns the index of e in list emp
      emp.erase(index);
   }
};

class Employee {
   private:
   //attributes
   Department d;

   public:
   void addDept();
   void removeDept();
};
```

Implementing Associations as Classes

If an association has some attributes associated, it should be implemented using a separate class. For example, consider the one–to–one association between Employee and Project as shown in the figure below.



Implementation of WorksOn using C++

```
class WorksOn {
   private:
   Employee e;
   Project p;
   Hours h;
   char * date;

   public:
   // class methods
};
```

Implementing Constraints

Constraints in classes restrict the range and type of values that the attributes may take. In order to implement constraints, a valid default value is assigned to the attribute when an object is instantiated from the class. Whenever the value is changed at runtime, it is checked whether the value is valid or not. An invalid value may be handled by an exception handling routine or other methods.

Example

Consider an Employee class where age is an attribute that may have values in the range of 18 to 60. The following C++ code incorporates it −

```
class Employee {
   private: char * name;
   int age;
   // other attributes

   public:
   Employee() {                  // default constructor
      strcpy(name, "");
      age = 18;                 // default value
   }

   class AgeError {};           // Exception class
   void changeAge( int a) {    // method that changes age
      if ( a < 18 || a > 60 )  // check for invalid condition
      throw AgeError();         // throw exception
      age = a;
   }
};
```

Java Virtual Machine

- The JVM is the specification for a software program that executes code and provides the runtime environment for that code.
- The JVM is how we run our Java programs. We configure the JVM's settings and then rely on it to manage program resources during execution.
- *“Write once, run anywhere”*

Contd.

- Multiple different environment in a single system
- The Java Virtual Machine, or JVM, is an abstract computer that runs compiled Java programs.
- The JVM is "virtual" because it is generally implemented in software on top of a "real" hardware platform and operating system.
- The JVM plays a central role in making Java portable.
- It provides a layer of abstraction between the compiled Java program and the underlying hardware platform and operating system.

What makes the JVM lean and mean?

- The JVM is lean because it is small when implemented in software so that it can fit in as many places as possible -- places like TV sets, cell phones, and personal computers.
- The JVM is mean because it of its ambition. "Ubiquity!" is its battle cry.

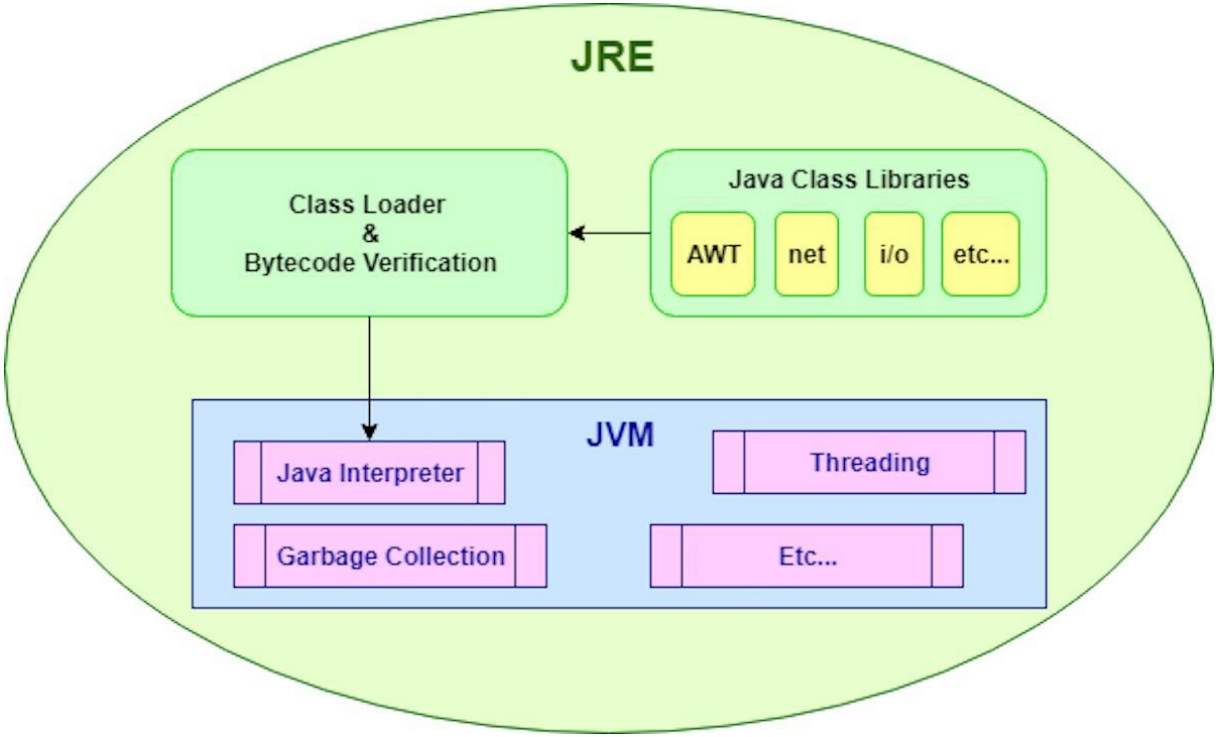
What the JVM is used for?

- There are two primary functions,
  - to allow Java programs to run on any device or operating system
  - to manage and optimize program memory

Java Runtime Environment (JRE)

- A *runtime environment* is a piece of software that is designed to run other software.
- As the runtime environment for Java, the JRE contains the Java class libraries, the Java class loader, and the Java Virtual Machine.
- The *class loader* is responsible for correctly loading classes and connecting them with the core Java class libraries.
- The *JVM* is responsible for ensuring Java applications have the resources they need to run and perform well in your device or cloud environment.
- The *JRE* is mainly a container for those other components, and is responsible for orchestrating their activities.

Contd.



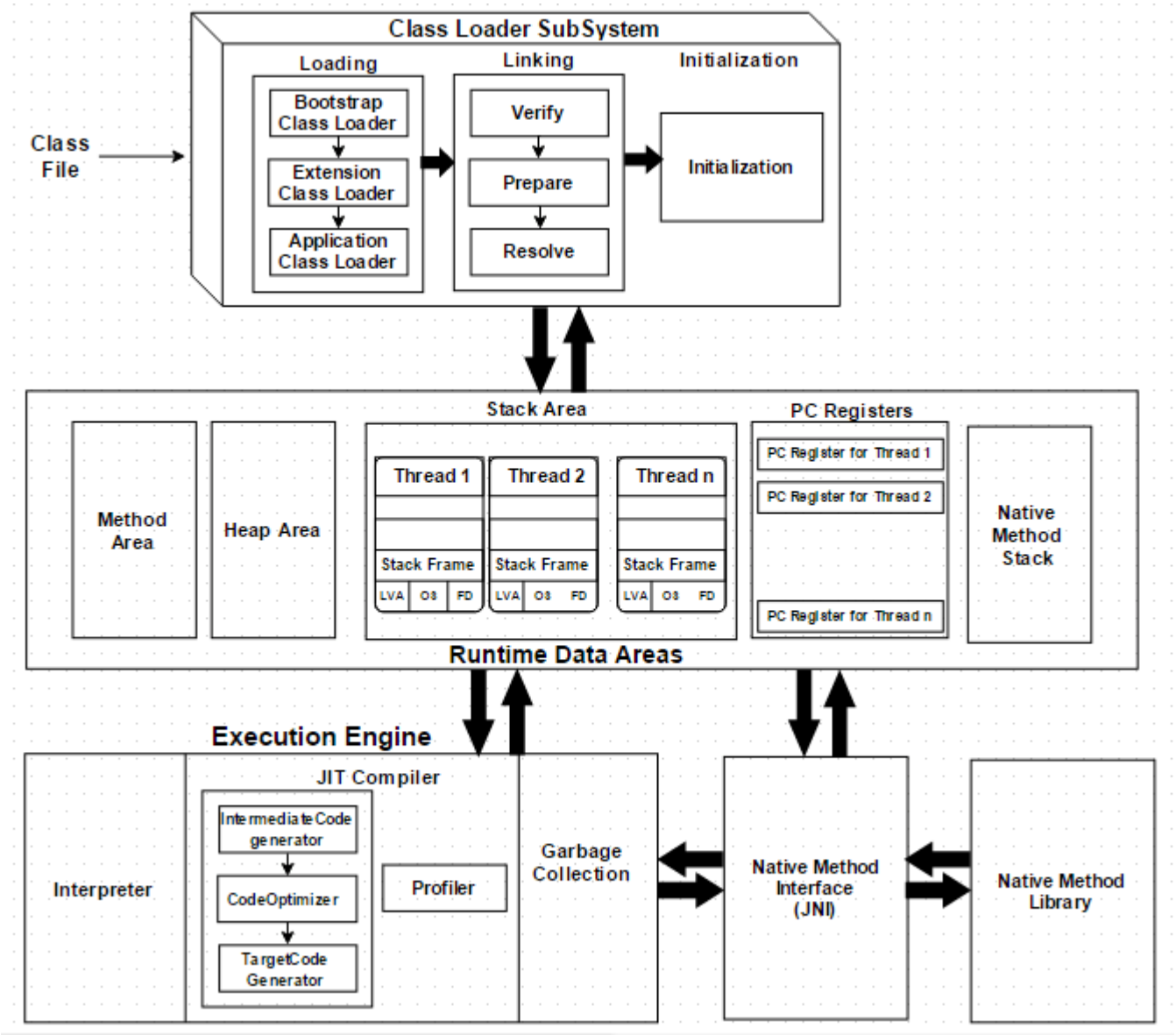
A layered architectural view shows that the JRE contains the JVM, class loader, and Java class libraries

Contd.

- The JRE smoothens over the diversity of operating systems.
- Automatic memory management is one of the JRE's most important services, ensuring that programmers don't have to manually control the allocation and reallocation of memory.
- In short, the JRE is a sort of meta-OS for Java programs. It's a classic example of *abstraction*, abstracting the underlying operating system into a consistent platform for running Java applications.
- The JRE is the on-disk system that takes your Java code, combines it with the necessary libraries, and starts the JVM to execute it.

Contd.

- The JRE contains libraries and software that your Java programs need to run. As an example, the Java class loader is part of the Java Runtime Environment. This important piece of software loads compiled Java code into memory and connects the code to the appropriate Java class libraries.
- Java memory consists of three components: the heap, stack and metaspace (which was previously called permgen).
- **Metaspace** is where Java keeps your program's unchanging info like class definitions.
- **Heap space** is where Java keeps variable content.
- **Stack space** is where Java stores function execution and variable references.



JVM Architecture

Memory management in the JVM

- The JVM manages memory through a process called garbage collection, which continuously identifies and eliminates unused memory in Java programs.
- Garbage collection happens inside a running JVM.

Garbage Collection

- Garbage collection (GC) is the process that aims to free up occupied memory that is no longer referenced by any reachable Java object, and is an essential part of the Java virtual machine's (JVM's) dynamic memory management system.
- The space occupied by previously referenced objects is freed and reclaimed to enable new object allocation.
- Quickly free unreferenced memory in order to satisfy an application's allocation rate so that it doesn't run out of memory.
- Reclaim memory while minimally impacting the performance (e.g., latency and throughput) of a running application.

Two kinds of garbage collection

1. Reference counting
2. Tracing collectors

Reference counting

- Reference counting collectors keep track of how many references are pointing to each Java object.
- Once the count for an object becomes zero, the memory can be immediately reclaimed.
- This immediate access to reclaimed memory is the major advantage of the reference-counting approach to garbage collection.
- Keeping all reference counts up to date can be quite costly

Challenges

- Keeping the reference counts accurate.
- The complexity associated with handling circular structures.

Tracing collectors

- All live objects are found by iteratively tracing all references and subsequent references from an initial set of known to be live objects.
- The initial set of live objects (also called *root objects*) are located by analysing the registers, global fields, and stack frames at the moment when a garbage collection is triggered.
- After that, the tracing collector follows references from these objects and queues them up.

Contd.

- Marking all found referenced objects live means that the known live set increases over time.
- .Once the tracing collector has found all live objects, it will reclaim the remaining memory.
- Tracing collectors differ from reference-counting collectors in that they can handle circular structures.
- The main attraction of most tracing collectors is the marking phase, which entails a wait before being able to reclaim non-referenced memory.

Tracing collector algorithms

- Tracing collectors are most commonly used for memory management in dynamic languages.
- They're still the two most common algorithms that implement tracing garbage collection
  - Copying
  - mark-and-sweep garbage collection

Copying collectors

- Traditional copying collectors use a **from-space** and a **to-space** i.e. two separately defined address spaces of the heap.
- At the point of garbage collection, the live objects within the area defined as **from-space** are copied into the next available space within the area defined as **to-space**. When all the live objects within the **from-space** are moved out, the entire **from-space** can be reclaimed.

Pros & Cons

- Main advantage of copying collectors is that objects are allocated together tightly in the **to-space**, completely eliminating fragmentation.
- Copying collectors are also known as *stop-the-world collectors*, which means that no application work can be executed for as long as the garbage collection is in cycle.
- It’s a time consuming algorithm and also not suitable where a large set of live objects are concerned because we have to make enough space to keep all the live objects. So, Its slightly memory inefficient.

Mark-and-sweep collectors

- A *mark-and-sweep collector* traces references and marks each found object with a "live" bit.
- Once the marking process is completed, in the sweep phase, a collector will traverse the entire heap to locate all unmarked locations of consecutive memory address spaces.
- Unmarked memory is free and reclaimable.
- The collector then links together these unmarked addresses into organized free lists.
- After, the sweep phase is complete allocation will begin again.
- Most commercial JVMs deployed in enterprise production environments run mark-and-sweep (or marking) collectors

Pros and Cons

- It optimizes the memory allocation and reduces fragmentation.
- The mark phase is dependent on the amount of live data on the heap and the sweep phase is dependent on the heap size. Since we have to wait until both the *mark* and *sweep* phases are complete to reclaim memory.
- this algorithm causes pause-time challenges for larger heaps and larger live data sets.
- Through GC-Tuning options we can accommodate various application scenarios and needs.
- Tuning for every load change and application modification is a repetitive task, however, as the tuning is only valid for a specific workload and allocation rate.

Implementations of mark-and-sweep

- Parallel approach
- Concurrent (or mostly concurrent) approach.

Parallel collectors

- *Parallel collection* means that resources assigned to the process are used in parallel for the purpose of garbage collection.
- Most commercially implemented parallel collectors are monolithic stop-the-world collectors -- all application threads are stopped until the entire garbage collection cycle is complete.
- In production environments, application threads cannot do any work during a GC
- It's having a major impact on response-time sensitive applications, especially if there are lot of references to trace.
- It will happen when many live or complex data structures on the heap.
- For a monolithic parallel approach using all resources in parallel, this entire time will be a pause, and that pause corresponds to the entire GC cycle.

Concurrent Collectors

- *Concurrent* means that some (or most) garbage collection work is performed concurrently with the running application threads.
- It is needed enough time to trace the live set and reclaim the memory before the application runs out of memory.
- Heuristics have been designed to determine when to start garbage collection and when to do various GC optimizing tasks and how much at a time, etc.

Why tuning doesn't replace garbage collection

- Most tuning parameters -- such as allocation rate, object sizes, timing of response-time sensitive tasks, and how fast objects die -- are tuned specifically for the application's allocation rate, such as the test workload at hand. The end result could be either (or both) of these:
  - Things that worked during testing fail in production.
  - Workload or application changes requires to re-tune the application entirely.
- Tuning is something that will always need to be repeated!
- Concurrent garbage collectors in particular can require a lot of tuning, especially in production environments. Heuristics are needed to match the specific application's needs for the expected worst-case load.
- The end result becomes a very rigid configuration, leading to a lot of resource waste.

Summary

- Different garbage collection algorithms and approaches will meet different application needs. Tracing collectors are most commonly used in commercial Java environments.
- Parallel garbage collection uses available resources in parallel to perform GC. This tactic is usually implemented as a monolithic, stop-the-world collector, using all available system resources for a fast GC. Parallel GC thus provides higher throughput, but all application threads must wait until it's finished, which impacts latency.
- Concurrent GC does its work while application threads are still running. The timing of concurrent GC is tricky because it needs to be finished before your application requires memory.

Contd.

- Generational garbage collection helps postpone fragmentation, but does not eliminate it. Generational GC divides the heap into two spaces, one for allocating young objects and one for objects that (being still referenced) have survived young-space GC. Use a generational collector for any Java application that has many short-lived small objects that will die within their first collection cycle.
- Compaction is the only way to handle fragmentation completely. Most collectors have to perform compaction as a stop-the-world operation. Applications having long running time with more reference complexity having more heterogeneous object-size distribution. These factors will result in longer pauses to complete compaction. Larger heap size also impacts the compaction pause because there will likely be more live data and more references to update.

- Tuning can help postpone OutOfMemoryErrors but the trade-off of too much tuning is rigidity. Be sure about the consequences of a non-dynamic approach before setting it as, A too-rigid configuration will most likely break under dynamic production loads.