

Object Oriented Programming

Introduction to Java

Module Content

- Introduction to Java
- Object Oriented Concepts – Recap
- Collections and Generics
- Thread Implementation
- Design Patterns

Assessments

- Continuous Assessments – 50%
 - Mid Term Exam (MCQ) – 20% (Week 8)
 - Online Exam – 10% (Week 11)
 - Group Project Submission – 20% (Last Week)
- Final Examination (Online) – 50%

Lecturers

- Ms.Thilmi Anuththara Kuruppu – Lecturer in Charge (Malabe)

thilmi.k@sliit.lk

- Mr.Udara Samaratunga (Malabe)
- Ms.Janani Tharmaseelan (Malabe)
- Dr.Kalpani Manathunge(Metro)
- Ms.Shanika Abeyrathne(Kandy)

OOP – Learning and Support

- Lectures – 2 Hours
- Tutorial – 1 Hour (Sometimes embedded with Lectures)
- Lab – 2 Hours
- Homework – Typically $\frac{1}{2}$ hour to 1 hour of work
- Quizzes – 5 minute Quizzes in each Lab similar to how OOC was conducted. Will be based on the Homework.
- Self Help Labs/Tutorials – Links to specific Help Labs/Tutorials will be provided so that you can cover material that you lack on your own.
- Help Desk – We will run a Help Desk from the 2nd Week onwards, you can get an appointment to get Help for OOP
- Video Lectures – Some of the Lectures will be recorded and made available in the course web.

How to get a good Grade

- Do the obvious things
 - Attend Lectures, Tutorials, Labs
 - Do your Assignments, Tutorials by yourself
 - Don't wait till the last minute to realize that you don't understand something. First try things on your own, go through the provided material, if you still can't get help from the OOP Team.
- The Final Exam is an Online Exam
 - Install Eclipse Oxygen (Latest version) in your Home Computer.
 - Try out programs on your own.
 - Genuinely attempt the Homework that is given to you each week. This includes Tutorials given.
- Explore things on your own. Go through and try out material in the Internet about Java Programming.
- Work towards developing an impressive Group Project.

Group Project

- You need to be groups of 4 Members
- Same Group Members will be there for the Software Engineering Module.
- The Case Study will be given to you in the Software Engineering Module in Week 2.
- You will develop a Java Web Based Application. A recorded lecture introducing these topics will be made available from Week 3.
- Submission and Presentation are due in the last week of the Semester.

Asking Questions

- Talk to your Lecturer/Instructor Directly.
- Use the Courseweb Moodle Forum to ask Questions.
- Get an appointment for a Help Desk (Details from 2nd Week onwards)

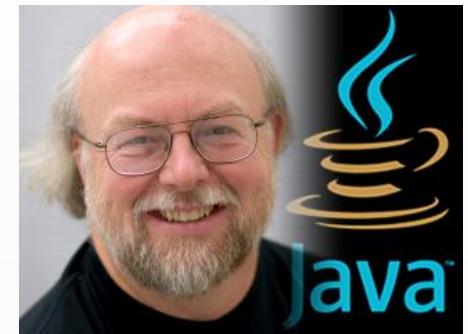
Learning Outcomes

At the end of the Lecture students should be able to

- List the differences between C++ and Java
- List how a Java program is compiled and executed in a Computer.
- List Features that make Java unique
- Write, Compile and Execute simple Java programs
- Write a Java program including
 - Input / output commands
 - Variable
 - Sequence
 - Selection
 - Repetition

Java

- 1991 - James Gosling, Sun Microsystems, Inc.
- Originally a platform independent language for programming home appliances and was called “Oak” later renamed “Java” in 1995.
- Later (1994) used for World Wide Web applications (since byte code can be downloaded and run without compiling it)
- Eventually used as a general-purpose programming language (for the same reason as above plus it is object-oriented)
- Why the name “Java”? Java was then named “Java”, paying homage to the large amounts of coffee consumed by the team.
- Now owns by Oracle





- Full-fledged application programming language
- Additional capability as a Web programming language (currently the strength of its application base)
- A pure OO programming language
- NOT radical or especially new
- Adopts its looks from C++, and its behavior from Smalltalk
- Compiled to processor-neutral instruction set then interpreted on each supporting platform
- Extremely fast adoption rate! (due to WWW)



Buzzwords

Simple	Java has a concise, cohesive set of features that makes it easy to learn and use.
Secure	Java provides a secure means of creating Internet applications.
Portable	Java programs can execute in any environment for which there is a Java run-time system.
Object-oriented	Java embodies the modern, object-oriented programming philosophy.
Robust	Java encourages error-free programming by being strictly typed and performing run-time checks.
Multithreaded	Java provides integrated support for multithreaded programming.
Architecture-neutral	Java is not tied to a specific machine or operating system architecture.
Interpreted	Java supports cross-platform code through the use of Java bytecode.
High performance	The Java bytecode is highly optimized for speed of execution.
Distributed	Java was designed with the distributed environment of the Internet in mind.
Dynamic	Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.



Terminology

- Class - A collection of data and methods that operate on that data.
- Method - A group of statements in a class that handle a task.
- Attribute - A property of an instance of a class.
- Interface - A skeleton class.
- Package - A group of logically related codes (classes & interfaces).



Terminology

- Bytecodes
 - A set of instructions that look like machine code, but are not specific to any processor.
- Virtual Machine
 - The environment in which Java runs. The JVM is responsible for executing the bytecodes and has responsibility for the fundamental capabilities of Java.

C++ vs Java

```
// C++ Program
#include <iostream>

int main ( )
{
    std::cout << "Hello World !" <<
                  std::endl;

    return 0;
}
```

Helloworld.cpp

Output :

```
// Java Program

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

Helloworld.java

Hello World !

First Java Program

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

Comments

```
// Java Program : prg_01.java  
// Printing a String
```

- Comments provide information to the people who read the program
- Comments are removed by the preprocessor, therefore the compiler ignores them
- In Java, there are two types of comments
 - Single line comments //
 - Delimited comments /* */ for comments with more than one line.

Everything is Object Oriented

```
public class HelloWorld
```

- In Java is fully object oriented, even the simplest program needs to be written using a class.
- HelloWorld is the name of the class.
- A public class needs to be stored in a file name that matches the class name. i.e. The above code needs to be saved in a filename called HelloWorld.java

The main method

```
public static void main(String[] args)
```

- Java programs begin executing at method `main`.
The `main` method must be given as above.
- The `main` method is one of the methods in a class
- `void` methods do not return a value.

Output Statement

```
System.out.println("Hello World !");
```

- Instructs the computer to display the data within brackets to the screen.
- “Hello World !” :String / String Literal. What you need to display on screen
- System.out.println() moves the cursor to the next line after displaying the data

Exercise - 1

- Write a Java program to display your name and address in 3 lines.

Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Java Development Environment

1. Install the Java JDK. The latest Java SE JDK is 9. We will use Java SE JDK 8 in this course.
2. Editing a program
 - Type a Java program (source code)
 - e.g :
 - vi editor (Linux)
 - notepad (Windows DOS prompt)
 - IDE (Eclipse, IntelliJ)
 - Extension : .java

Java Development Environment cont...

3. Compiling a Java program

- The java compiler (javac) compiles java code to an intermediate language called Java Byte Code. This is a platform neutral low level language which can be translated to machine code. Compiled java programs are stored in files with the extension .class

4. Running a Java Program

- The java interpreter is used to run a compiled program on your computer. Each platform has its own Java Virtual Machine which translates Java Bytecode to machine code.

5.

Integrated Development Environments (IDEs)

- Provides tools that support the software development process
- Includes, editors for writing & editing programs, debuggers for locating logical errors, design tools etc....
 - Eclipse
 - IntelliJ

Java Architecture

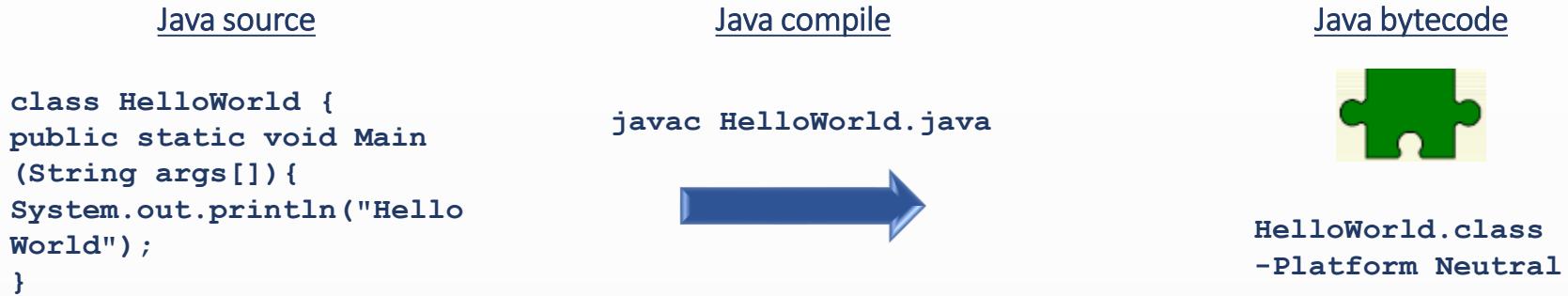
JAVA Application (Bytecodes)

JAVA Virtual Machine

Any Operating System

Any Hardware Platform

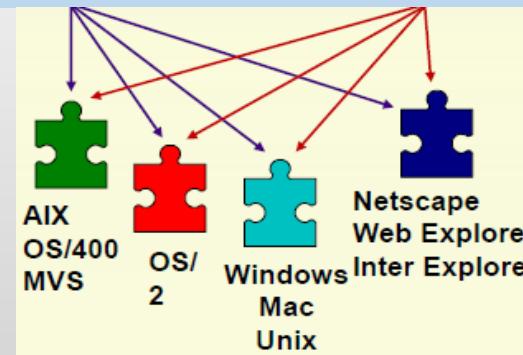
Platform Independence



- Write application once, runs on any
- Written in software - defined by Oracle
Assured through compatibility test suite

Virtual Machine - spec of microprocessor

Interpret Translate



Vendors Port VM to:

- *Operating Systems*
- *Enable in Browsers*

Java source compiled into intermediary bytecode

- Application runs anywhere Java VM is ported
- Applet runs in any Java enabled browser

Java's magic Byte Code

- Generated by Java compiler
 - Instead of generating machine language as most compilers do, the Java compiler generates byte code.
- Easily translated to machine language of various kinds of computers
- Executed by Java interpreter
- Invisible to programmer
 - You don't have to know anything about how byte code works to write a Java program.

Language Translation

A ***source program*** is the one that you write in the Java language and that always has a file extension of ***.java***.

An ***object program*** is the binary byte-code program generated by the Java compiler, which always has a file extension of ***.class***.

The ***.class*** file generated by the Java compiler contains ***bytecode*** which is a low-level code similar to machine language, but generic and not specific to any particular CPU.

Java Virtual Machine

A given computer must have its own Java interpreter as part of a Java Virtual Machine, or JVM, to translate the generic bytecode into machine language for that CPU.

Virtual Machine (VM) interprets bytecodes into native machine language and runs it. Different VM exists for different computers, since bytecode does not correspond to a real machine.

Why Use Byte Code?

Disadvantages:

- Requires both compiler and interpreter
- Slower program execution

Advantages:

- Portability
 - Very important
 - Same program can run on computers of different types (useful with the Internet)
- Small in size
 - Linking at runtime
 - Easy to exchange over a network

Java Program Patterns

- Stand Alone Applications
 - Java programs which run in command console
 - Applications with GUI (awt/swing)
- Web-based Java Applications
 - Servlets
 - Programs that run inside request/response oriented servers
 - JavaServer Pages
 - An extension to the servlet architecture. Allow for the separation of the display of web content and the generation of that content

J.... confusion

- JVM (Java Virtual Machine)
 - JVM is a part of both the JDK and JRE that translates Java byte codes and executes them as native code on the client machine.
- JRE (Java Runtime Environment)
 - It is the environment provided for the java programs to get executed. It contains a JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc.
- JDK (Java Development Kit)
 - JDK contains tools needed to develop the Java programs (javac, java, javadoc, appletviewer,jdb, javap, rmic....),and a JRE to run the programs.
- Java SDK (Java software development kit)
 - SDK comprises a JDK and extra software, such as application servers, debuggers, and documentation.
- Java SE
 - Java Platform, Standard Edition (Java SE) lets you develop and deploy Java applications on desktops and servers (Same as SDK)
- J2SE, J2ME, J2EE
 - Any Java edition from 1.2 to 1.5

Exercise

What is the meaning of “write once, run anywhere”? Select the correct options:

1. Java code can be written by one team member and executed by other team members.
2. It is for marketing purposes only.
3. It enables Java programs to be compiled once and can be executed by any JVM without recompilation.
4. Old Java code doesn’t need recompilation when newer versions of JVMs are released.

Printing Values

- System.out.println() and System.out.print()

```
public class Print1 {  
  
    public static void main(String args[]) {  
        System.out.print("This is line 1");  
        System.out.print(" still line 1");  
        System.out.println(" lets move to the next line ");  
        System.out.println("finally line 2");  
    }  
}
```

Print1.java

Printing Values

- System.out.println() and System.out.print()

```
int no = 50;
long population = 70000000;
double salary = 4500.34;
float rate = 34.5f;

System.out.println("no = " + no);
System.out.println("population = " + population);
System.out.println("salary = " + salary);
System.out.println("rate = " + rate);
```

Print2.java

Printing Values

- System.out.println() and System.out.print()

```
int no = 50;
long population = 70000000;
double salary = 4500.34;
float rate = 34.5f;
```

```
System.out.println("no = " + no + "\n"
+ "population = " + population + "\n"
+ "salary = " + salary + "\n"
+ "rate = " + rate);
```

Print3.java

Inputting Values from the Keyboard

- Using `java.util.Scanner`

```
3 import java.util.Scanner;  
4  
5 public class Input {  
6     public static void main(String args[]) {  
7         String name;  
8         int age;  
9         float salary;  
10        Scanner myScanner = new Scanner(System.in);  
11        System.out.print("Enter your name : ");  
12        name = myScanner.next();  
13        System.out.print("Enter your age : ");  
14        age = myScanner.nextInt();  
15        System.out.print("Enter your salary : ");  
16        salary = myScanner.nextFloat();  
17  
18        System.out.println("Name = " + name);
```

Input.java

Java vs C++ Language

- Java control structures are identical in syntax
 - selection – if, switch,
 - Repetition – while, do while, for
- Within a method the major difference is the print commands and the input commands.
- The basic data types integers and float are used in the same way. There is a separate data type in Java for string data called String.
- Calculations are also identical.
- There is a slight difference on how arrays are declared (i.e. similar to C++ dynamic arrays)

Use of Variables

- Same as in C++ (See Variables.cpp)

```
public class Variables {  
    public static void main(String args[]) {  
        int no = 50;  
        long population = 70000000;  
        double salary = 4500.34;  
        float rate = 34.5f;  
  
        System.out.println(no);  
        System.out.println(population);  
        System.out.println(salary);  
        System.out.println(rate);  
    }  
}
```

Variables.java

Calculations

- Same as in C++ (Calculations.cpp)

```
int no = 50;
long population = 70000000;
double salary = 4500.34;
float radius = 30.0f;

int remainder = no % 3;
double contribution = population * 100;
double area = 22.0/7*radius*radius;
```

Calculations.java

Selection - If

- Same as in C++

```
System.out.println("5. Kurunagala");
System.out.println("6. Jaffna");
System.out.println("0. Exit");

opt = 5;
System.out.print("Option : ");
System.out.println(opt);

if (opt == 1)
    System.out.println("Malabe Campus");
else if (opt == 2)
    System.out.println("Metro Campus");
else if (opt == 3)
    System.out.println("Matara Centre");
else if (opt == 4)
    System.out.println("Kandy Centre").
```

If.java

Selection - Switch

- Same as in C++

```
System.out.println("5. Kurunagala");
System.out.println("6. Jaffna");
System.out.println("0. Exit");

opt = -5;
System.out.print("Option : ");
System.out.println(opt);
switch (opt) {
    case 1 : System.out.println("Malabe Campus");
    |           | break;
    case 2 : System.out.println("Metro Campus");
    |           | break;
    case 3 : System.out.println("Matara Centre");
    |           | break;
```

Switch.java

Repetition - while

- Same as in C++

```
int r = 1;
while (r < 100) {
    System.out.println(r);
    r++;
}
System.out.println();
r = 50;
while (r > 0) {
    System.out.print(r + " ");
    r -= 5;
}
```

While.java

Repetition - for

- Same as in C++

```
for (int r = 1; r<100; r++) {  
    System.out.println(r);  
}  
  
System.out.println();  
for (int r = 50; r > 0; r-=5) {  
    System.out.print(r + " ");  
}
```

For.java

Repetition – do while

- Same as in C++

```
int r = 1;
do {
    System.out.println(r);
    r++;
} while (r < 100);
System.out.println();
r = 50;
do {
    System.out.print(r + " ");
    r -= 5;
} while (r > 0);
```

DoWhile.java

Exercise - 2

- Write a java program to input the length and the width of a rectangle and calculate and print the perimeter.

Exercise - 3

- Write a program to input 3 integers and print the largest ad the smallest of the 3 numbers entered.

Exercise - 4

- Write a program to input 10 numbers from the keyboard and find how many odd numbers and how many even numbers were entered.

Object Oriented Programming

Java Classes and Objects

Learning Outcomes

At the end of the Lecture students should be able to get a revision on OOC you learnt in Year 1

- Object Oriented Programming
 - Classes and Objects
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism
 - Interfaces
-
- We will also look at key differences between writing Simple Object Oriented Programs in C++ and Java.

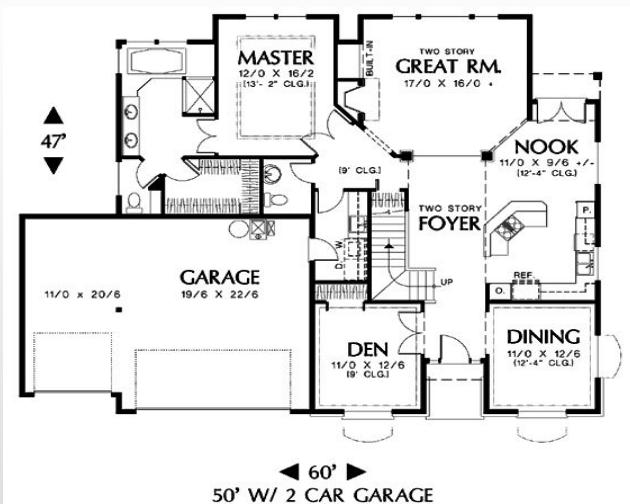
Object Oriented Programming

- Object Oriented Programming is a method of implementation in which programs are organized as a collection of objects which cooperate to solve a problem.
- Allows to solve more complex problems easily

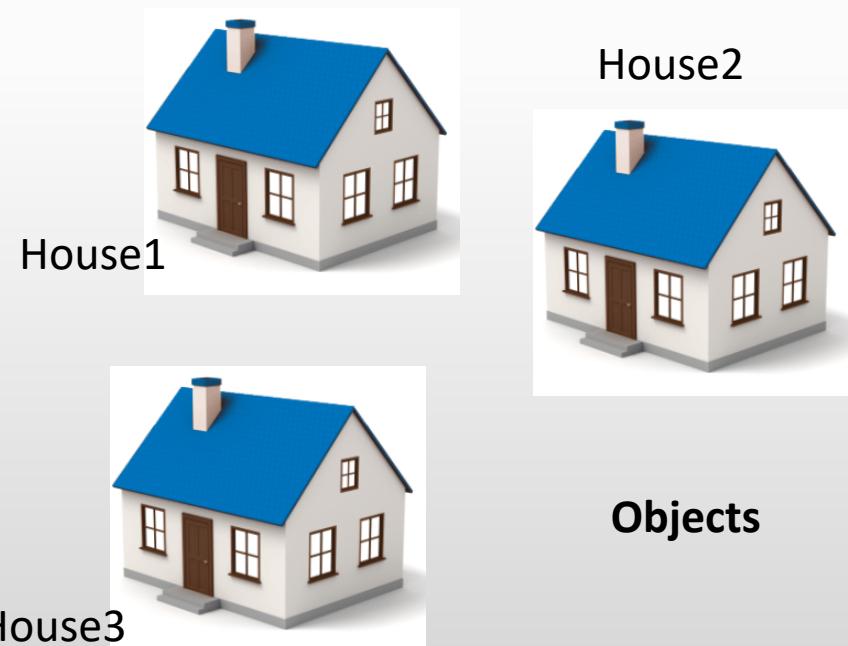


Classes and Objects

- An Object is a specific instance of the data type (class)
- A class is a blue print of an object.



Class House



Objects

- Objects are instances of classes, which we can use to store data and perform actions
- We need to define a class including the properties and methods and then create as many objects which has the same structure of the class (House example)

Class in Java

```
class Student{  
    private :  
        int studentNo;  
        char name[30];  
        int CA_mark;  
        int Final_mark;  
    public:  
        void assignMarks(int pCA, int pFin) {  
        }  
        int calculateTotal() {  
        }  
        void printDetails() {  
        }  
};
```

C++

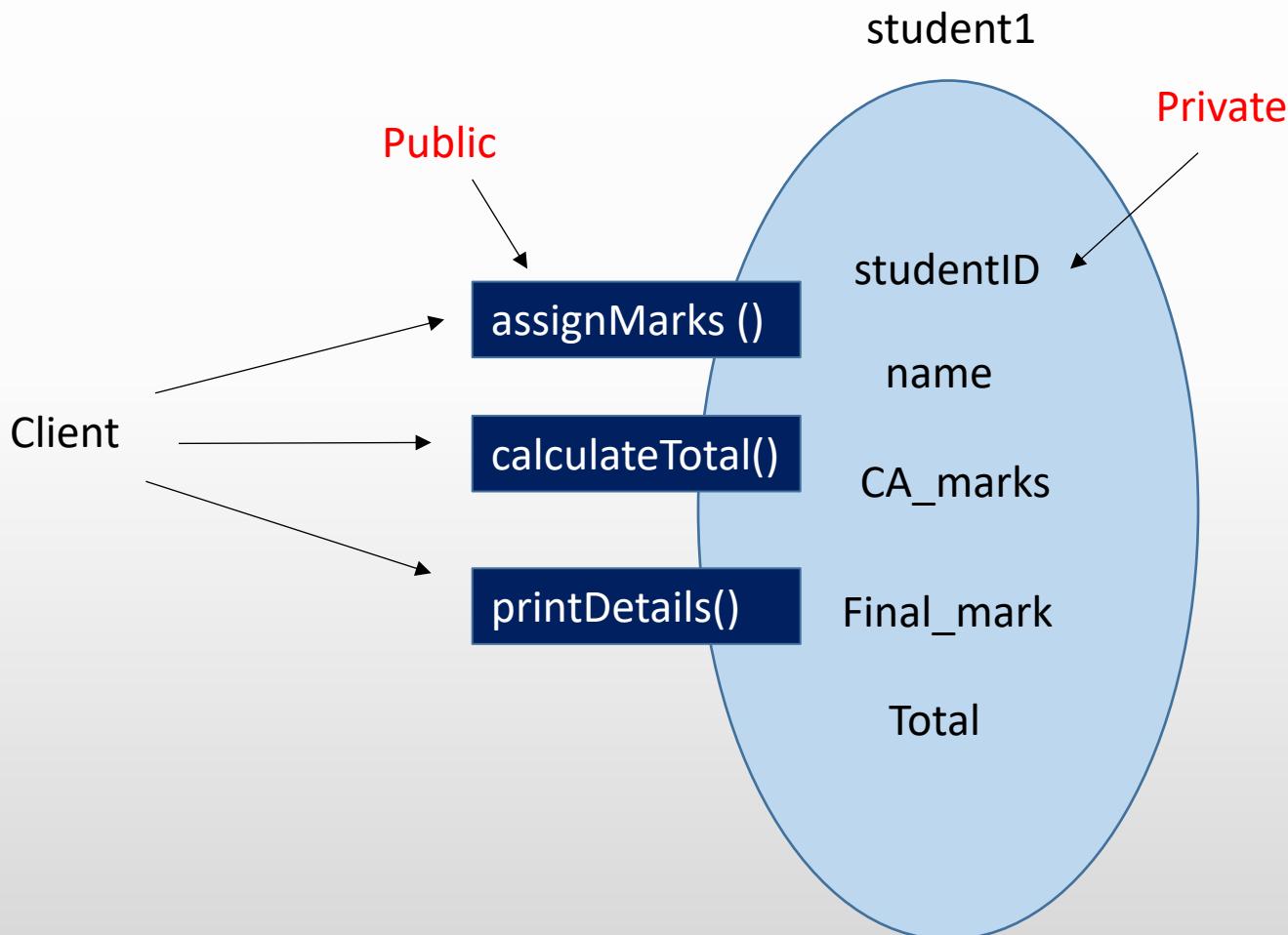
```
class Student {  
    private int studentNo;  
    private String name;  
    private int CA_mark;  
    private int Final_mark;  
  
    public void assignMarks(int pCA,  
                           int pFin) {  
    }  
    public int calculateTotal() {  
    }  
    public void printDetails() {  
    }  
}
```

Java

Java vs C++

- All methods are implemented in the class definition in Java.
- Each property, method needs a specific access modifier e.g. private, public, protected
- In Java there is no semi colon at the end of the class
- In Java you only have dynamic objects.
- Since Java has an automatic garbage collector, you do not need to use a command line delete to remove objects from memory.
- We use the dot operator instead of the -> operator to access methods in Java.

Private & Public



Creating Objects

```
Student student1 = new Student();  
Student student2 = new Student();  
// We do not use * for pointers in Java
```

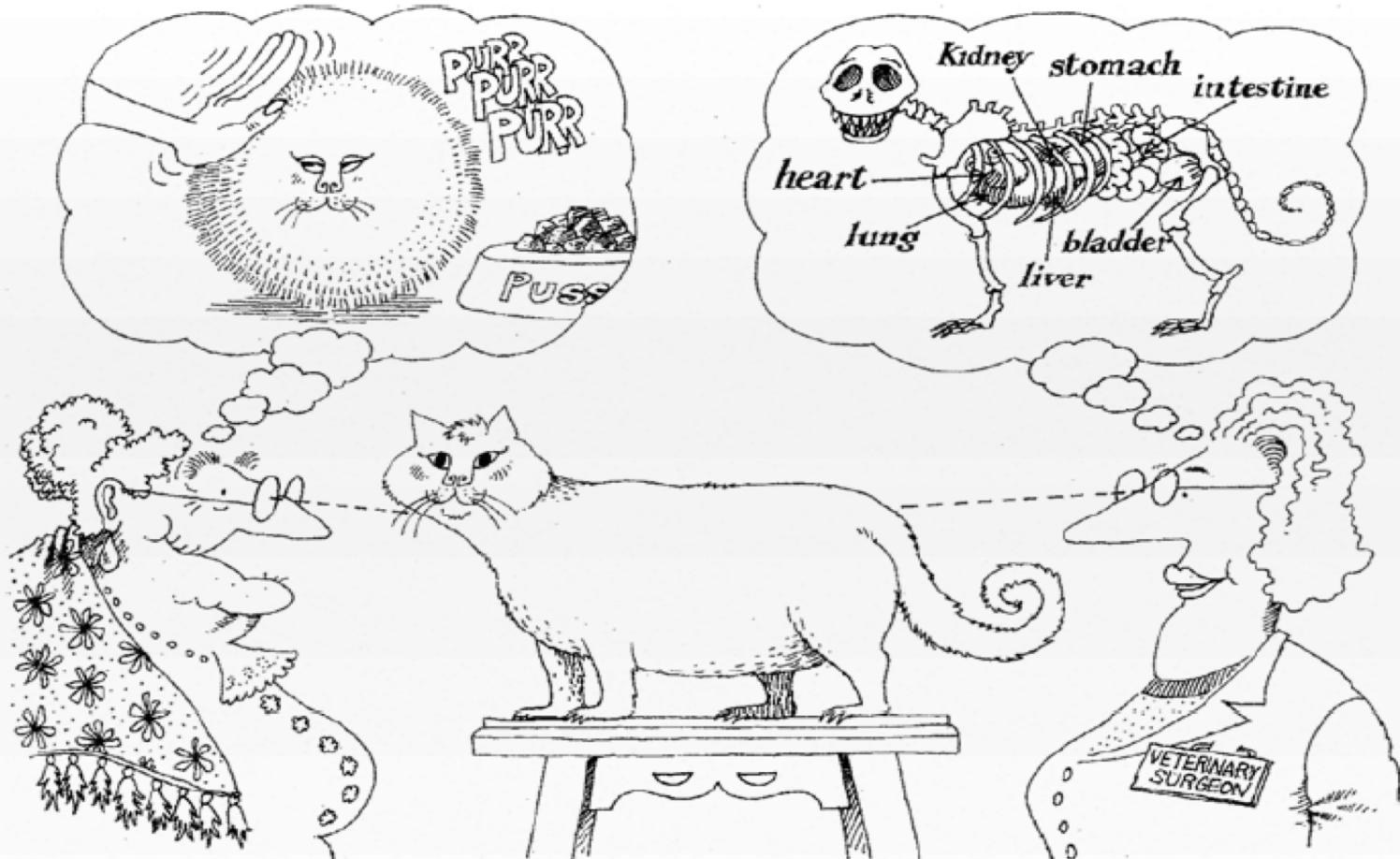
student1
studentNo – 1011
Name – Ajith Silva
CA_mark - 56
Final_mark -60

student2
studentNo – I131
Name – Surani Fernando
CA_mark - 70
Final_mark -65

Methods and Properties

- In C++ properties are called **data members**, other popular names for **properties** are **attributes**, **variables**.
- In C++ **methods** are called **member functions**, other popular names are **operations**, **behaviors**. These are really functions.

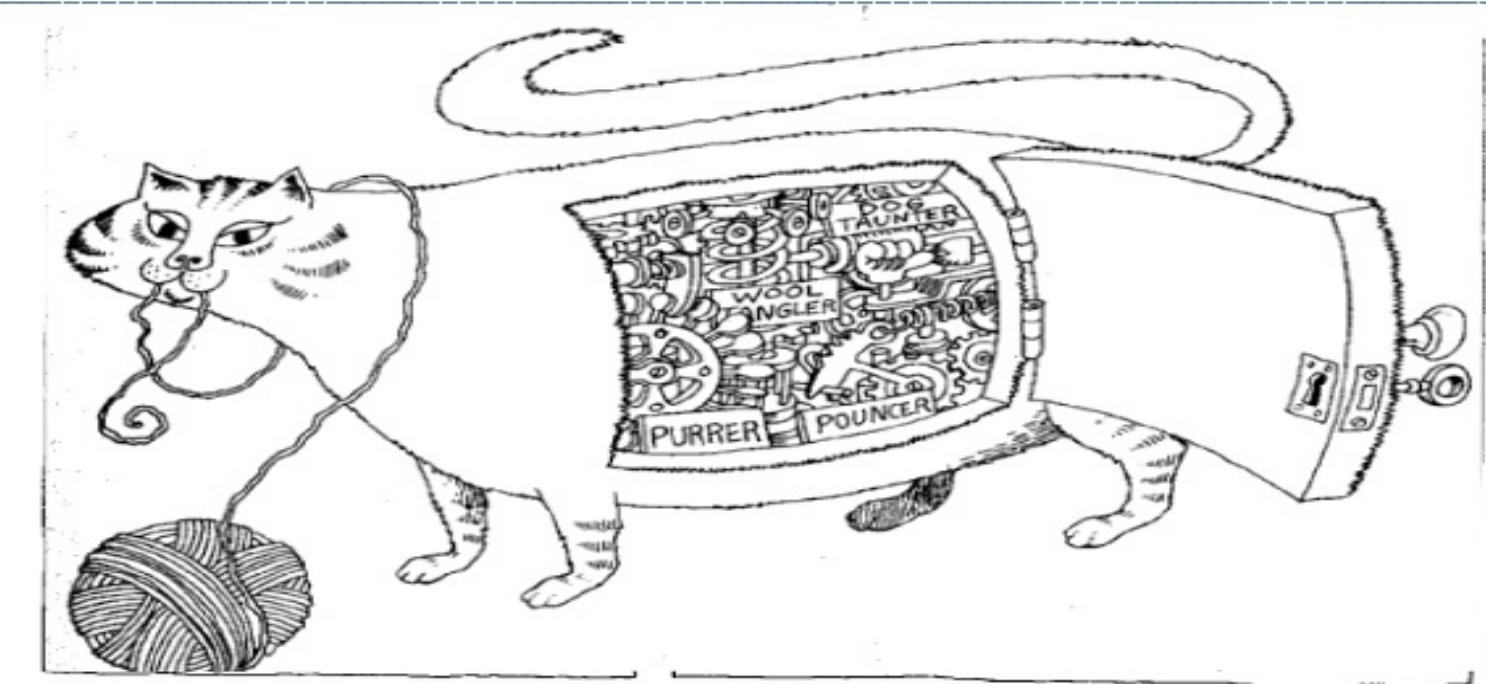
Abstraction



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

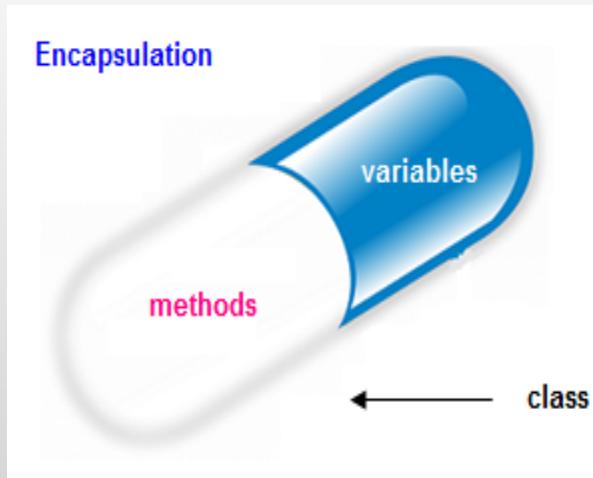
Encapsulation

Encapsulation hides the details of the implementation of an object



Encapsulation

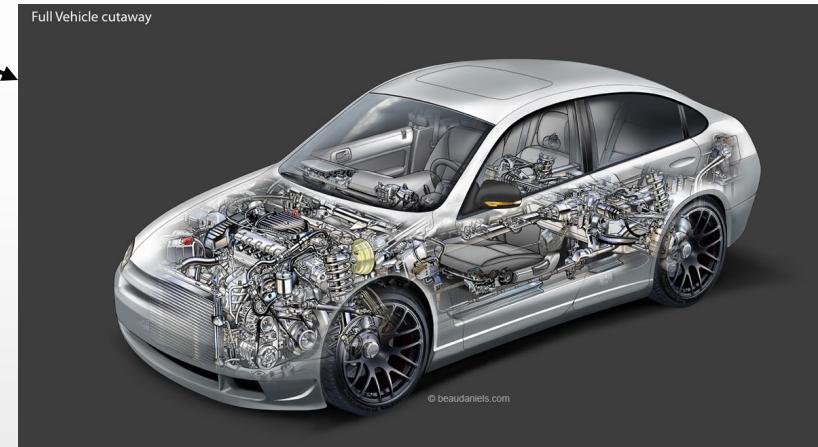
- It is the process of grouping related attributes and methods together, giving a name to the unit and providing an interface for outsiders to communicate with the unit.



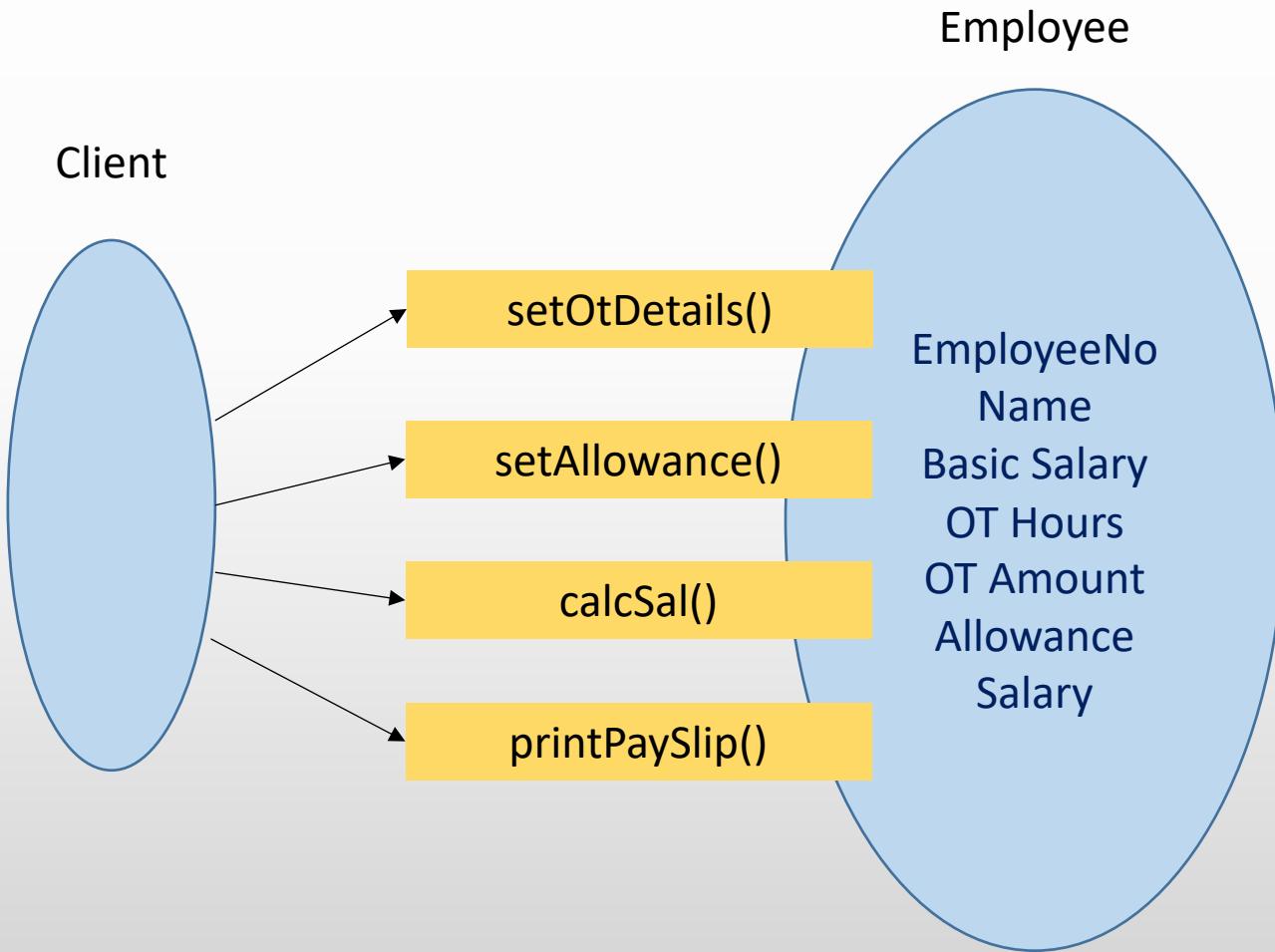
Information Hiding

- Hide certain information or implementation decision that are internal to the encapsulation structure (class)
- The only way to access an object is through its public interface
 - Public – anyone can access / see it
 - Private – no one except the class can see/ use it

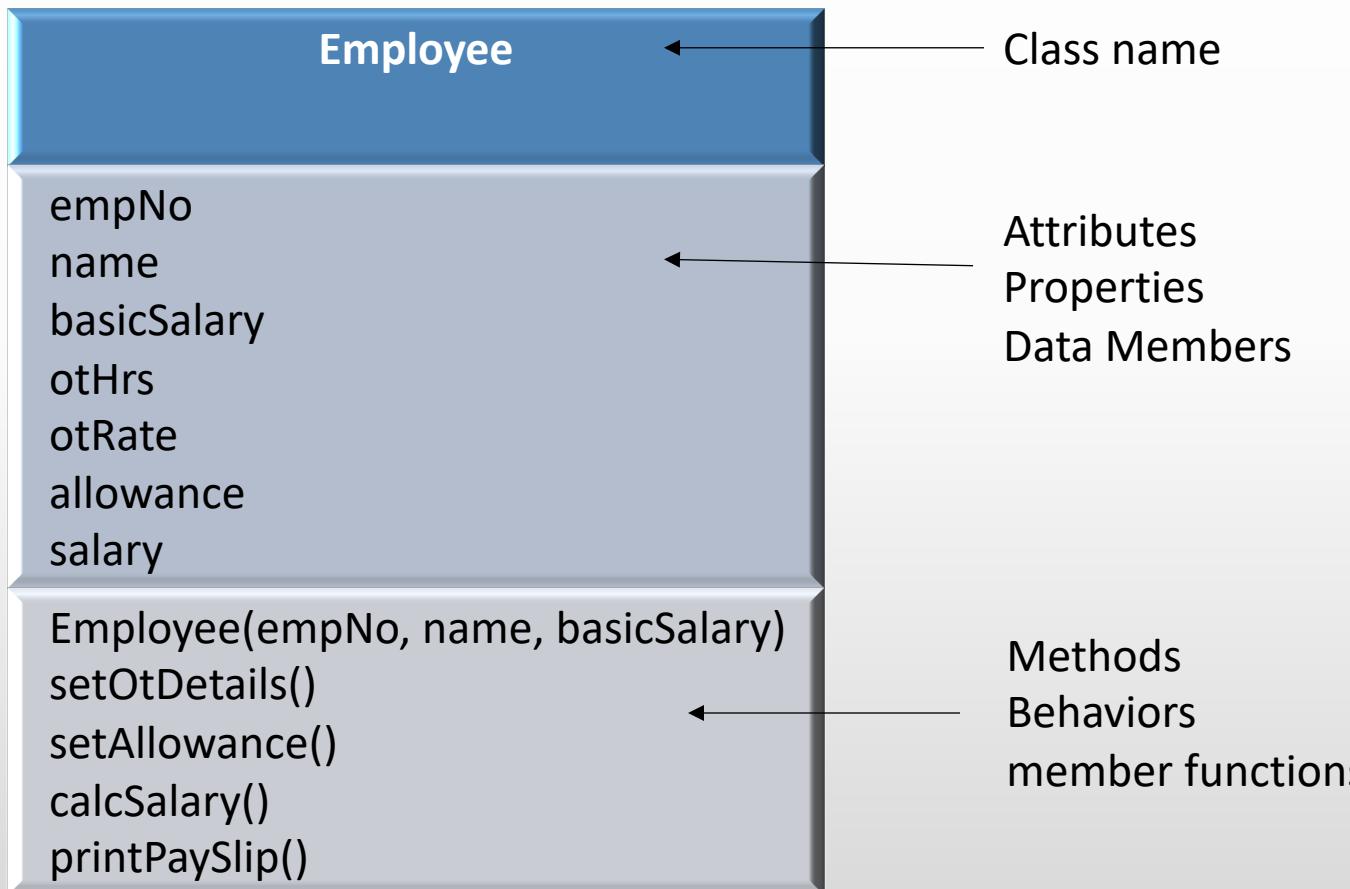
Interface



Interface



Terminology



Constructor

- The constructor is used to initialize the object when it is declared.
- The constructor does not return a value, and has no return type (not even void)
- The constructors have the same name as the class.
- There can be default constructors or constructors with parameters
- When an object is declared the appropriate constructor is executed.

Constructors

- Default Constructors
 - Can be used to initialized attributes to default values

```
public Rectangle () {  
    width = 0;  
    length = 0;  
}
```

- Overloaded Constructors (Constructors with Parameters)
 - Can be used to assign values sent by the main program as arguments

```
public Rectangle (int w, int l) {  
    width = w;  
    length = l;  
}
```

Getters and Setters

- In general properties are declared as private preventing them from being accessed from outside the class.
- Typically an attribute will have a getter (accessor) (A get method to return its value) and a setter (mutator) (A set method to set a value).
- e.g. a property called length will have a getter defined as `int getLength()` and a setter defined as `void setLength()`. Both these methods will be declared as public methods.

this keyword

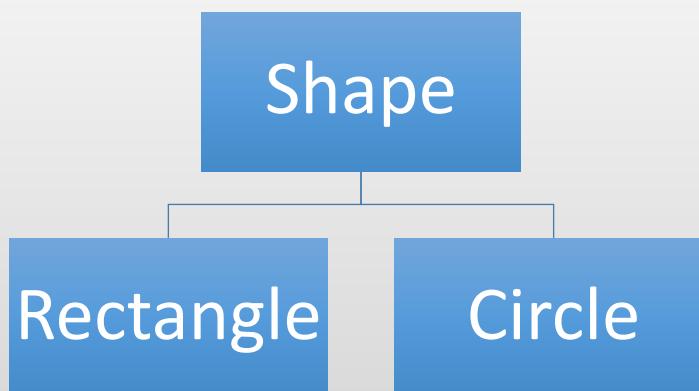
- The this **keyword** can be used to refer to any member of the current object from within an instance Method or a constructor.

```
public Employee(int pempno, String name, double pbasicSal) {  
    employeeNo = pempno;  
    this.name = name;  
    basicSalary = pbasicSal;  
}
```

- We need to use this.name to refer to the property name to distinguish it from the parameter name.

Generalization/Inheritance

- Child class **is a type** of the parent class
- Used to showcase reusable elements in the class diagram
- Child classes “inherit” the attributes and methods defined in the parent class



C++ vs Java - Inheritance

C++

```
class Circle : public Shape {
```

Java

```
class Circle extends Shape {
```

Java has a simpler inheritance mechanism where base class is extended as public.

C++ has multiple inheritance compared to Java's Single Inheritance.

C++ vs Java - Inheritance

C++

```
Circle (string tname, int r) : Shape ( tname ) {  
    radius = r;  
}
```

Java

```
public Circle (String tname, int r) {  
    super(tname);  
    radius = r;  
}
```

When you want to call a base class constructor C++ Requires to explicitly name the base class. In Java we use the super keyword to access the direct descendent class.

However this implies that in Java you can't directly call a class higher in the hierarchy e.g. the Grandfather class which is not in C++

C++ vs Java - Inheritance

C++

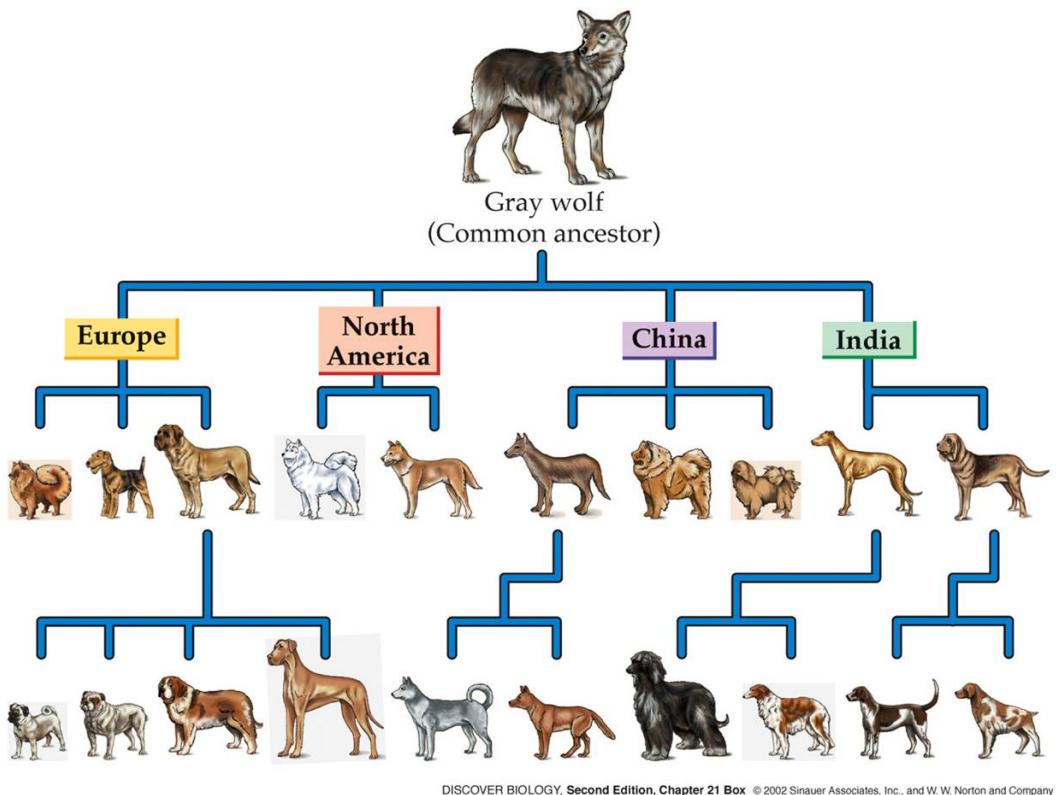
```
virtual void speak() {}
```

Java

```
public void speak() {}
```

All methods in Java are virtual by default. In C++ we need to explicitly define polymorphic methods.

Object Class



- A class hierarchy is similar to the taxonomy of animals that shows their ancestry.
- There are many breeds of dogs. A well known fact is that the common origin of a dog is the Gray Wolf.
- All Java classes are derived from a class called Object. This includes all the existing Java built in classes and the classes that you write. The methods and properties of the Object class is accessible to any Java class that you create.

Inheritance Shapes Example C++

```
23  class Rectangle: public Shape{  
24      protected:  
25          int width;  
26          int height;  
27      public :  
28          Rectangle (string tname, int w, int h) : Shape ( tname)  
29          {  
30              width = w;  
31              height = h;  
32          }  
33          int area( )
```

Shape_example.cpp

Inheritance Shapes Example Java

```
1  class Shape {  
2      protected String name;  
3      public Shape() {};  
4      public Shape (String tname) { ...  
5      }  
6      public void print() { ...  
7      }  
8      public int area(){ return 0;}  
9  }  
10 class Rectangle extends Shape { ...  
11 }  
12 class Circle extends Shape { ...  
13 }  
14 class ShapeApp {  
15     public static void main(String args[]) {  
16         Rectangle R = new Rectangle("Rectangle", 4 , 6);  
17         Circle C = new Circle("Circle", 3 );  
18     }  
19 }
```

Shape_example.java

Polymorphism

- Ability to assign a different meaning or usage to something in different contexts
- Consider the request (analogues to a method)
“please cut this in half” taking many forms



For a cake:

- Use a knife
- Apply gentle pressure

For a cloth:

- Use a pair of scissors
- Move fingers in a cutting motion

Animal Example

```
2  + class Animal { ...
20 }
21 + class Cat extends Animal { ...
30 }
31 + class Dog extends Animal { ...
40 }
41 + class Cow extends Animal { ...
50 }
51 - class AnimalApp {
52     public static void main(String args[]) {
53         Animal ani[] = new Animal[4];
54         ani[0] = new Cat("Micky the Cat");
55         ani[1] = new Dog("Rover the Dog");
56         ani[2] = new Cow("roo the Cow");
57         ani[3] = new Animal("no name");
58     - for (int r=0;r<4; r++)
59         ani[r].song();
```

Animal_example.cpp

Animal_example.java



SLIIT

Discover Your Future

IT2030

Object Oriented Programming

Lecture 02

Object Oriented Concepts Recap

Dr. Kalpani Manathunga

Learning Outcomes

At the end of the Lecture you should know

- Object Oriented Programming
- Classes and Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Interfaces

We will also look at key differences between writing simple object-oriented programs in C++ and Java.

Object Oriented Programming

- Object Oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.
- • •

(Reference : Grady Booch, et.al (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 41)

Object Oriented Programming

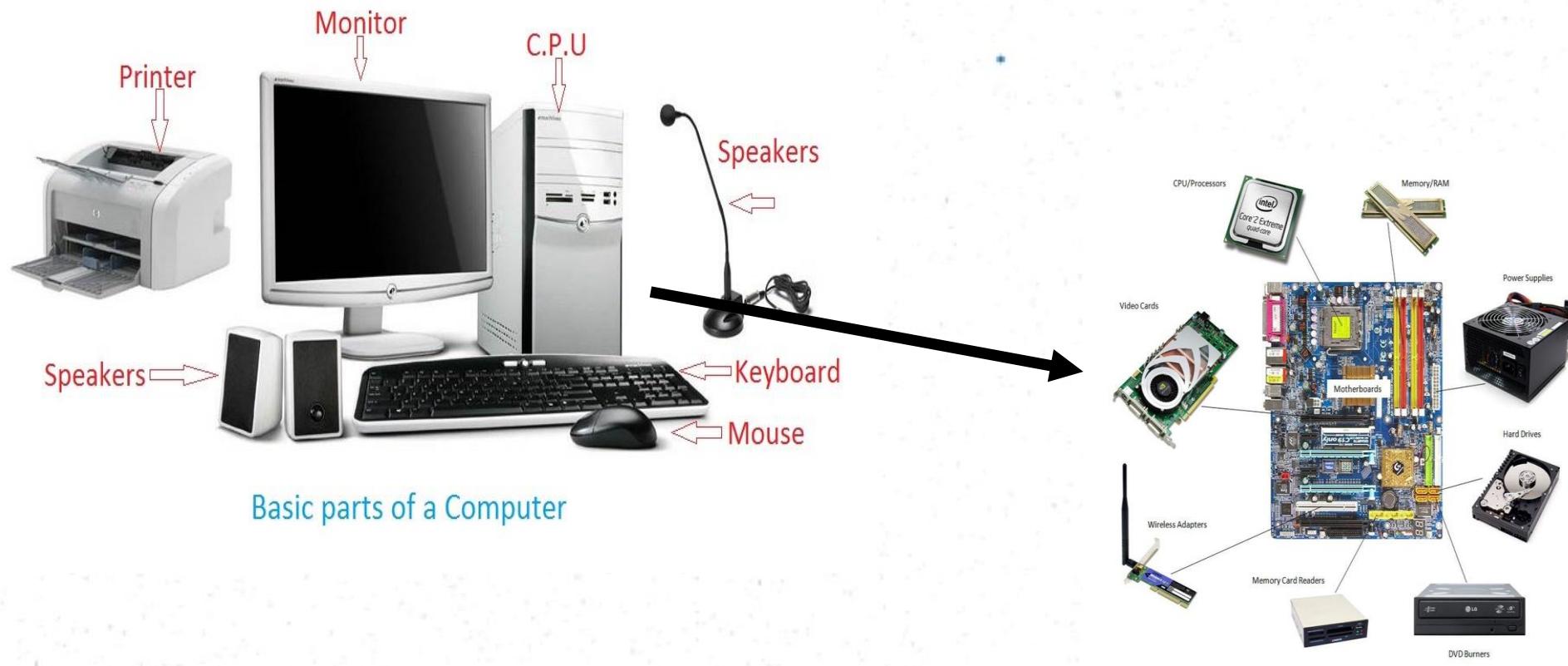
- Programs are organized as a collection of objects which cooperate to solve a problem.
- Allows to solve more complex problems easily.
- Objects contain both data and methods needed.



Object Oriented Programming

- A complex system is developed using smaller sub systems
- Sub systems are independent units containing their own data and functions
- Can reuse these independent units to solve many different problems

A Computer System



Classes

- A class is the abstract definition of the data type. It includes the data elements that are part of the data type, and the operations which are defined on the data type.
- It is an **Entity** which could be a thing, person or something that is imaginary.



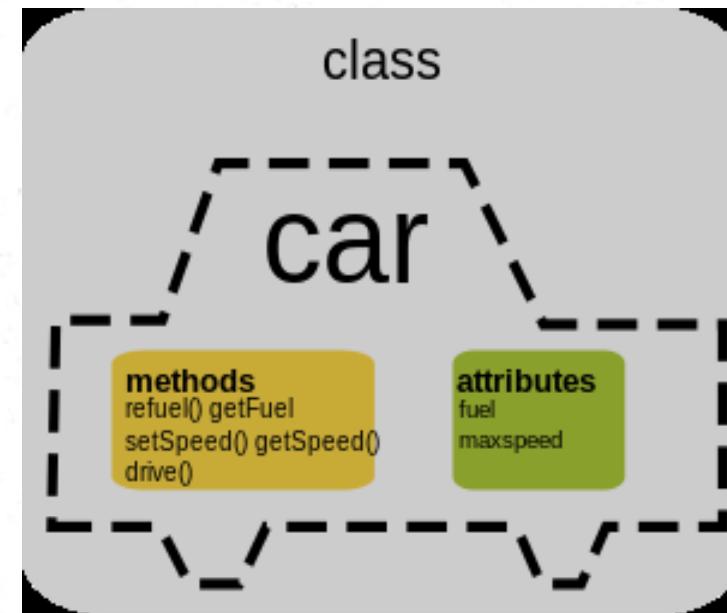
Classes

- An entity can be described by the data (properties) and its behavior (methods)

⋮
⋮
⋮

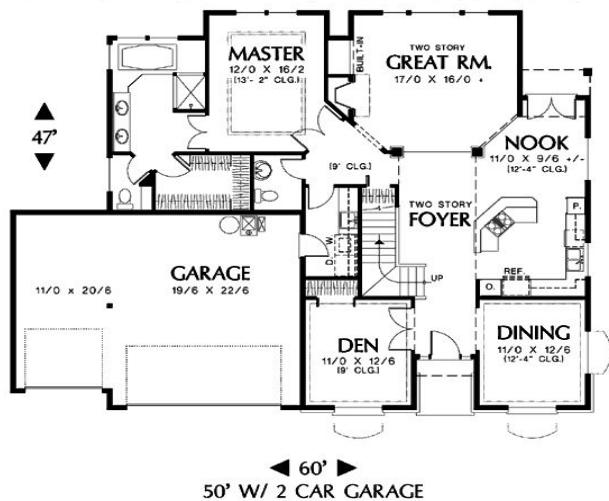
Class Name
Attributes/Properties
Methods

e.g.:



Classes and Objects

- An Object is a specific instance of the data type (class)
- A class is a blueprint of an object.



Class House



House1



House2



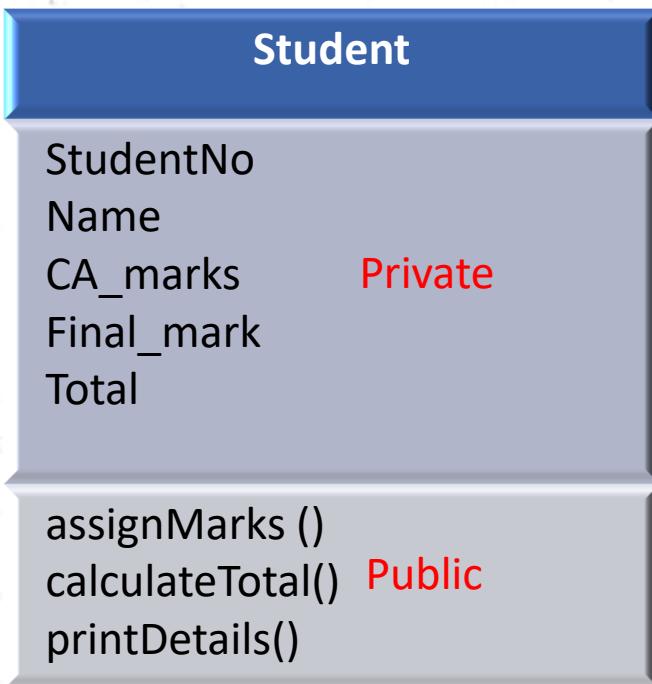
House3

Objects

Objects

- Objects are instances of classes, which we can use to store data and perform actions.
- We need to define a class including the properties and methods.
- Then create as many objects as needed which have the same structure of the class (House example).

Class in C++



```
class Student{
private :
    int studentNo;
    char name[30];
    int CA_mark;
    int Final_mark;
public:
    void assignMarks(int pCA, int pFin);
    int calculateTotal();
    void printDetails();
};
```

* *
* *
* *
* *

Class in Java

```
class Student{  
    private :  
        int studentNo;  
        char name[30];  
        int CA_mark;  
        int Final_mark;  
  
    public:  
        void assignMarks(int pCA, int pFin) {  
        }  
        int calculateTotal() {  
        }  
        void printDetails() {  
        }  
};
```

C++

```
class Student {  
    private int studentNo;  
    private String name;  
    private int CA_mark;  
    private int Final_mark;  
  
    public void assignMarks(int pCA, int pFin)  
    { }  
    public int calculateTotal() {  
    }  
    public void printDetails() {  
    }  
}
```

Java

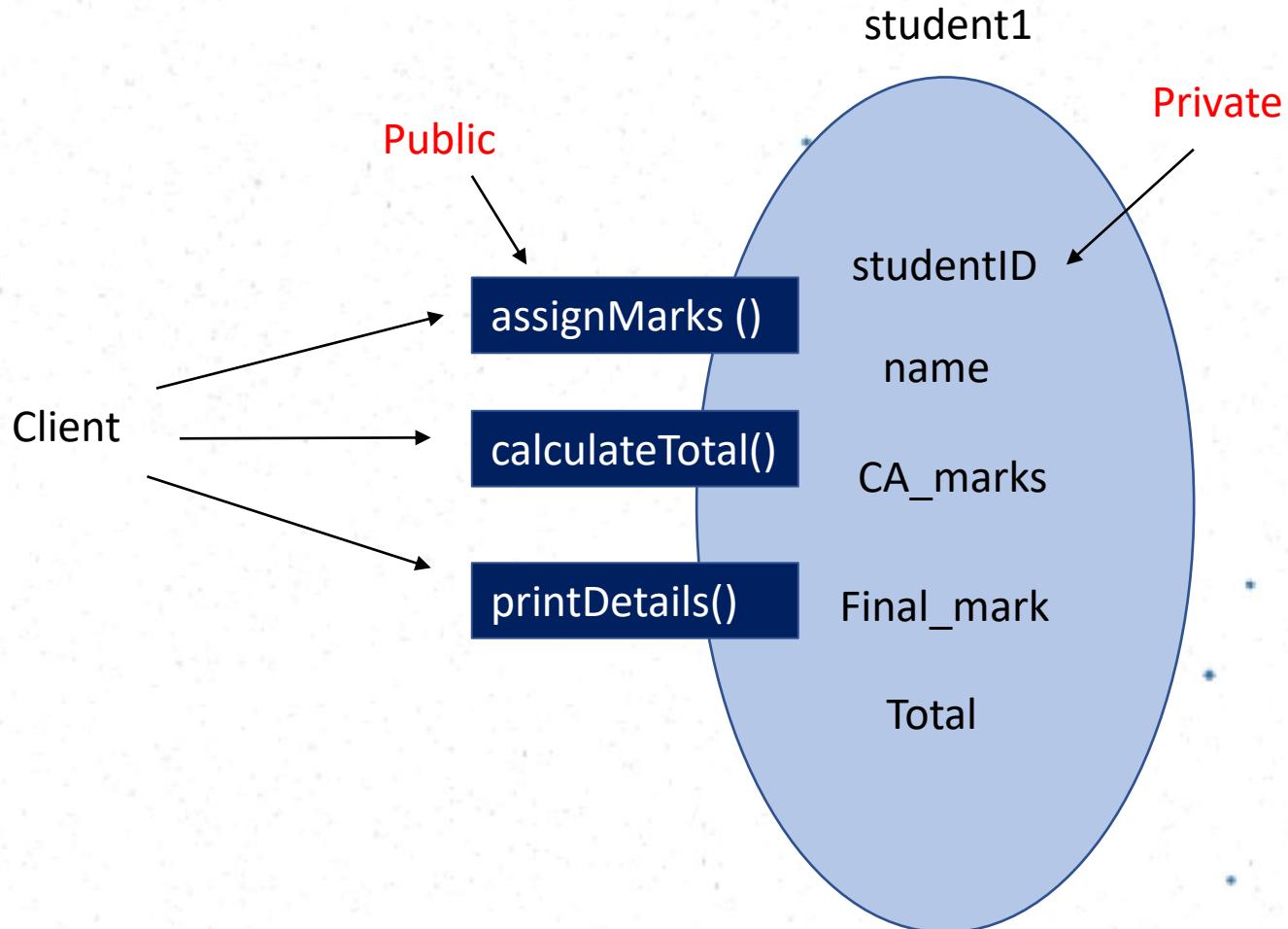
Java vs C++

- All methods are implemented in the class definition in Java.
- Each property and method require a specific access modifier (e.g. private, public, protected, friendly-default)
- In Java there is no semi colon at the end of the class
- In Java you only have dynamic objects
 - Since Java has an automatic garbage collector, you do not need to use a command line delete to remove objects from memory.
 - We use the “dot” operator instead of the “->” operator to access methods in Java.

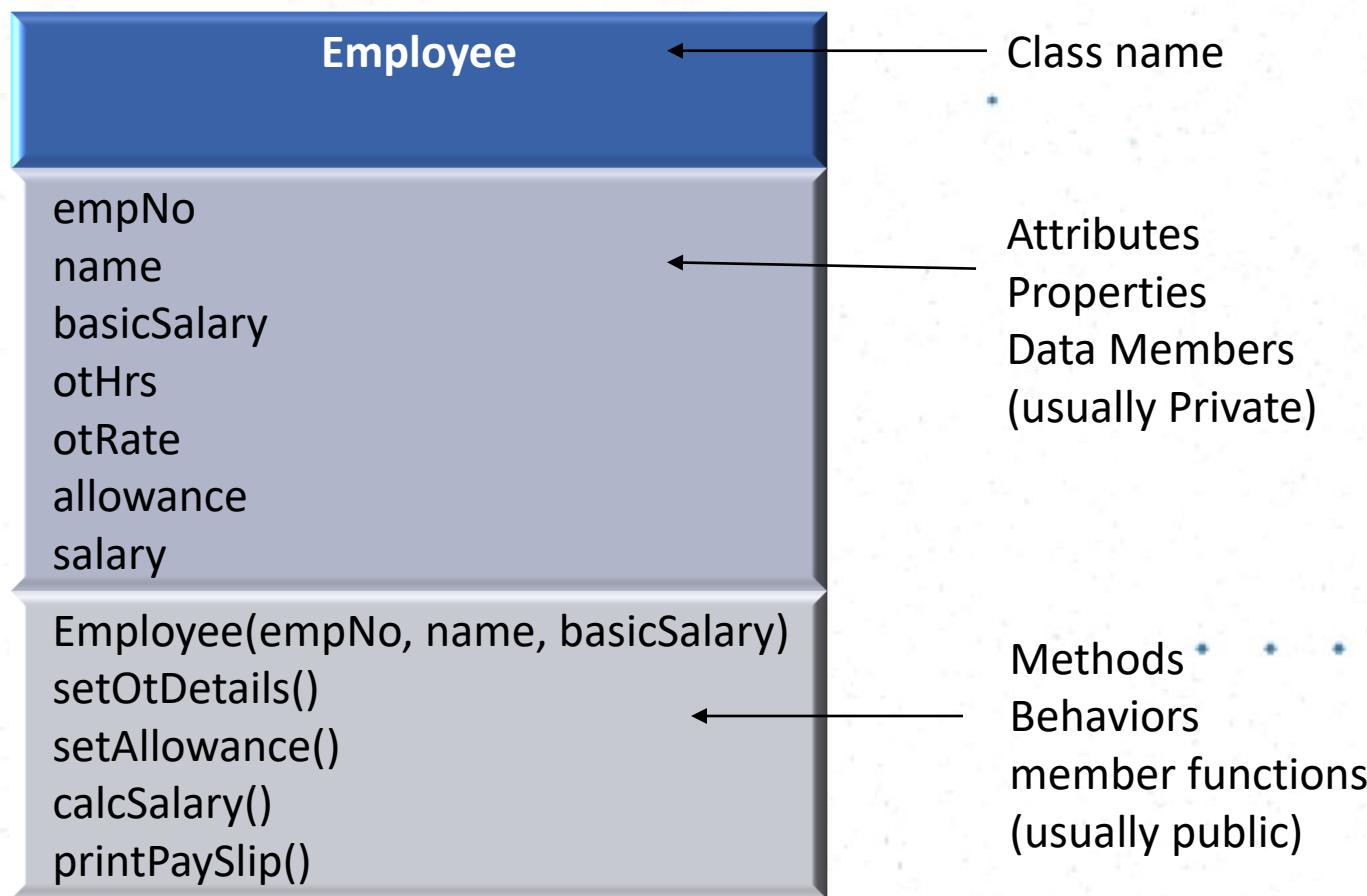
Private & Public

- The private part of the definition specifies the data members of a class
- These are hidden, not accessible outside the class and can only be accessed through the operations defined in the class
- The public part of the definition specifies the operations as function prototypes
- These operations, or methods as referred in Java, can be accessed by the main program

Private & Public



Terminology



Creating Objects

```
Student student1 = new Student();
Student student2 = new Student();
// We do not use * for pointers in Java
```

student1
studentNo – 1011
Name – Ajith Silva
CA_mark - 56
Final_mark -60

student2
studentNo – I131
Name – Surani Fernando
CA_mark - 70
Final_mark -65

Methods and Properties

C++	Java
Data members	Attributes (properties or variables)
Member functions	Methods (operations or behaviours)

- In C++ properties are called **data members**, other popular names for **properties** are **attributes, variables**.
 - In C++ **methods** are called **member functions**, other popular names are **operations, behaviors**. These are really functions.

Abstraction

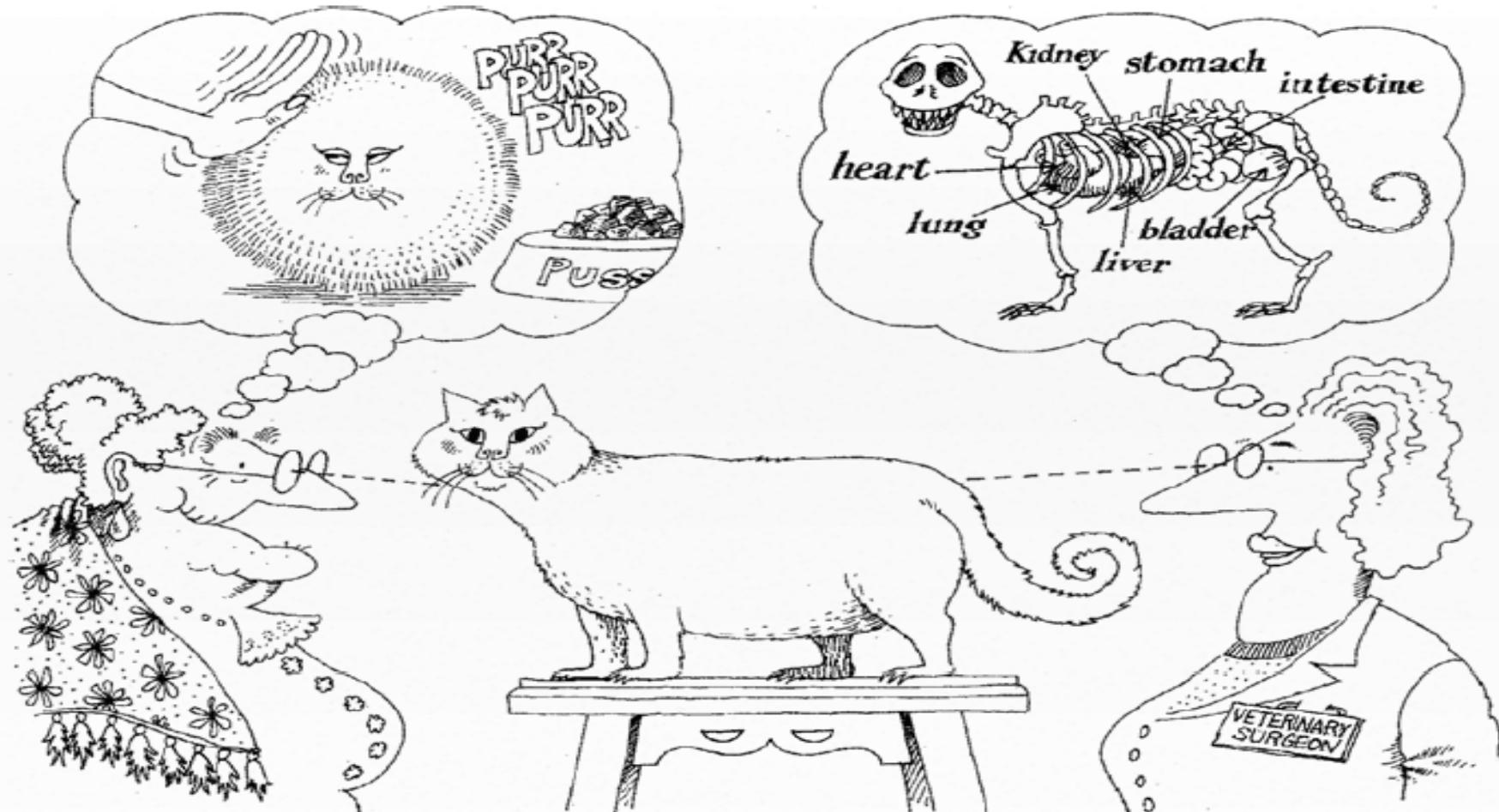
- An abstraction denotes the *essential characteristics* of an object that distinguish it from all other kinds of objects
- Thus, it provides crisply defined conceptual boundaries, relative to perspective of the viewer.

(Reference : Grady Booch, etal (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 44)

Abstraction

- Abstraction is the process of removing characteristics from ‘something’ in order to reduce it to a ***set of essential characteristics*** that is needed for the particular system.

Abstraction



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Example

Identify different classes that may exist in a Hospital Management System.

- Can you figure out different attributes of objects that exist in this system?
- What are the attributes that can be omitted?

Example contd.,

Hospital Management System

Object : Receptionist objects

Attributes: age,weight,height,bSalary,address,name,

numberofChildern,staffID, allowance, telephoneNumber

What are the attributes that can be omitted?

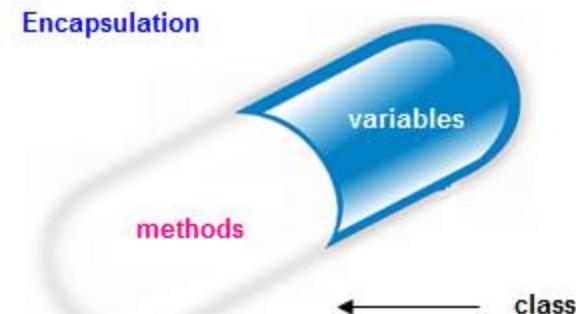
age,weight,height, numberofChildern, allowance

What are the attributes needed ?

name,staffID, address, telephoneNumber

Encapsulation

- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior.
- Encapsulation serves to separate the contractual interface of an abstraction and its implementation.
- . . .

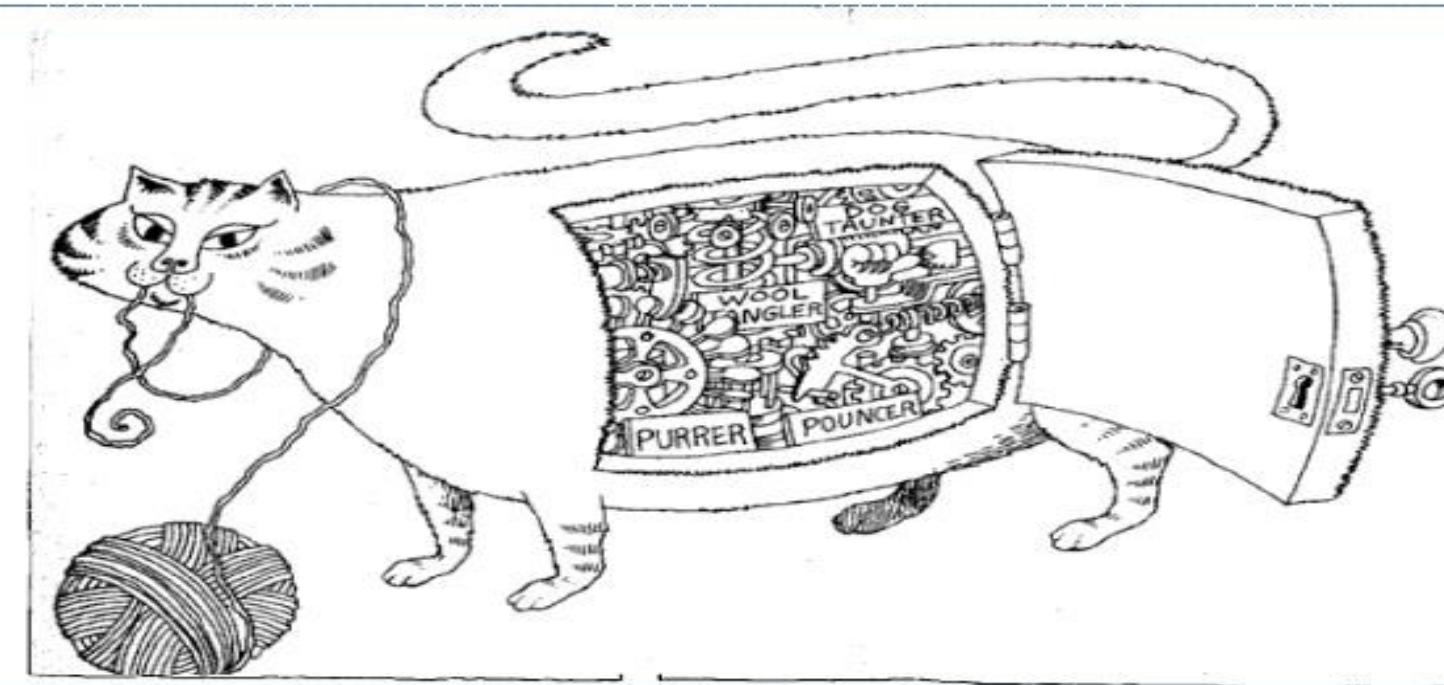


(Reference : Grady Booch, et al (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 52)

Encapsulation

Encapsulation hides the details of the implementation of an object

- •
- •
- •



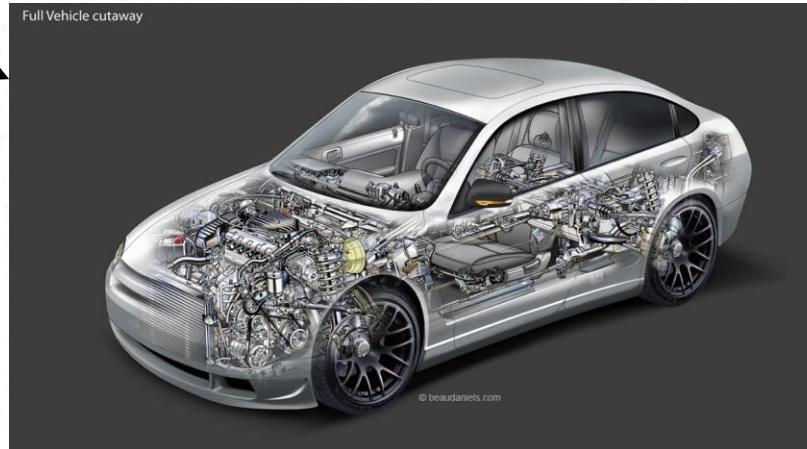
Information Hiding

- Hide certain information or implementation decision that are internal to the encapsulation structure (class)
- Objects can be made accessible through public modifiers.
 - Public – anyone can access / see it
 - Private – no one except the defining class can see/ use it
 - Java has two more modifiers as default and protected

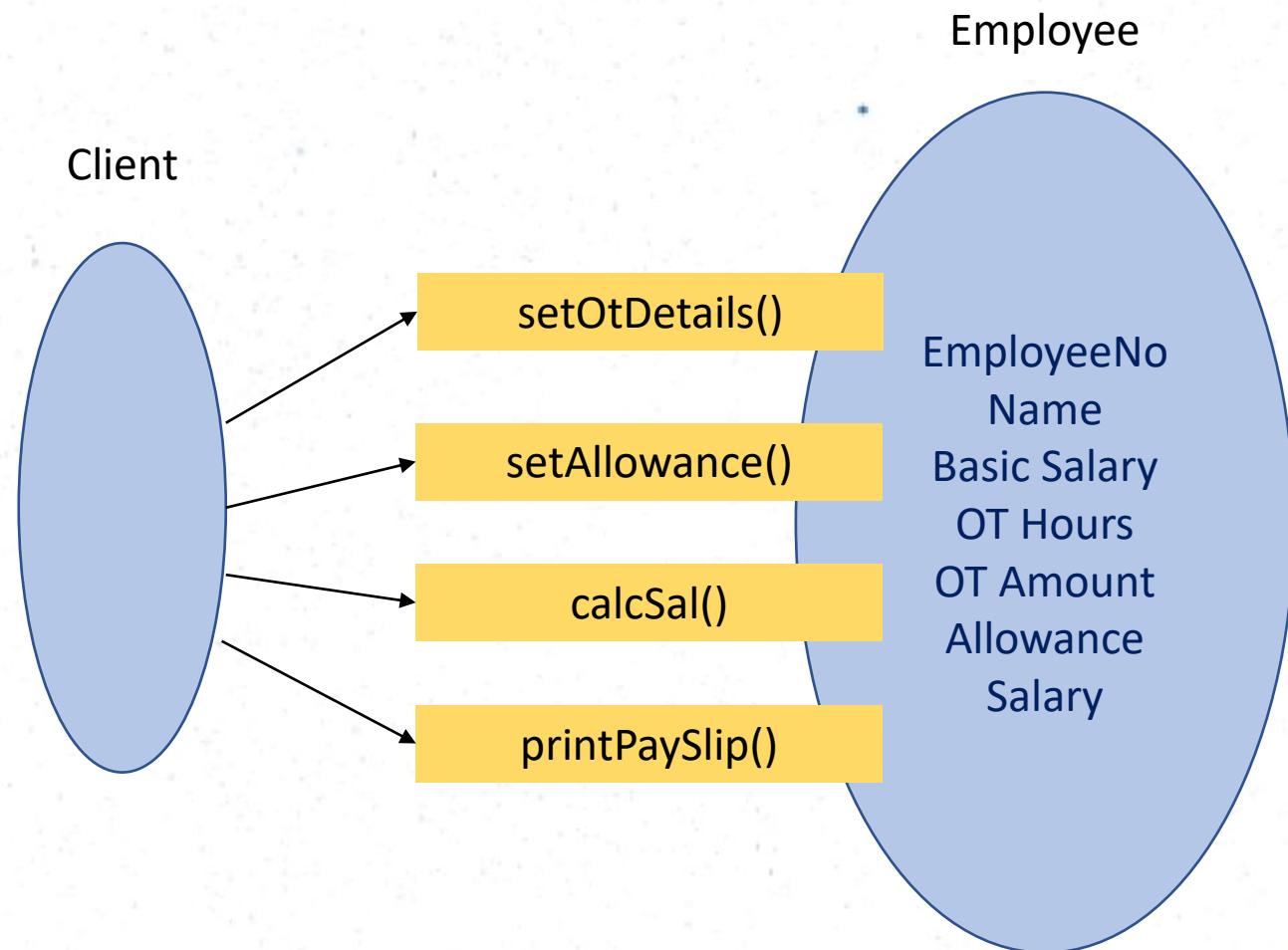
Interface



Full Vehicle cutaway



Interface



Example

- The Receptionist class define the structure and the behavior of the receptionist.
- ID, name, age define who the receptionist is.
- GenerateBill method will define what the receptionist will do.
- Such behaviors will be defined by the object which will be created from the receptionist class.
- This means the behavior of the Receptionist (or interface of a class) is defined by the methods that operate on its instance data.

Exercise 4 – Sample Answer Contd.

```
public class Receptionist{  
    private int staffID;  
    private String name;  
    private String telephoneNumber;  
    . . .  
    public Boolean checkRoomAvailability(int roomNumber){ }  
    public double generateBill(){ }  
    public void takeCustomerFeedback() { }  
}
```

Creating objects

- Implement a class called MyMain with a main method.
- Create objects from Receptionist class to demonstrate the methods.

Example

```
class MyMain {  
    public static void main(String args[]) {  
        Receptionist recep1 = new Receptionist();  
        . . .  
        boolean status = recep1.checkRoomAvailability(3);  
        double bill = recep1.generateBill();  
        recep1.takeCustomerFeedback();  
    }  
}
```

Constructor

- Constructor is used to initialize the object when it is declared.
- Constructor is a method which has the same name as the class name.
- Constructor does not return a value, and has no return type (not even void)
- There can be default constructors with no parameters and constructors with parameters
- When an object is declared the appropriate constructor is executed.

Constructors

- Default Constructors
 - Can be used to initialize attributes to default values

```
public Rectangle () {  
    width = 0;  
    length = 0;  
}
```

- Overloaded Constructors (Constructors with Parameters)
 - Can be used to assign values sent by the main program as arguments

```
public Rectangle (int w, int l) {  
    width = w;  
    length = l;  
}
```

Example

- Add a default constructor to the Receptionist class
 - And another parameterized constructor to initiate all attributes
- ⋮

Sample Answer

```
public class Receptionist{  
    private int staffID;  
    private String Name;  
    private String TelephoneNumber;  
  
    public Receptionist(){  
        this.staffID = 0;  
        this.name = "abc";  
        this.telephoneNumber = null;  
    }  
  
    public Receptionist(int pID, String pName, String pTelephoneNumber){  
        this.staffID = pID  
        this.name= pName ;  
        this.telephoneNumber = pTelephoneNumber;  
    }  
    .....  
}
```



SLIIT

Discover Your Future

IT2020

Object Oriented Programming

Lecture 03

Object Oriented Concepts – Part 2

Dr. Kalpani Manathunga

Learning Outcomes

In the previous lecture,

- Classes and Objects
- Abstraction
- Encapsulation

At the end of the Lecture you should know

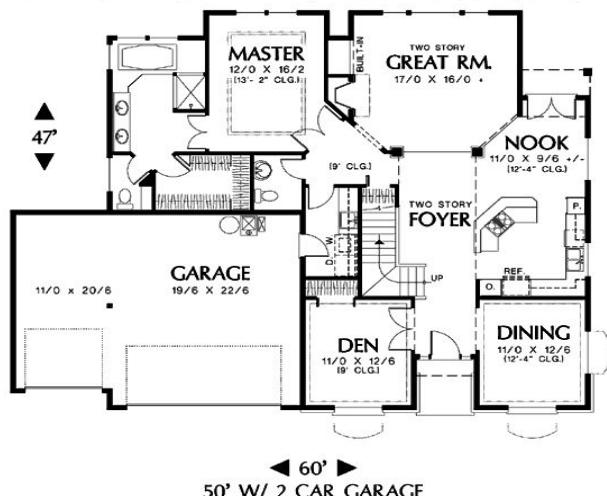
- Get and set methods
- Inheritance
- Polymorphism

Object Oriented Programming

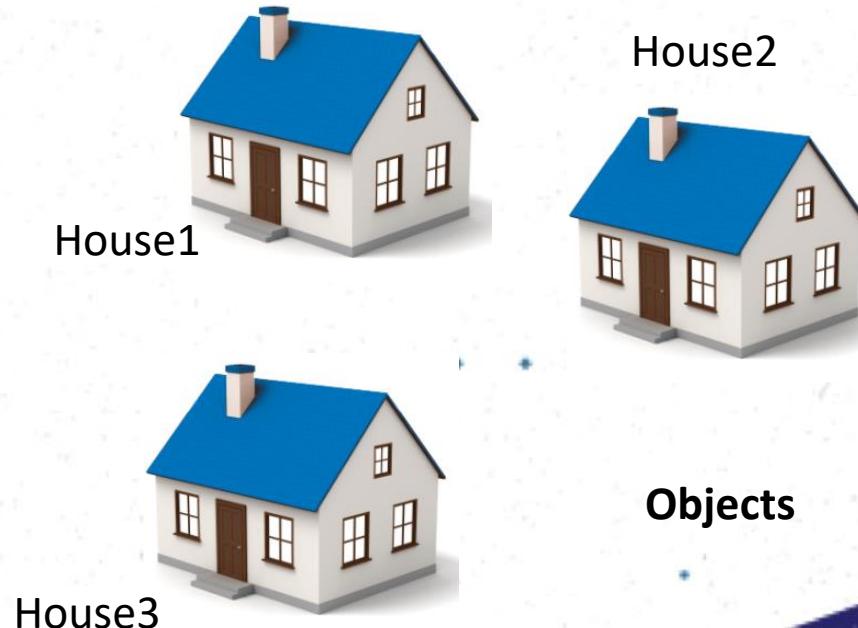
- Complex problems are broken into smaller subsystems or modules, each solving a particular subproblem
- Set of objects interact with one another
- Objects are derived from class definitions, contain data and methods

Classes and Objects

- A Class is an entity described using data members and methods
- An Object is a specific instance of the data type (class)



Class House

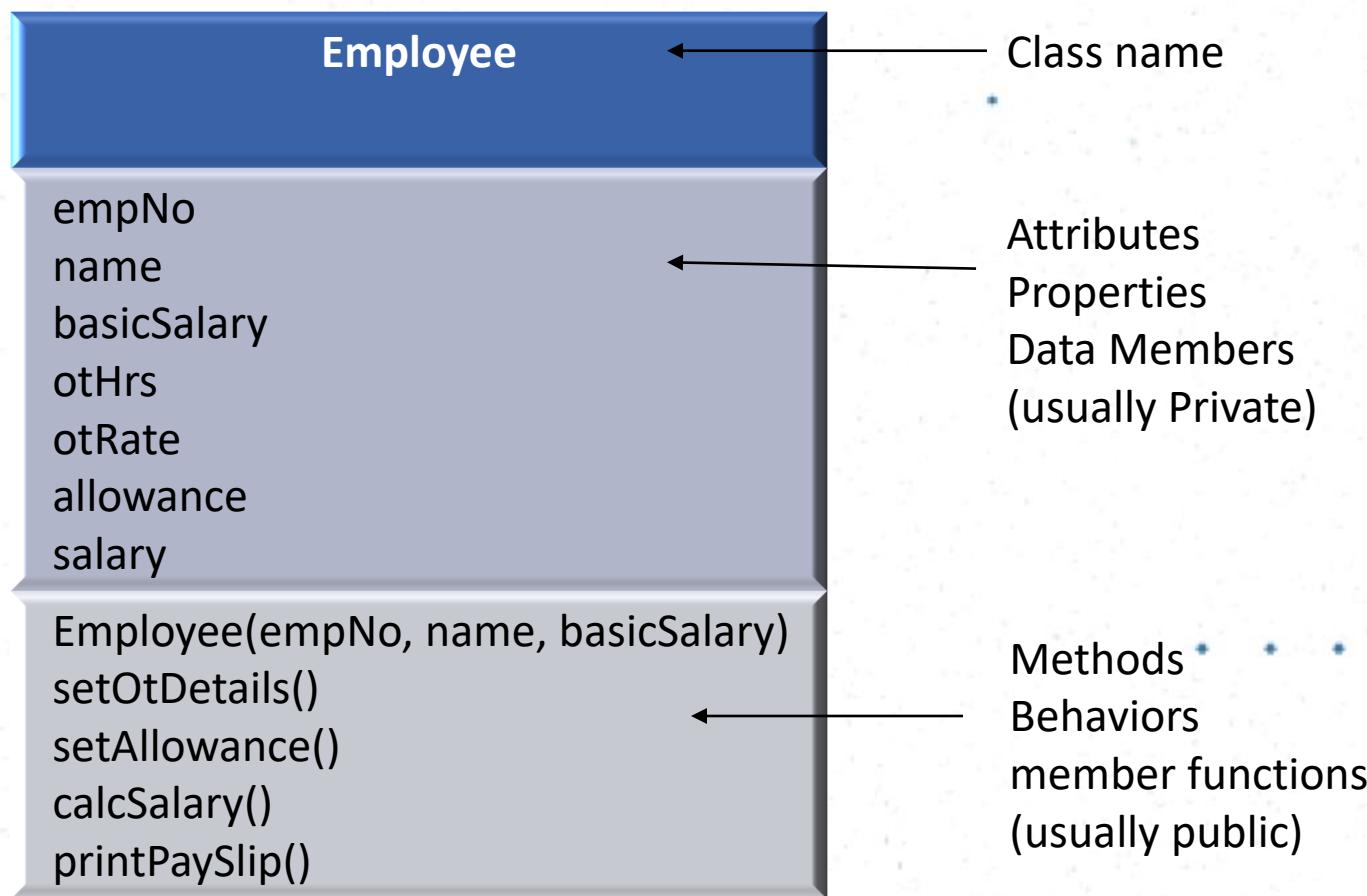


House2



Objects

Terminology



Getters and Setters

- In general properties are declared as *private* preventing them from being accessed from outside the class
- Typically an attribute will have a getter (accessor) - a get method to return its value
- And a setter (mutator) - a set method to set a value
- e.g. a property called length will have a getter defined as `int getLength()` and a setter defined as `void setLength()`. Both these methods will be declared as public methods.

“this” keyword

- “this” **keyword** can be used to refer to any member of the current object within an instance method or a constructor.

```
public Employee(int pempno, String name, double pbasicSal) {  
    employeeNo = pempno;  
    this.name = name;  
    basicSalary = pbasicSal;  
}
```

- We need to use `this.name` to refer to the property name to distinguish it from the parameter name.

Exercise

- Add getters and setters to the Receptionist class.

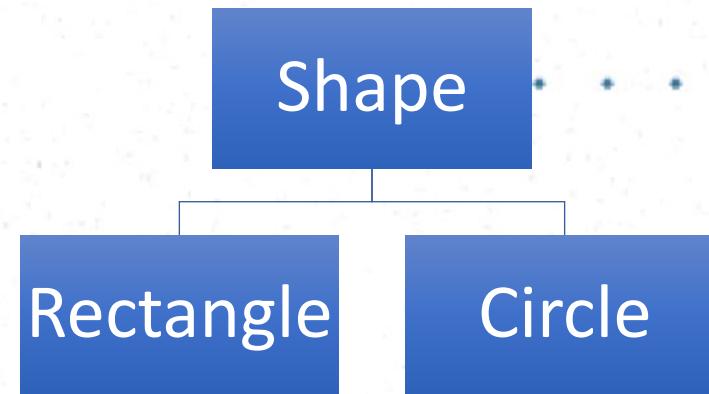


Exercise – Sample Answer

```
public class Receptionist{  
    .....  
    public Receptionist(int pID, String pName, String pTelephoneNumber){  
        this.staffID = pID  
        this.name= pName ;  
        this. telephoneNumber = pTelephoneNumber;  
    }  
    public void setID(int ID){  
        this.staffID =ID;  
    }  
    public int getID(){  
        return this.staffID;  
    }  
}
```

Generalization/Inheritance

- Inheritance is a mechanism in which one object acquires all properties and behaviors of a parent object
- Child class **is a type** of the parent class
- Inheritance promotes code reusability
- Child classes “inherit” the attributes and methods defined in the parent class



C++ vs Java - Inheritance

C++

```
class Circle : public Shape {
```

Java

```
class Circle extends Shape {
```

Java has a simpler inheritance mechanism where base class is extended as public.

C++ has multiple inheritance compared to Java's Single Inheritance.

C++ vs Java - Inheritance

C++

```
Circle (string tname, int r) : Shape ( tname) {  
    radius = r;  
}
```

Java

```
public Circle (String tname, int r) {  
    super(tname);  
    radius = r;  
}
```

When you want to call a base class constructor C++ Requires to explicitly name the base class. In Java we use the super keyword to access the direct descendent class.

However this implies that in Java you can't directly call a class higher in the hierarchy e.g. the Grandfather class which is not in C++

C++ vs Java - Inheritance

C++

```
virtual void speak() {}
```

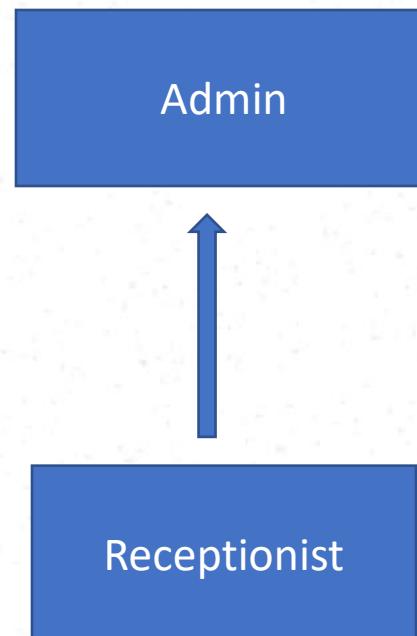
Java

```
public void speak() {}
```

- All methods in Java are virtual by default. Hence, methods are overridable
- In C++ we need to explicitly define polymorphic methods using virtual keyword. Then only sub classes can override such methods.

Example

Think of a way to implement generalization for our example scenario, Hospital Management System

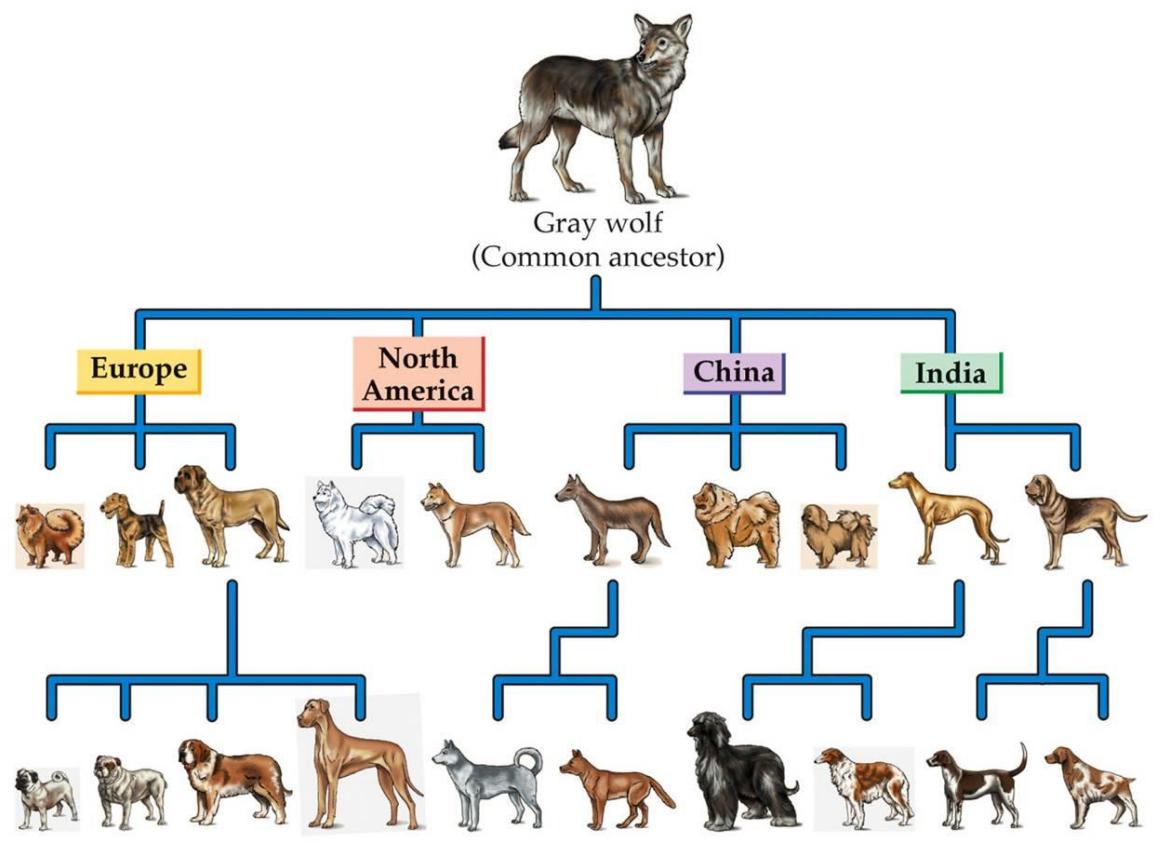


Example from Receptionist class

```
public class Admin{  
    protected int staffID;  
    protected String name;  
}  
  
public class Receptionist extends Admin{  
    private String telephoneNumber;  
}  
}
```

Object Class

- A class hierarchy is like the taxonomy of animals that shows their ancestry.
- There are many breeds of dogs. A well-known fact is that the common origin of a dog is the Gray Wolf.
- All Java classes are derived from a class called **Object**.
- This includes all the existing Java built in classes and the classes that you write.
- The methods and properties of the Object class are accessible to any Java class that you create.
- [Object class](#) in the Javadoc



DISCOVER BIOLOGY, Second Edition, Chapter 21 Box © 2002 Sinauer Associates, Inc., and W. W. Norton and Company

Inheritance Shapes Example C++

```
23  class Rectangle: public Shape{  
24      protected:  
25          int width;  
26          int height;  
27      public :  
28          Rectangle (string tname, int w, int h) : Shape ( tname)  
29          {  
30              width = w;  
31              height = h;  
32          }  
33          int area( )
```

Shape_example.cpp

Inheritance Shapes Example Java

```
1  class Shape {  
2      protected String name;  
3      public Shape() {};  
4      public Shape (String tname) { ...  
5      }  
6      public void print() { ...  
7      }  
8      public int area(){ return 0; }  
9  }  
10 }  
11 }  
12 class Rectangle extends Shape { ...  
13 }  
14 }  
15 class Circle extends Shape { ...  
16 }  
17 }  
18 class ShapeApp {  
19     public static void main(String args[]) {  
20         Rectangle R = new Rectangle("Rectangle", 4 , 6);  
21         Circle C = new Circle("Circle", 3 );  
22     }  
23 }
```

Shape_example.java

Polymorphism

- Ability to assign a different meaning or usage to something in different contexts
- Ability of an object to take many forms. Common with Inheritance concept
- Consider the request (analogues to a method) “*please cut this in half*” taking many forms



For a cake:

- Use a knife
- Apply gentle pressure

For a cloth:

- Use a pair of scissors
- Move fingers in a cutting motion

Animal Example

```
2  + class Animal { ...
20 }
21 + class Cat extends Animal { ...
30 }
31 + class Dog extends Animal { ...
40 }
41 + class Cow extends Animal { ...
50 }
51 - class AnimalApp {
52     public static void main(String args[]) {
53         Animal ani[] = new Animal[4];
54         ani[0] = new Cat("Micky the Cat");
55         ani[1] = new Dog("Rover the Dog");
56         ani[2] = new Cow("roo the Cow");
57         ani[3] = new Animal("no name");
58     - for (int r=0;r<4; r++)
59         ani[r].song();
```

Animal_example.cpp

Animal_example.java

Thank you!

Object Oriented Programming

Week 03
Reference Materials

Dynamic Objects

```
Rectangle *r;  
  
r = new Rectangle ();  
  
r -> setWidth(100);  
r -> setLength(50);  
cout<<"Area is : "<< r -> calcArea();  
  
delete r;
```

The arrow (->) is used to access the public methods of a dynamic object

Dynamic Objects

- Most programming languages only support Dynamic Objects.
- You need to delete the allocated memory in C++.
- The new command is used to allocate memory for an object.
- The delete command needs to be used to deallocate memory (release memory) once we have finished using the objects.

Dynamic Objects

```
64 Rectangle *rec3, *rec4;  
65  
66 rec3 = new Rectangle();  
67 rec4 = new Rectangle(20, 10);  
68  
69 cout << "Rectangle 3 = length - "  
70     << rec3->getLength()  
71     << ", width - "  
72     << rec3->getWidth()  
73     << endl;  
74  
75 cout << "Rectangle 4 = length - "  
76     << rec4->getLength()  
77     << ", width - "  
78     << rec4->getWidth()  
79     << endl;  
80  
81 delete rec3;  
82 delete rec4;  
83
```

```
4 class Rectangle {  
5     private:  
6         int width;  
7         int length;  
8     public:  
9         Rectangle();  
10        Rectangle(int l, int w );  
11        void setWidth(int w);  
12        int getWidth();  
13        void setLength(int l);  
14        int getLength();  
15        int calcArea();  
16        ~Rectangle();  
17    };
```

Output

```
Rectangle 3 = length - 0, width - 0  
Rectangle 4 = length - 20, width - 10  
Destructor Runs for Rec with Len = 0 and width = 0  
Destructor Runs for Rec with Len = 20 and width = 10
```

Dynamic Objects

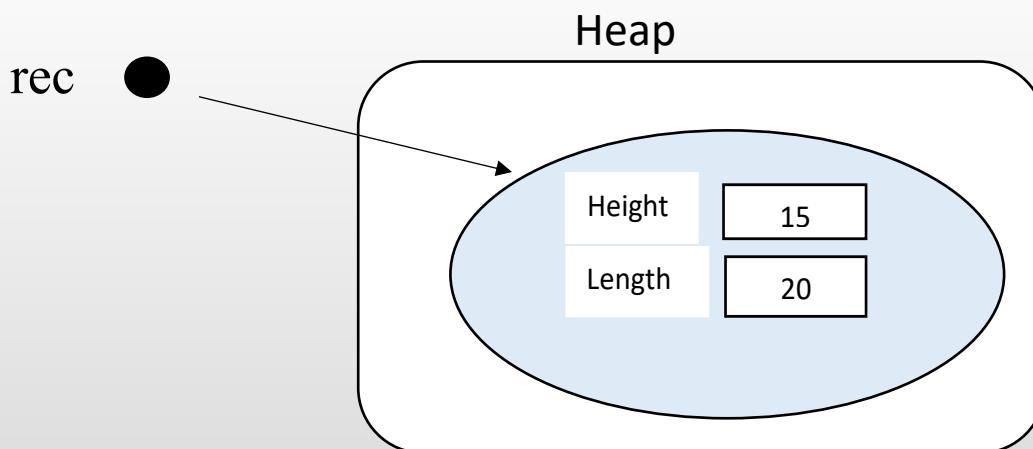
- Most programming languages only support Dynamic Objects.
- **Java objects** are similar to C++ **dynamic Objects** as bellow.

Dynamic Objects

```
Rectangle rec; // currently this references no object
```

rec ●

```
rec = New Rectangle(15,20); // new operator returns a reference to the object  
it created
```



Overloaded Constructors

Base class

```
class Shape
{
protected:
    char name[20];
public:
    Shape();
    Shape ( char tname[])
    {
        name = tname;
    }
};
```

Derived class

```
class Rectangle: public Shape{
protected:
    int length;
    int width;
public:
    Rectangle (char tname[], int l, int w) : Shape ( tname)
    {
        length = l;
        width = w;
    }
};
```

Sample Code

Overriding methods – area()

Base class

```
class Shape
{
    protected:
        char name[20];
    public:
        Shape();
        ...
        int area() {
            return 0;
        }
    };
}
```

Derived class

```
class Rectangle: public Shape{
    protected:
        int length;
        int width;
    public :
        Rectangle();
        ...
        int area() {
            return length * width;
        }
    };
}
```

Sample Code

Overriding

- In the previous example we saw that the Rectangle class redefines the area method()

```
int area()
```

- This definition is exactly the same as in the Shape class. This is called overriding.
- If a sub class has a different behavior of a method we can override it and redefine the code associated.
- Please Note that in overloading there is a difference in parameters and overloading usually happens within the same class.

Overriding

- Consider the following code

```
Shape sh("myshape");
Rectangle rec("whiteboard", 10,5);
cout << sh.area(); // will produce 0, Shape::area() called
cout << rec.area(); // will produce 50, Rectangle::area() called
```

Passing Objects to methods

- In OOC when we covered Aggregation we did an example where we passed an Employee object to a method in the Department class.

Aggregation



- Whole : Department
- Part : Employee
- A Department has one or more employees
- This implies that the **Part can exist without the Whole.**

Aggregation

- In aggregation the objects have their own life cycles, but there is a ownership.
- The Department and Employee objects have their own life cycles.
- If the Department object is deleted, still the Employee objects can exist.
- If the Employee objects is deleted, still the Department object can exist.

Aggregation – C++ implementation

```
class Employee
{
private :
    string empID;
    string name;
public :
    Employee(string pempID, string pname)
    {
        empID = pempID;
        name = pname;
    }
    void displayEmployee()
    {
        cout << "empID = " << empID << endl;
        cout << "name = " << name << endl;
        cout << "*****" << endl;
    }
    ~Employee() {cout << "Deleting Employee" << empID << endl;
    }
};
```

Sample Code

Aggregation - C++ implementation

```
class Department
{
private:
    Employee *emp[2];
public:
    Department() {};
    void addEmployee(Employee *emp1, Employee *emp2)
    {
        emp[0] = emp1;
        emp[1] = emp2;
    }
    void displayDepartment()
    {
        for(int i = 0; i < SIZE; i++)
            emp[i]->displayEmployee();
    }
~Company() {cout << "Department shutting down" << endl;
}
};
```

Aggregation - C++ implementation

```
int main()
{
    Department*ABC = new Department();
    Employee *e1 = new Employee("E001",
"Nimal");
    Employee *e2 = new Employee("E002",
"Jagath");
    ABC->addEmployee(e1, e2);
    ABC->displayDepartment();
    delete ABC;
    e1->displayEmployee();
    e2->displayEmployee();
    return 0;
}
```

After the company ABC is deleted the two employees exists.

Output

```
empID = E001
name = Nimal
*****
```

```
empID = E002
name = Jagath
*****
```

Company shutting down

```
empID = E001
name = Nimal
*****
```

```
empID = E002
name = Jagath
*****
```

Aggregation - C++ implementation

```
int main() {
    Department*ABC = new Department();
    Employee *e1 = new Employee("E001",
"Nimal");
    Employee *e2 = new Employee("E002",
"Jagath");
    ABC->addEmployee(e1, e2);
    delete e1;
    delete e2;
    Employee *e3 = new Employee("E003",
"Kamal");
    Employee *e4 = new Employee("E004",
"Lal");
    ABC->addEmployee(e3, e4);
    ABC->displayDepartment();
    return 0;
}
```

After E001 and E002 is deleted still the company exist and new employees can be added.

Output

Deleting EmployeeE001

Deleting EmployeeE002

emplID = E003

name = Kamal

emplID = E004

name = Lal

Object Oriented Programming

Week 03

A few Things specific to Java

Learning Outcomes

At the end of the Lecture students should be able to get details of a few specific things related to the Java Programming Language.

- Primitive Data types
- Object Memory Allocation
- Object variables are reference type parameters
- Static properties and methods
- Final keyword
- Passing Objects as parameters
- Returning Objects
- Overloading vs Overriding

The Primitive Types

- Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.
 - Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
 - Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
 - Characters This group includes char, which represents symbols in a character set, like letters and numbers.
 - Boolean This group includes boolean, which is a special type for representing true/false values.

Integer Data Type

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

- Unlike languages like C, C++ the sizes of Integers, Floats are fixed and are platform neutral.

Float Data Type

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

- Unlike languages like C, C++ the sizes of Integers, Floats are fixed and are platform neutral.

Type Conversion and Casting

- Auto Conversion happens in Java for Simple Data Types in situations
 - The two types are compatible.
 - The destination type is larger than the source type.
- Manual Casting is required when the destination is smaller than the source type.

- e.g.

```
double d = 56.0;  
int no = (int) d;
```

One Dimensional Arrays

- Slightly different from C/C++
 - Define Array
 - Allocate Memory

```
int data[]; // Array declaration  
data = new int[10]; // Allocating Memory
```

Objects and Memory

- Lets consider how an Object is allocated in memory.

```
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
class BoxDemo2 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double val
```

Objects and Memory

- Declaring Objects in Java is a two step process.
 - First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
 - Second, you must use the **new** operator to dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.
 - This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Objects and Memory

- Lets consider how an Object is allocated in memory.

Statement

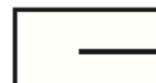
```
Box mybox;
```

Effect

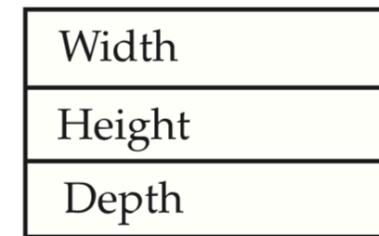


mybox

```
mybox = new Box();
```



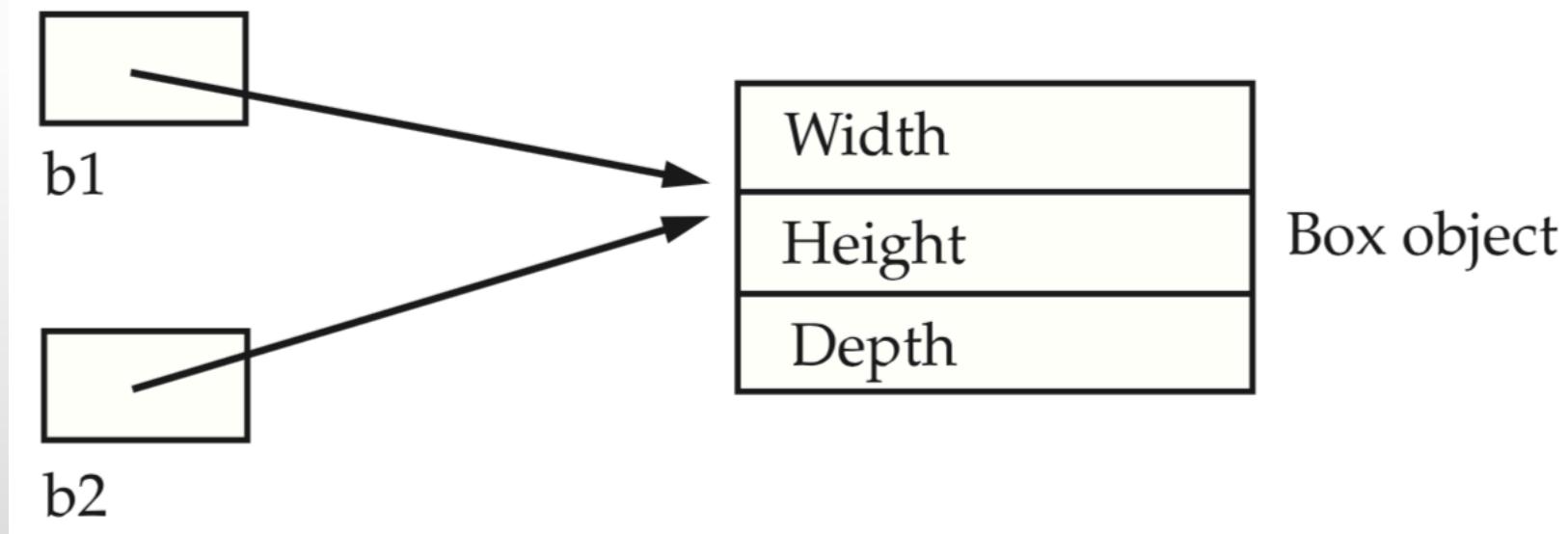
mybox



Box object

Objects and Memory

```
Box b1 = new Box();  
Box b2 = b1;
```



Static Members

- Attributes and methods (members of a class) can be defined as static.
- Static members do not belongs to an individual object.
- Static members are common to all the instances (objects of the same class).
- Static members are stores in static memory (a common memory location which can be everybody)

Static Members

```
class Student {  
    private String ditno;  
    private String name;  
    private static String batchId;  
  
    public Student(String mditno, String mname) {  
        ditno = mditno;  
        name = mname;  
    }  
    public void setBatchId(String mbatchId) {  
        batchId = mbatchId;  
    }  
}
```

StaticDemo.java

Static Members

```
public static void setBatchId2(String mbatchId) {  
    batchId = mbatchId;  
}
```

- `setBatchId2()` is a static method
- Static Methods can be called directly using the class name.

```
Student.setBatchId2("Metro Y1B1");
```

Static Members

```
public class Static {  
    public static void main(String args[]) {  
        Student s1 = new Student("IT15123412", "Tharidi");  
        Student s2 = new Student("IT15132343", "Kumudu");  
        s1.setBatchId("Malabe - Y1B2");  
        System.out.println(s1.getBatchId() + " - " + s1.getDitNo());  
        System.out.println(s2.getBatchId() + " - " + s2.getDitNo());  
        Student.setBatchId2("Metro Y1B1");  
        System.out.println(s1.getBatchId() + " - " + s1.getDitNo());  
        System.out.println(s2.getBatchId() + " - " + s2.getDitNo());  
    }  
}
```

Metro Y1B1

IT15123412
Tharidi

IT15132343
Kumudu

StaticDemo.java

In this code the methods `setBatchId()` and `setBatchId2()` both change the static variable `batchId`. This is common to both objects `s1` and `s2` and stored in the stack where as the instance variables `name` and `ditno` are different for each object.

Static Modifiers

- The static modifier indicates that the attributes and methods are common to all the objects in the whole class rather than to an individual object.
- A static method does not operate on an object:
- `ClassName.methodName(parameterList)`
- Many attributes and methods we use are static:

`System.out`

`Integer.parseInt()`

`System.in`

`Character.isLetter()`

`Color.RED`

`Math.random()`

`Math.PI`

Understanding Static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Understanding Static

- Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.
- Methods declared as **static** have several restrictions:
 - They can only directly call other **static** methods.
 - They can only directly access **static** data.
 - They cannot refer to **this** or **super** in any way.

Static Demo (JTCR pg 145)

```
class Main {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;  
    }  
    public static void main(String args[]) {  
        meth(42);  
    }  
}
```

StaticDemo2.java

The static variables and the code is executed when the class is loaded.

final properties

- Final is used to declare constants.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- These can be declared as above in the class or initialized in the constructor.
- A static final variable is a global constant.

Passing Object as a Parameter

(JTCR pg 138)

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // return true if o is equal to the invoking object  
    boolean equalTo(Test o) {  
        if(o.a == a && o.b == b) return true;  
        else return false;  
    }  
}
```

PassingObjects.java

Here a Test object is passed as a parameter to the equalTo() method

Object Parameter to a Constructor

(JTCR pg 135)

```
// Here, Box allows one object to in:  
class Box {  
    double width;  
    double height;  
    double depth;  
    // Notice this constructor. It takes  
    Box(Box ob) { // pass object to con-  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
}
```

ObjectConstructor.java

Here a Box type object is passed to the overloaded Box constructor.

Returning an Object (JTCR pg 138)

```
// Returning an object.  
class Test {  
    int a;  
    Test(int i) {  
        a = i;  
    }  
    Test incrByTen() {  
        Test temp = new Test(a+10);  
        return temp;  
    }  
}
```

ReturnObjects.java

A Test object is created and returned by the method incrByTen()

Passing Objects as References (JTCR pg 175)

```
// Objects are passed through references
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}
```

ObjectReference.java

In the meth() method the parameter o is an object, any changes will affect the arguments that are sent.

Overloading

- Overloading occurs when there are methods with different signatures.

Overloading Demo (JTCR pg 129)

```
class OverloadDemo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    // Overload test for one integer parameter.  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
    // Overload test for two integer parameters.  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
    // Overload test for a double parameter  
    double test(double a) {  
        System.out.println("double a: " + a);  
        return a*a;  
    }  
}
```

Overloading.java

The test() method is overloaded.

Overriding

- Overriding occurs in inheritance when a descendant class replaces a method with the same signature.

Overriding Demo (JTCR pg 175)

```
class B extends A {  
    int k;  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
    // display k - this overrides show() in A  
    void show() {  
        System.out.println("k: " + k);  
    }  
}
```

Override.java

The show() method in class B overrides the show() method in the class A

References

- JTCR – Java the Complete Reference, Herbert Schildt, 9th Edition, Oracle Press.

Object Oriented Programming

Week 04

Abstract Classes, Interfaces and
Exception Handling

Learning Outcomes

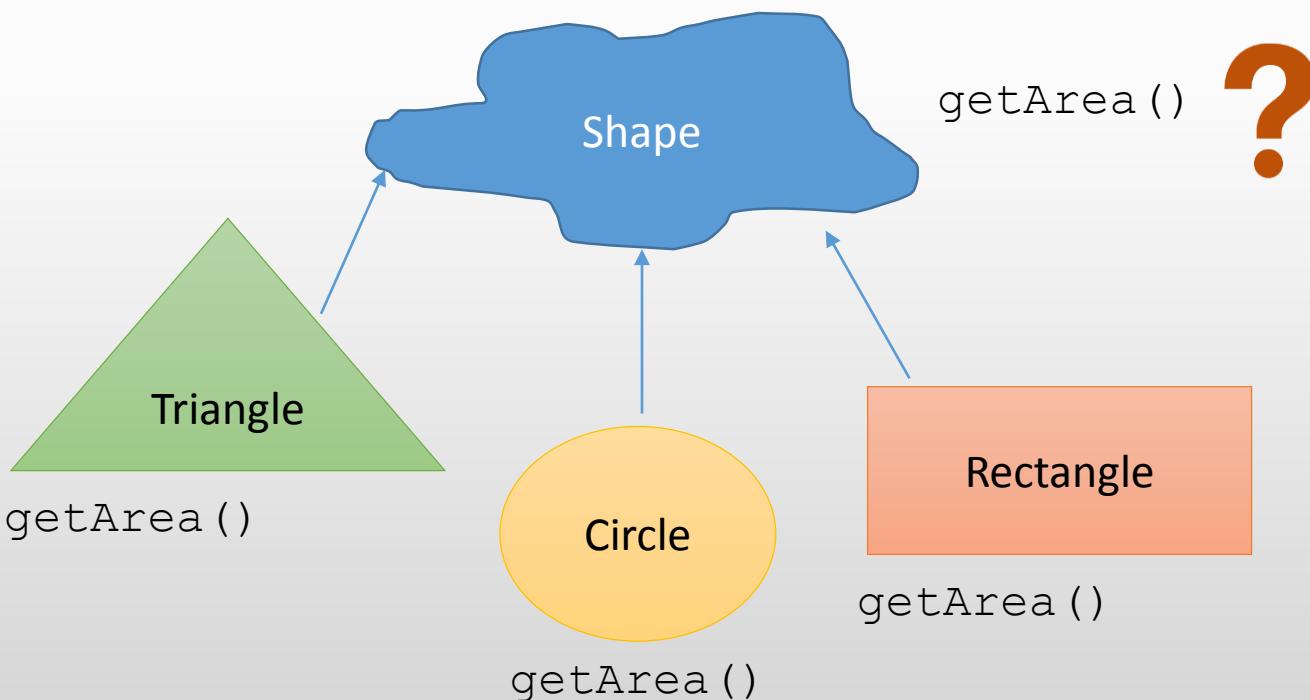
At the end of the Lecture students should be able to apply the following concepts in the programs that you write.

- Abstract Classes
- Interfaces
- Exception Handling – Catching runtime Errors
- Packages
- Access Modifiers default

Abstract Classes

- Used in situations in which you will want to define a superclass where some methods don't have a complete implementation.
- We are expecting the sub classes will implement these abstract methods.
- This situation is applicable when a superclass is unable to create a meaningful implementation for a method. e.g an `getArea()` method of a `Shape` class.

Abstract Class



Abstract Classes

- Refer the following class called Shape.

```
class Shape {  
    public double getArea() {  
        // How to Implement this code  
    }  
}
```

- Any Shape has an area. So that class shape can contain a method called getArea()

But how to calculate the area of a shape?

If we are asked to calculate the area of a circle, then we know how to calculate the area of a circle.

```
class Circle extends Shape {  
    double radius;  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
}
```

Abstract Classes

- `getArea()` method will have different implementations depending on the child class.
- Class `Shape` behave as a super class.
- Any shape has an area. So that class `Shape` can contain a method called `getArea()`.
- But implementing `getArea()` method is possible only when we know the child class.
- The implementation of `getArea()` method is different from one child class to another.

```
class Rectangle extends Shape {  
    double width, height;  
    public double getArea() {  
        return width * height;  
    }  
}
```

Abstract Class

- The methods which cannot be implemented MUST be defined as abstract.

```
abstract class Shape {  
    abstract public double getArea();  
}
```

- An abstract method does not have a method implementation (**not even {}**).
- We cannot invoke (call) an abstract method.
- The class which contain at least one abstract method MUST be defined abstract

Abstract Class

- Every child class MUST override all the abstract methods of the parent class.
- If a child class did not override all the abstract method of the parent, then the child class also become abstract.

Abstract Classes

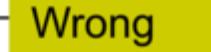
- Cannot create instances (objects) from an abstract class.
- Abstract Classes force the child class to contain methods defined by the parent class.
- Their purpose is to behave as parent (base) classes.
- Abstract classes are very generic
- Usually a class hierarchy is headed by an abstract class
- Classes from which objects can be instantiated - concrete
- The ability to create abstract methods is powerful – each new class that inherits is forced to override these methods.
ex. Mouse clicks and drags

Abstract Classes

- An abstract class can behave as a data type.

Shape s1; 

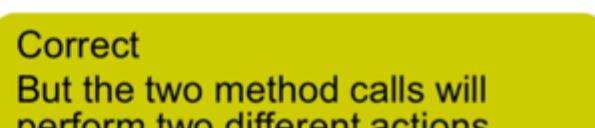
- But cannot create instances (object) from an abstract class.

s1 = new Shape(); 

- But super class variables can refer to child class objects.

s1 = new Circle();
Shape s2 = new Rectangle() 

- Can call the getArea() method belongs to the child.

s1.getArea();
s2.getArea(); 
But the two method calls will perform two different actions

Coding exercise

- Let's try out the first coding exercise!
- Assume that you need to create a class **Animal** that has a two methods **eat()** and **makeSound()** and the subclasses **Dog** and **Cat**. Implement the three classes and necessary methods using the concept of the abstract classes. The implemented classes should be capable of creating the given program and generating the given output.

Program

```
Animal dog = new Dog();  
  
dog.eat();  
  
dog.makeSound();
```

```
Animal cat = new Cat();  
  
cat.eat();  
  
cat.makeSound();
```

Output

I am eating

Woof woof

I am eating

Meaw meaw

Sample answer

```
public abstract class Animal {  
    public void eat() {  
        System.out.println("I am eating");  
    }  
  
    abstract void makeSound();  
}
```

```
public class Dog extends Animal{  
  
    void makeSound() {  
        System.out.println("Woof woof");  
    }  
}
```

```
public class Cat extends Animal{  
  
    void makeSound() {  
        System.out.println("Meow Meow");  
    }  
}
```

Sample Answer

```
public class AnimalMain {  
    public static void main(String[] args) {  
        Animal dog = new Dog();  
        dog.eat();  
        dog.makeSound();  
  
        Animal cat = new Cat();  
        cat.eat();  
        cat.makeSound();  
  
    }  
}
```

I am eating
Woof woof
I am eating
Meow Meow

Interfaces

- Is a contract between a class and the outside world.
- When a class implements an interface it promises to provide the behavior in the interface (Implement the methods specified in the interface).
 - Similar to abstract class but all methods are abstract
- **Interfaces can also store constants.** These are essentially public static final variables.

Interfaces

```
interface Callback {  
    void callback(int param);  
}  
  
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
    void nonIfaceMeth() {  
        System.out.println("Classes that implement interfaces " +  
            "may also define other members, too.");  
    }  
}
```

Interface.java

Interfaces

- Are fully abstract classes.
- Similar to classes, but
 - All the methods in an interface are defined with out a body.
- Methods are automatically abstract
- Cannot contain class variables. But can contain constants (final variables)

```
interface Inter1 {  
    int COUNT = 10;  
    void test1();  
    void test2();  
}
```

Interfaces

```
class Test implements Inter1 {  
    public void test1() {  
    }  
    public void test2() {  
    }  
}
```

Must Implement all methods in Interface (otherwise should be defined as an abstract class)

Access level must be public

Coding exercise

- Let's try a coding exercise!
- Create the Printable interface. Create two classes Employee and Book with suitable attributes. Have a constructor to assign attributes.
- Implement the printable interface in the Employee and Book class.
- Create MyMain class with a main method and create objects of the Book and Employee classes.
- Print the details of the book and the employee.

```
interface Printable {  
    void print();  
}
```

Sample answer

```
public interface Printable {  
    void print();  
}
```

```
public class Book implements Printable{  
    private int id;  
    private String name;  
    private String author;  
  
    public Book(int id, String name, String author) {  
        this.id = id;  
        this.name = name;  
        this.author = author;  
    }  
  
    public void print() {  
        System.out.println("Id: "+id);  
        System.out.println("Name: "+name);  
        System.out.println("Author: "+author);  
        System.out.println();  
    }  
}
```

```
public class Employee implements Printable{  
    private int id;  
    private String name;  
    private String address;  
  
    public Employee(int id, String name, String address) {  
        this.id = id;  
        this.name = name;  
        this.address = address;  
    }  
  
    public void print() {  
        System.out.println("Id: "+id);  
        System.out.println("Name: "+name);  
        System.out.println("Address: "+address);  
        System.out.println();  
    }  
}
```

Sample answer

```
public class MyMain {  
  
    public static void main(String[] args) {  
        Printable printable = new Employee(1, "Amal", "Matara");  
        printable.print();  
  
        printable = new Book(1, "Amra Yahaluwo", "T.B. Ilangaratne");  
        printable.print();  
    }  
}
```

Id: 1
Name: Amal
Address: Matara

Id: 1
Name: Amra Yahaluwo
Author: T.B. Ilangaratne

Interfaces

- A Class can implement any number of interfaces.

```
class Test implements Inter1, Inter2, Inter3 {
```

- The class MUST override all the methods mentioned in all the interfaces.
- If the class did not override at least one method of the interfaces then the class become abstract.

Interfaces

- Java do not allow multiple inheritance.

X

```
class PartTimeStudent extends Employee, Student {
```

- Classes can extend exactly one class and implement any number of interfaces.

✓

```
class PartTimeStudent implements Employee, Student {
```

- Using interfaces we can achieve some thing similar to multiple inheritance.
- An interface can extends another interface.

Interfaces

- Interfaces are designed to support dynamic method resolution at run time.
- it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Abstract classes vs Interfaces

- Use an abstract class if
 - You want to share the code.
 - Expect to have common methods or properties
 - You want to have access modifiers other than public
 - You want to have properties which not static or not final
- Use an interface if
 - You want Unrelated classes to implement the interfaces.
 - Want to take advantages of multiple inheritances

Data access control

- **Member access modifiers** – control access to class members through the keywords **public**, **protected** , **default (friendly, no specifier)** and **private**
- **public** – members declared as public are accessible from anywhere in the program when the object is referenced.
- **protected** – subclasses can access protected method/data from the parent class.
- **Default** – such a class, method, or field can be accessed by a class inside the same package only.
- **private** – members declared as private are accessible *ONLY* to methods of the class in which they are defined. All private data are always accessible through the methods of their own class.
- **USUALLY:** data (**instance variables**) are declared **private**, methods are declared **public**.
- **NOTE:** using **public data** is uncommon and **dangerous** practice

Controlling access – Class member

Member Restriction	this	Subclass	Package	General
public	✓	✓	✓	✓
protected	✓	✓	✓	—
default	✓	—	✓	—
private	✓	—	—	—

Here a member is either a property or a method.

Java Packages

- *Packages* are containers for classes. They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Java Packages

- The Java Package name consists of words separated by periods. The first part of the name represents the organization which created the package. The remaining words of the Java Package name reflect the contents of the package. The Java Package name also reflects its directory structure.
- Importing a Class from an Existing Java Package Through the import statement.
* - all classes
- Example `import java.io.*;`
`import java.awt.Font;`
- CANNOT import a package, only a class:
`import java.io; - wrong!`

Default Packages

- If a class is in the same package as the class that uses it, it does not need **import** statement.
- **Default package** - all the complied classes in the current directory. If a package is not specified, the class is placed in the default package.

Java Standard Packages

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.) . This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).
java.net	Contains classes for supporting networking operations.

Errors and Exceptions

Object Oriented Programming (OOP)

Year 2 – Semester 1

Introduction

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. **Errors** are the mistakes that can make a program go wrong.

An error may **produce an incorrect output** or may **terminate the execution of the program abruptly** or even may **cause the system to crash**. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

Types of Errors

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile - Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile time errors. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Example : Compile Time Errors

- /* This program contains an error */

```
public class Error1 {  
  
    public static void main(String args[]) {  
        System.out.println("Hello Java!") // Missing ;  
    }  
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in the above Program, the following message will be displayed in the screen:

```
Error1.java :5: ';' expected System.out.println  
("Hello Java! ")
```

1 error

Most of the compile-time errors are due to typing mistakes.
The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments, initializations.
- Bad references to objects
- Use of = in place of == operator

Other errors we may encounter are related to directory paths. An error such as

javac: command not found

means that we have not set the path correctly. We must ensure that the path includes the directory where the Java executables are stored. Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string

Example : Run-Time Errors

```
class Error2 {  
    public static void main (String args[ ]) {  
        int a = 10;  
        int b = 5;  
        int c = 5;  
        int x = a / (b-c) ; // Division by zero  
  
        System.out.println("x = " + x) ; int y = a / (b+c);  
        System.out.println("y = " + y);  
    }  
}
```

Error2.java

Program is syntactically correct and therefore does not cause any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

java.lang.ArithmaticException: / by zero at Error2.main(Error2.java:8)

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

Coding Exercise

- Create a simple java program that will generate the run time exception for Converting invalid string to an Integer
- You can use `Integer.parseInt(String s)` method for the conversion

Sample Answer

```
public class EX1 {  
    public static void main(String[] args) {  
        String number = "123a";  
        int value = Integer.parseInt(number);  
    }  
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "123a"  
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
at java.lang.Integer.parseInt(Integer.java:580)  
at java.lang.Integer.parseInt(Integer.java:615)  
at EX1.main(EX1.java:5)
```

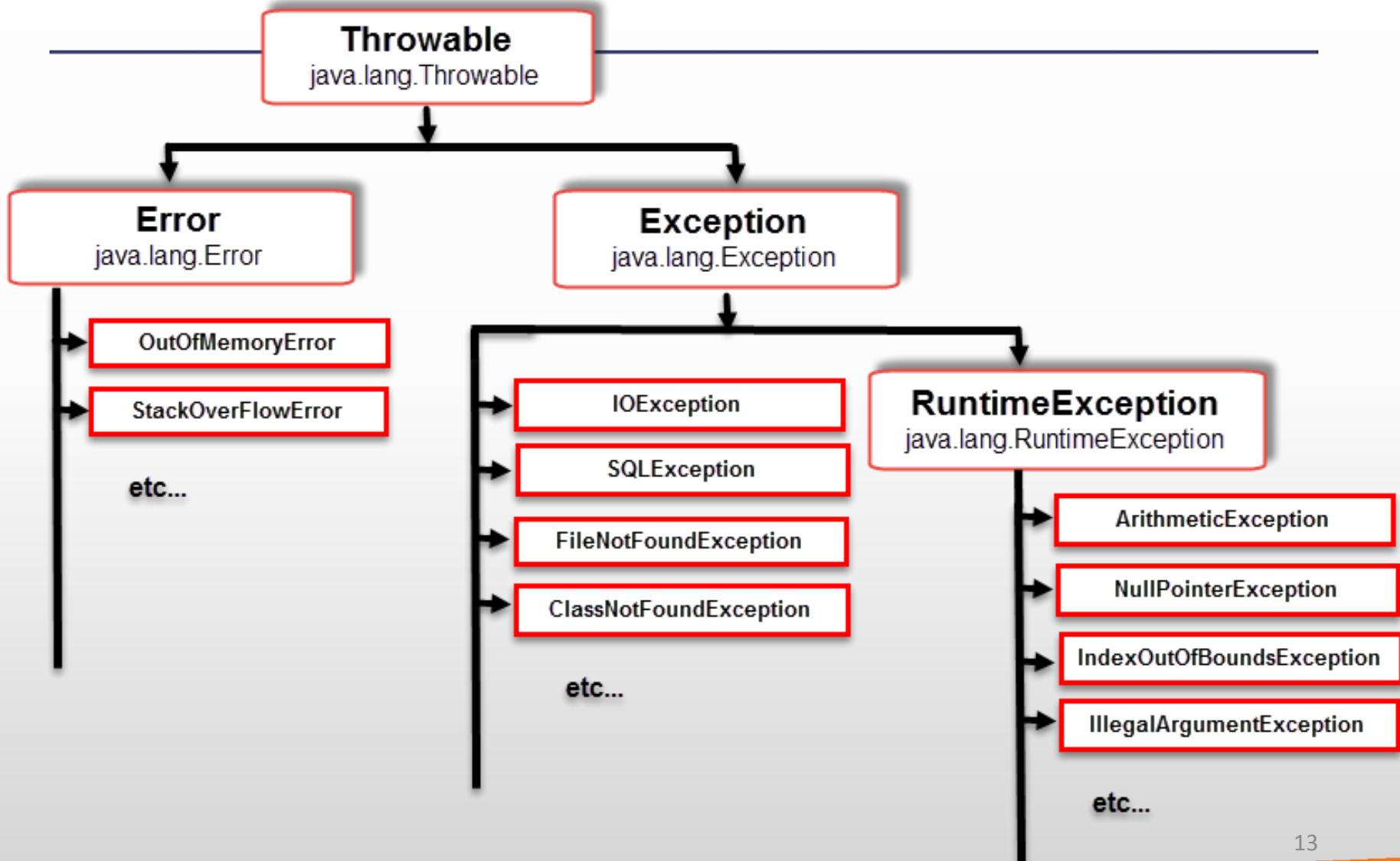
What is an Exception?

An exception is an *abnormal event* that arises during the execution (run time) of the program and disrupts the normal flow of the program.

Java classifies exceptions as

- ❑ Errors
- ❑ Exceptions
 - Checked Exceptions
 - Unchecked Exceptions

Hierarchy of Java Exception Classes



Error Class

- An Error is a subclass of Throwable that indicates **serious problems**.
- Errors represent critical errors, once occurs the system is not expected to recover from (irrecoverable).
- Errors can be generated from mistake in program logic or design.
- **Most applications should not try to handle it.**

e.g. OutOfMemoryError, VirtualMachineError, AssertionError

Exception Class

This class represents the exceptional conditions that user must handle or catch.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Checked Exceptions

- Checked Exception in Java is all those Exception which requires being catches and handled during compile time.
- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions

e.g.IOException, SQLException etc. Checked

Unchecked Exceptions

- Unchecked exceptions are usually caused by incorrect program code or logic such as invalid parameters passed to a method.
- The classes that extend RuntimeException are known as unchecked exceptions.
- Unchecked exceptions are checked at runtime.

e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException

Exception Handling

There are two ways to handle Exceptions:

- **Throw out and ignore**

When an Exception occur inside a code of program simply throw it out and ignore that an Exception occur declare the methods as **throws** Exception.

- **Catch and handle**

If a code segment is generating an Exception place it inside a try block. Then mention the way to handle the Exception inside a catch block use **try-catch** blocks

Exception Handling Using try and catch

1. Find the problem (Hit the exception).
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

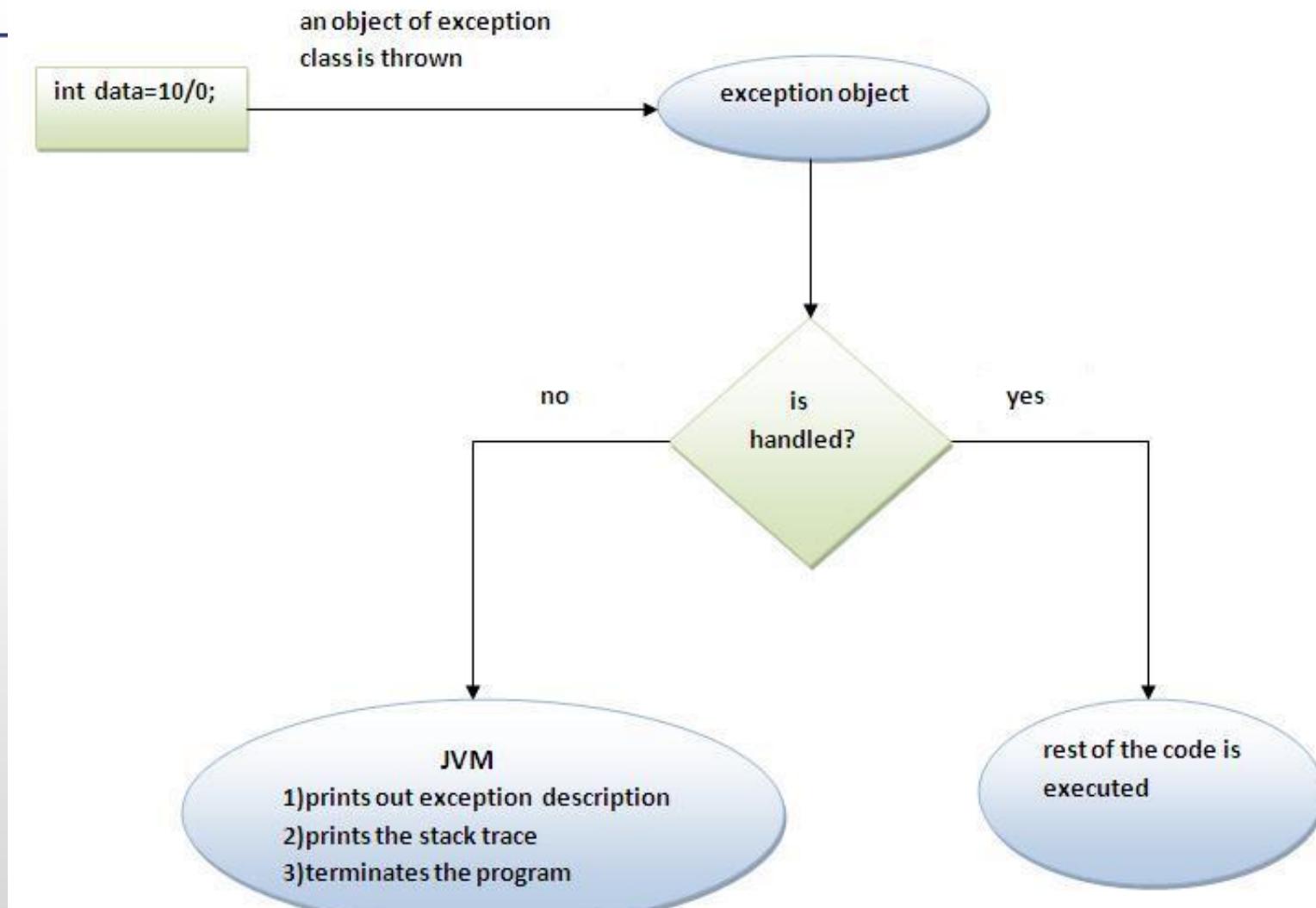
Provides two benefits.

- Allows you to fix the error.
- Prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch.
- Syntax:

```
try {  
    statement; // generates an exception  
  
} catch ( Exception-type e ) {  
    statement; // processes the exception  
}
```

Internal Working of Java try-catch Block



Example: try and catch

```
class Exc2 {  
  
    public static void main(String args[]) {  
        int d, a;  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmaticException e) { // catch divide-by-zero error  
            System.out.println("Exception: " + e);  
            System.out.println("Division by zero.");  
        }  
    }  
}
```

Exception.java

Note:

The call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed.

Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch** mechanism.

Coding Exercise

- You can now catch handle the exception that you generated in the earlier exercise, by displaying a meaningful message.

Sample answer

```
public class EX1 {  
    public static void main(String[] args) {  
        try {  
            String number = "123a";  
            int value = Integer.parseInt(number);  
        } catch (NumberFormatException ex) {  
            System.out.println("Wrong number format");  
        }  
    }  
}
```

Multiple catch clauses

- More than one exception could be raised by a single piece of code.
- To handle this type of situation, programmer can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try /catch** block.

```
try {  
    statement; // generates an exception  
  
} catch ( Exception1-type e ) {  
    statement; // processes the exception  
} catch ( Exception2-type e ) {  
    statement; // processes the exception  
}
```

Example: Multiple catch clauses

```
class MultipleCatches {  
  
    public static void main(String args[]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[] = {1};  
            c[42] = 99;  
        } catch (ArithmetricException e) {  
            System.out.println("Divide by 0: " + e);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index oob: " + e);  
        }  
        System.out.println("After try/catch blocks.");  
    }  
}
```

MultiExceptionCatch.java

Catching More Than One Type of Exception with One Exception Handler

- In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.
- In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException | SQLException ex) {  
    System.out.println(ex);  
}
```

Note:

If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter ex is final and therefore cannot assign any values to it within the catch block.

Nested try Statements

-
- ❑ When a try catch block is present inside another try block then it is called the nested try catch block.
 - ❑ Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.
 - ❑ If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don't handle exception.

Syntax:

```
//Main try block
try {
    statement 1;
    statement 2;
    //try-catch block inside another try block
    try {
        statement 3;
        statement 4;
        //try-catch block inside nested try block
        try {
            statement 5;
            statement 6;
        } catch(Exception e2) {
            //Exception Message
        }
    } catch(Exception e1) {
        //Exception Message
    }
} catch(Exception e3) { //Catch of Main(parent) try block
    //Exception Message
}
```

Example: Nested try Statements

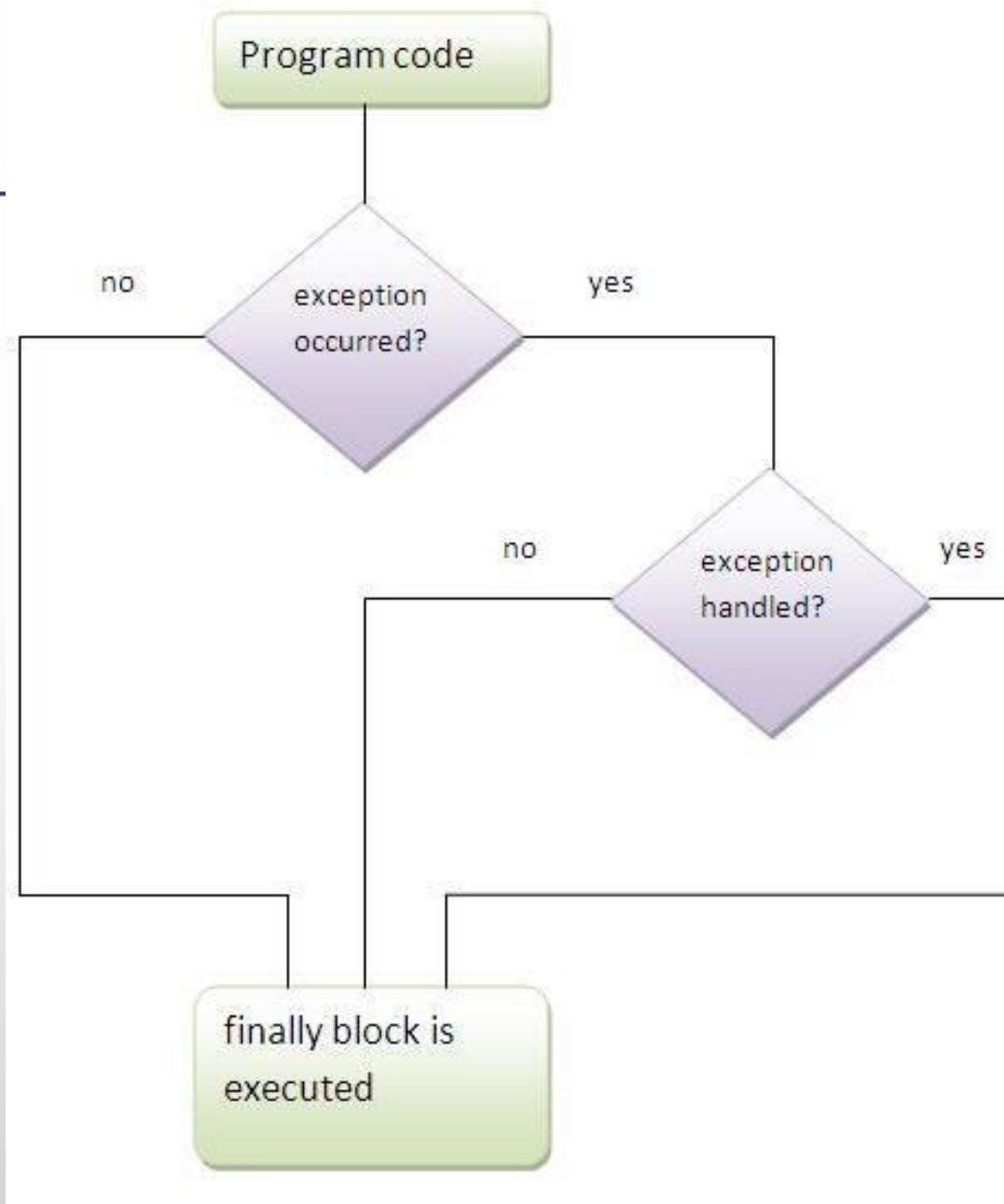
```
public class NestingDemo {  
  
    public static void main(String[] args) {  
        try { //try-block2  
            try { //try-block3  
                try {  
                    int arr[] = {1, 2, 3, 4};  
                    System.out.println(arr[10]);  
                } catch (ArithmaticException e) {  
                    System.out.print("Arithmatic Exception");  
                    System.out.println(" handled in try-block3");  
                }  
            } catch (ArithmaticException e) {  
                System.out.print("Arithmatic Exception");  
                System.out.println(" handled in try-block2");  
            }  
        } catch (ArithmaticException e3) {  
            System.out.print("Arithmatic Exception");  
            System.out.println(" handled in main try-block");  
        } catch (ArrayIndexOutOfBoundsException e4) {  
            System.out.print("ArrayIndexOutOfBoundsException");  
            System.out.println(" handled in main try-block");  
        } catch (Exception e5) {  
            System.out.print("Exception");  
            System.out.println(" handled in main try-block");  
        }  
    }  
}
```

Finally Block

- **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.
- **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows
- When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources
- Syntax:

```
try {  
    statement ;  
} finally{  
}
```

```
try {  
    statement ;  
} catch ( Exception1-type e ) {  
    statement ;  
} finally{  
}
```



Example: finally block

```
class TestFinallyBlock{  
  
    public static void main(String args[]){  
  
        try{  
  
            int data=25/0;  
  
            System.out.println(data);  
  
        } catch(ArithmaticException e){  
  
            System.out.println(e);  
  
        } finally{  
  
            System.out.println("finally block is always executed");  
  
            System.out.println("rest of the code...");  
  
        }  
  
    }  
  
}
```

Coding Exercise

- Add a finally block to the exception that you handled earlier, display the message “end of the operation”

Sample answer

```
public class EX1 {  
    public static void main(String[] args) {  
        try {  
            String number = "123a";  
            int value = Integer.parseInt(number);  
        } catch (NumberFormatException ex) {  
            System.out.println("Wrong number format");  
        }finally {  
            System.out.println("End of operation");  
        }  
    }  
}
```

Exception Handling Using throws

- When an Exception occur inside a code of program simply throw it out and ignore.
- **throws** keyword is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.
- A **throws** clause lists the types of exceptions that a method might throw.
- Syntax

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Example: throws

```
import java.io.*;  
  
class Error03{  
    public static void main(String[] a) throws IOException{  
        BufferedReader in = new BufferedReader( new InputStreamReader (System.in));  
        String text=""; System.out.print("Enter an integer value : ");  
        text = in.readLine(); int num = Integer.parseInt(text);  
        System.out.println("You inserted "+num);  
    }  
}
```

IOError.java

Coding Exercise

- Create a class MathOp with the two methods to add and divide two integers.
- Write a class with a main method, call the two methods with following parameters

```
MathOp math = new MathOp();
math.add(2, 0);
math.divide(2, 0);
```

Sample answer

```
public class MathOp {  
  
    public int add(int a, int b) {  
        return a+b;  
    }
```

```
    public double divide(int a, int b)  
{  
        return a/b;  
    }  
}
```

```
public class EX2 {  
    public static void main(String[] args)  
    {  
        MathOp math = new MathOp();  
        math.add(2, 0);  
        math.divide(2, 0);  
    }  
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at MathOp.divide(MathOp.java:9)  
at EX2.main(EX2.java:6)
```

Coding Exercise

- Throw out the ArithmeticException from the divide method.
- Catch it and handle with a try catch block in the main method.

Sample answer

```
public class MathOp {  
  
    public int add(int a, int b) {  
        return a+b;  
    }  
  
    public double divide(int a, int b) throws ArithmeticException {  
        return a/b;  
    }  
}
```

```
public class EX2 {  
    public static void main(String[] args) {  
        MathOp math = new MathOp();  
        math.add(2, 0);  
        try {  
            math.divide(2, 0);  
        } catch (ArithmetricException ex) {  
            System.out.println("Error: division by zero");  
        }  
    }  
}
```

Creating Your Own Exception Subclasses

- There may be times when we would like to throw our own exceptions. We can do this by using the keyword `throw` as follows:

```
throw new Throwable-subclass;
```

Examples: `throw new ArithmeticException();`
`throw new NumberFormatException();`

- To define your own exception create a subclass of **Exception** (which is a subclass of **Throwable**).
- The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

-
- ❑ **Exception** class defines four public constructors. Two are shown here:
 - ❖ `Exception()`
 - ❖ `Exception(String msg)`

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Example

```
class MyException extends Exception {  
    MyException (String message) {  
        super(message);  
    }  
}  
  
class TestMyException {  
    public static void main (String args[ ]) {  
        int x = 5, y = 1000;  
  
        try {  
            float z = (float) x / (float) y;  
            if(z < 0.01) {  
                throw new MyException ("Number is too small");  
            }  
        } catch (MyException e) {  
            System.out.println ("Caught my exception");  
            System.out.println(e.getMessage());  
        } finally {  
            System.out.println ("I am always here");  
        }  
    }  
}
```

MyException.java

Commonly Used Exceptions

Exception	Meaning
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotFoundException	Type not found.

Reference

The Complete Reference 8th edition

Strings in Java

Lecture – 5A

Contents

- Introduction to String
- String manipulation
- StringBuffer
- StringBuilder

Strings in Java

- String is a sequence of characters
- Java implements strings as objects (created by class String)
- String, StringBuffer and StringBuilder classes are defined in java.lang package. Thus, are available to a program automatically
- All String, StringBuffer and StringBuilder classes are final
- String objects are *immutable*
- StringBuffer and StringBuilder objects are mutable

Common ways of creating String objects

- String class have many constructors

//method 1

```
char arr[] = {'a', 'b', 'c'};  
String s = new String(arr);
```

//method 2

```
String s = new String("abc");
```

//method 3

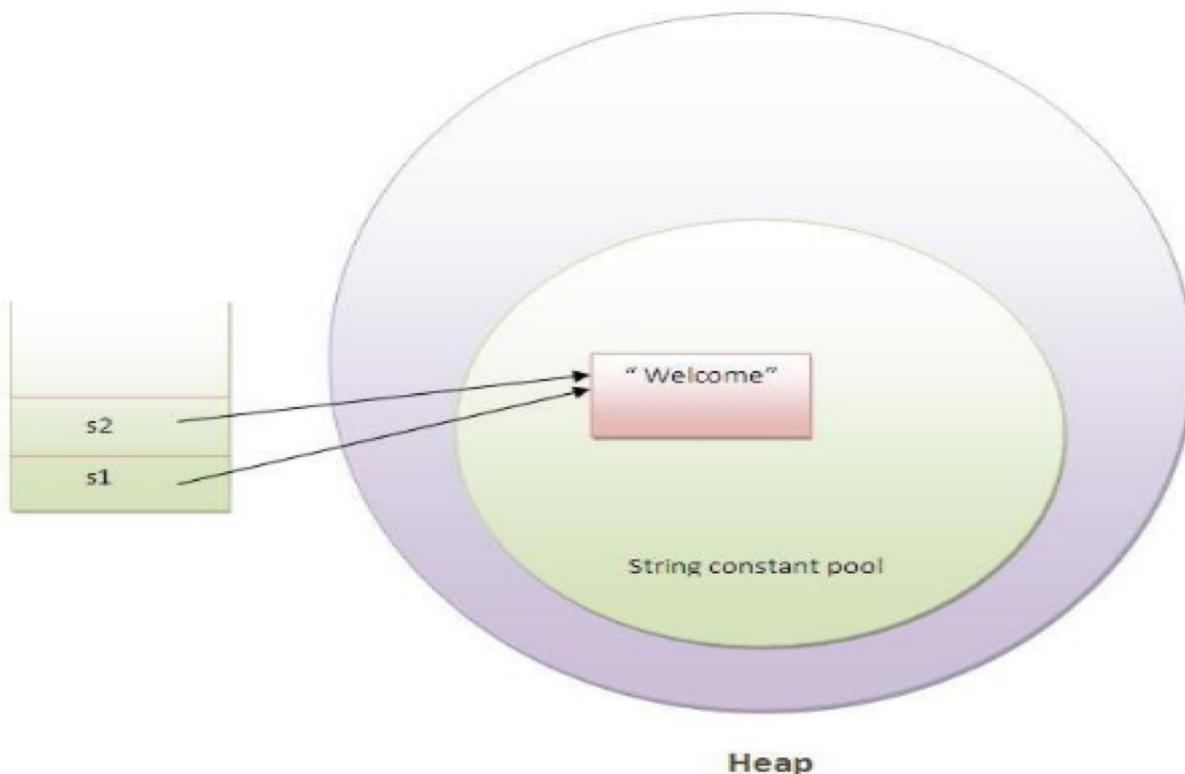
```
String s = "abc";
```

Creation of String Literals

- Each time you create a string literal, the JVM checks the string constant pool first
- If the string already exists in the pool, a reference to the pooled instance is returned
- If string doesn't exist in the pool, a new string instance is created and placed in the pool

Creation of String Literals cont.

```
String s1="Welcome";  
String s2="Welcome";//will not create new instance
```



Exercise 1

Draw the String pool for the below code

- a) String first = "Tooth";
 first = "Tooth" + " Fairy";
 String second =first +5;
- b) String first = "Tooth";
 first = "tooth" +"Fairy";
 String second =first +5;

String Literal & String Object

```
String str1 = "Hello World!!";  
String str2 = "Hello World!!";  
System.out.println(str1 == str2); // true
```

- When the String literal str2 is created, the string “Hello World” is not created again. Instead, it is str1 String is reused as it is already existing in the string constant pool.
- Since both str1 and str2 are referring to the same String in the pool, str1 == str2 is true.

String Literal & String Object cont.

```
String str3 = new String("Hello World!!");  
String str4 = new String("Hello World!!");  
System.out.println(str3 == str4); // false
```

- In this case, new String objects are created and separately referred by str3 and str4. Thus, str3 == str4 is false.

equals() versus ==

- Both `equals()` and `==` operator performs different operations
- `equals()` is a method that compares the characters in a string object
- `==` is an operator that compares two object references to see whether they refer to the same instance

Strings in Java are Immutable

- Strings are **immutable**. That is, once a String is constructed, its contents cannot be modified
- However, the variable declared as **String** reference can be changed to point at some other **String** instance
- It is not efficient to use **String** if you need to modify your string frequently (that would create many new Strings occupying new storage areas)

Exercise 2

Consider the below code block and state each statement return true or false

```
String s1= "Java";
String s2= "java";
String s3=new String("java");
String s4=new String("Java");
String s5=s4;
```

- 1) System.out.println (s1 == s2);
- 2) System.out.println (s1.equals(s2));
- 3) System.out.println (s1.equals("Java"));
- 4) System.out.println (s3.equals(s4));
- 5) System.out.println (s5.equals(s4));
- 6) System.out.println (s5 == s4);

String Operations

- `length()`
- `concat()`
- `toUpperCase()`
- `toLowerCase()`
- `charAt()`
- `indexOf()`
- `lastIndexOf()`
- `substring()`
- `replace()`
- `toCharArray()`
- `startsWith()`
- `endsWith()`
- `trim()`
- `split()`
- `equals()`
- `equalsIgnoreCase()`
- `join()` — new addition to JDK8

Exercise 3

Make a pair.

Go through the given String manipulation methods.

Teacher will assign a method.

Explain to other by using a suitable example

Question

What is the out put of the following code?

```
1 public class MyClass {  
2     public static void main(String[] args) {  
3         int age = 25;  
4         String s1 = "He is "+ age +" years old.";  
5         System.out.println(s1);  
6         String s2 = "Value of x = "+ 2 + 2;  
7         System.out.println(s2);  
8     }  
9 }
```

StringBuffer & StringBuilder

- As strings are immutable, Java provides two other classes to support mutable strings:
 - `StringBuffer`
 - `StringBuilder`

*both classes in `java.lang` package

- A `StringBuffer` or `StringBuilder` object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side-effect to other objects

StringBuffer, String Builder Operations

- `length()`
- `indexOf()`
- `lastIndexOf()`
- `charAt()`
- `replace()`
- `substring()`
- `getChars()`
- `append()`
- `insert()`
- `reverse()`
- `deleteCharAt()`

StringBuffer

```
1 public class StringBufferDemo {  
2     public static void main(String[] args) {  
3  
4         StringBuffer sb = new StringBuffer("Java StringBuffer Reverse Example");  
5         System.out.println("Original StringBuffer Content : " + sb);  
6         sb.reverse();  
7         System.out.println("Reversed StringBuffer Content : " + sb);  
8  
9     }  
10 }
```

Original StringBuffer Content : Java StringBuffer Reverse Example
Reversed StringBuffer Content : elpmaxE esreveR reffuBgnirts avaJ

StringBufferDemo0.java

StringBuilder

- Introduced in JDK5
 - `StringBuilder` is similar to `StringBuffer` except for one difference that it is not synchronized (not thread-safe)
- *In cases in which a mutable string is accessed by multiple threads, and no external synchronization is employed, you must use `StringBuffer` rather than `StringBuilder`

StringBuilder cont.

```
1 public class StringBuilderDemo {  
2  
3     public static void main(String[] args) {  
4         StringBuilder builder = new StringBuilder();  
5  
6         for (int i = 0; i < 5; i++) {  
7             builder.append("abc ");  
8         }  
9  
10        System.out.println(builder);  
11    }  
12 }
```

abc abc abc abc abc

StringBuilderDemo0.java

StringBuilder cont.

```
1 public class StringBuilderDemo1 {  
2  
3     public static void main(String[] args) {  
4         StringBuilder builder = new StringBuilder("abc");  
5         builder.insert(2, "xyz");  
6         System.out.println(builder);  
7     }  
8 }
```



abxyzc

StringBuilderDemo1.java

Java Collection Framework

Object Oriented Programming (OOP)

Year 2 – Semester 1

Collections in Java

The **Collections Framework** is a sophisticated hierarchy of interfaces and classes that provide state-of-art technology for managing groups of objects.

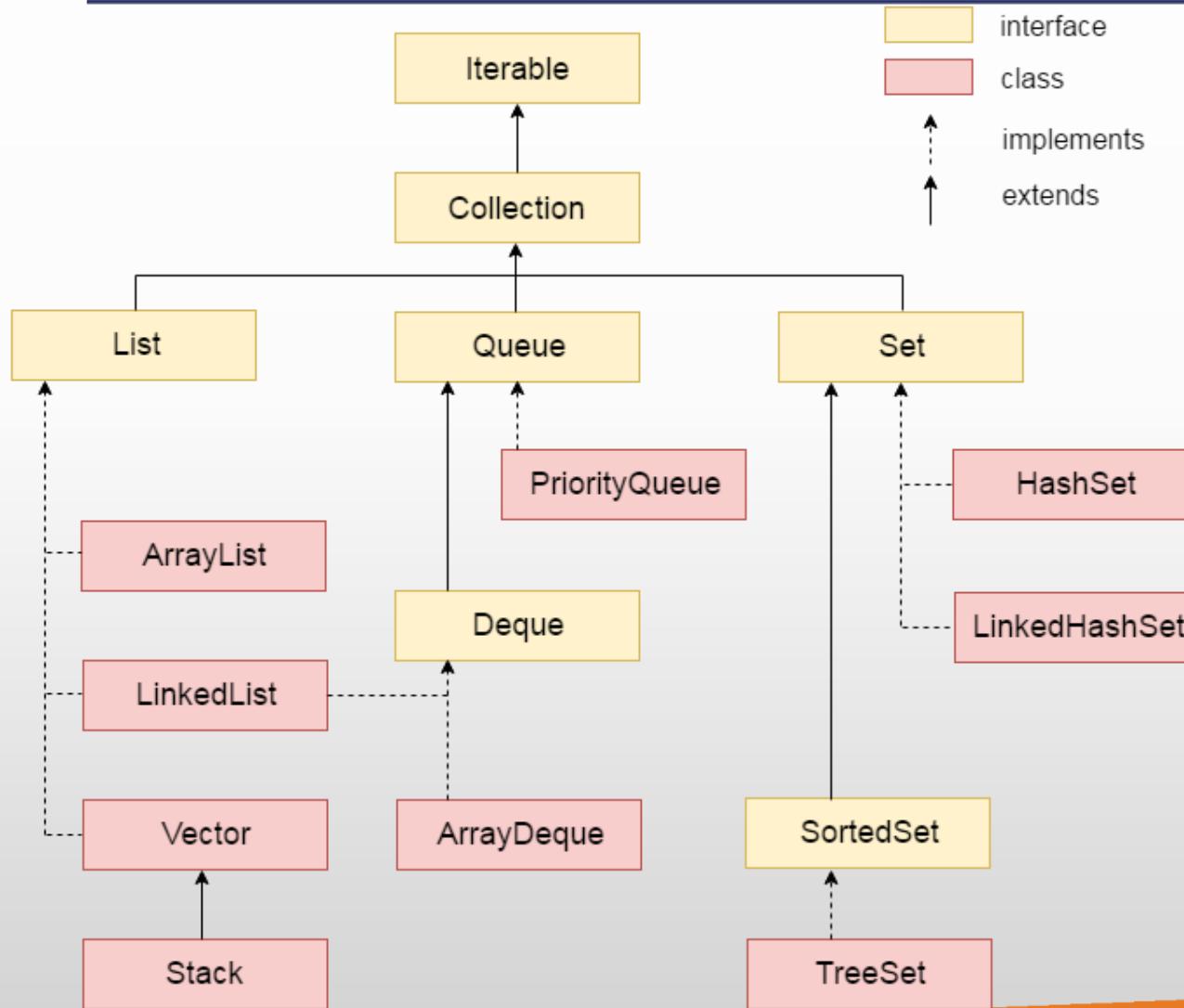
Java Reference Book 09: Page 497

The primary advantages of a collections framework are;

- Reduces programming effort
- Increases performance
- Provides interoperability between unrelated APIs
- Reduces the effort required to learn APIs
- Reduces the effort required to design and implement APIs
- Fosters software reuse

Generics make Collections type safe. Before generics, collections stored Object references; can store any type of object. Thus avoids run-time mismatch errors.

Hierarchy of Collections Framework



The **java.util** package contains all the classes and interfaces for Collection framework

Collection Interface

- Collection is the foundation upon which the Collection Framework is built on.

Method	Description
public boolean add(Object element)	is used to insert an element in this collection.
public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	is used to delete an element from this collection.
public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no of element from the collection.
public boolean contains(Object element)	is used to search an element.
public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator.
public Object[] toArray()	converts collection into array.
public boolean isEmpty()	checks if collection is empty.
public boolean equals(Object element)	matches two collection.
public int hashCode()	returns the hashcode number for collection.

List Interface

- ❑ Java.util.List is a child interface of Collection.
- ❑ List is an **ordered** collection of objects in which **duplicate values can be stored**. Since List preserves the insertion order it allows positional access and insertion of elements.
- ❑ List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

Queue Interface

- ❑ The java.util.Queue is a subtype of java.util.Collection interface.
- ❑ It is an ordered list of objects with its use limited to **inserting elements at the end of list and deleting elements from the start of list. It follows First In First Out (FIFO) principle.**
- ❑ Queue Interface is implemented by PriorityQueue class

Problem

Think of a School management application. You have a list of students stored in Database. To display those names on the user interface, You have to make a call to the database and store these elements and populate those list elements on UI.

What is the best approach you are suggesting?

Answer – An Array?

```
String arr[]=new String[5];
arr[0] = "Anne";
arr[1] = "Peter";
arr[2] = "Shenan";
arr[3] = "Pete";
arr[4] = "Diana";
```

Problems?

- Arrays are of fixed length. You can not change the size of the arrays once they are created.
- You can not accommodate an extra element in an array after they are created.
- Memory is allocated to an array during its creation only, much before the actual elements are added to it.

Answer – ArrayList Class

- ArrayList class extends the AbstractList class and implements the List interface.
- ArrayList support dynamic arrays (can increase and decrease size dynamically).
- Elements can be inserted at or deleted from a particular position
- ArrayList class has many methods to manipulate the stored objects
- If generics are not used, ArrayList can hold any type of objects.

Example : ArrayList

Ref: Java Complete Reference Pg: 512

```
import java.util.*;  
  
class ArrayListDemo {  
    public static void main(String args[]) {  
        // Create an array list.  
        ArrayList<String> al = new ArrayList<String>();  
  
        System.out.println("Initial size of al: " +  
                           al.size());  
  
        // Add elements to the array list.  
        al.add("C");  
        al.add("A");  
        al.add("E");  
        al.add("B");  
        al.add("D");  
        al.add("F");  
        al.add(1, "A2");  
  
        System.out.println("Size of al after additions: " +  
                           al.size());  
  
        // Display the array list.  
        System.out.println("Contents of al: " + al);  
  
        // Remove elements from the array list.  
        al.remove("F");  
        al.remove(2);  
  
        System.out.println("Size of al after deletions: " +  
                           al.size());  
  
        System.out.println("Contents of al: " + al);  
    }  
}
```

ArrayListDemo.java

Obtaining Array from an ArrayList

Reasons for converting array to an ArrayList

- To obtain faster processing time for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that don't understand collections

Example : ArrayList to Array

Ref: Java Complete Reference Pg: 514

```
import java.util.*;  
  
class ArrayListToArray {  
    public static void main(String args[]) { // Create an array list.  
        ArrayList<Integer> al = new ArrayList<Integer>();  
  
        // Add elements to the array list.  
        al.add(1);  
        al.add(2);  
        al.add(3);  
        al.add(4);  
  
        System.out.println("Contents of al: " + al);  
  
        // Get the array.  
        Integer ia[] = new Integer[al.size()];  
        ia = al.toArray(ia);  
  
        int sum = 0;  
  
        // Sum the array.  
        for(int i = 0 ; i < ia.length ; i++)  
            sum += i;  
  
        System.out.println("Sum is: " + sum);  
    }  
}
```

ArrayListToArray.java

Example : forEach Loop for Iterating Over Collections

```
import java.util.ArrayList;

public class ForEachExample {

    public static void main(String[] args) {
        ArrayList<String> items = new ArrayList<>();
        items.add("A");
        items.add("B");
        items.add("C");
        items.add("D");
        items.add("E");

        for (String item : items) {
            System.out.println(item);
        }
    }
}
```

ForEachExample.java

Problem

Think of a scenario where plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.

What is the best approach you are suggesting?

Answer – An ArrayList?

Problems?

- In ArrayList , Elements can be inserted at or deleted from a particular position
- What will happen if we remove one middle plate from the stack?

Answer – Stack Class

- Stack is a subclass of **Vector**
- Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle

Exercise 2

Problem

Think of a scenario of a Bank line where people who come first will done his transaction first.

What is the best approach you are suggesting?

Answer – Queue Interface

- The **Queue** interface extends **Collection**
- Defines queue data structure which is normally **First-In-First-Out**
- Elements are added from one end and elements are deleted from another end
- Most common classes are the [PriorityQueue](#) and [LinkedList](#) in Java
- In Priority Queue, elements are removed from one end, but elements are added according to the order defined by the supplied comparator.

Exercise 3

Problem

Think of an application that require a unique User ID as a input

What is the best approach you are suggesting?

Answer -Set Interface

- The `java.util.Set` interface is a subtype of `Collection` interface.
- A **Set** is a **Collection that cannot contain duplicate elements**. It models the mathematical set abstraction.
- Set interface is implemented by `SortedSet` interface and `HashSet` and `LinkedHashSet` classes.
- `SortedSet` interface declares the behavior of a set sorted in ascending order.
- `TreeSet` class implements this interface and `TreeSet` class are stored in ascending order.
- `HashSet` class stores the elements by using a mechanism called **hashing** and contains unique elements only

Problem

Think of a scenario where you ordering a 3-topping pizza.

Whether you say 'Pepperoni, mushrooms, and onions, please'

or 'Yoo hoo! Make that mushrooms, pepperoni, and onions'

What is the best approach you are suggesting?

Answer - HashSet Class

- HashSet extends AbstractSet and implements the Set interface.
- It creates a collection that uses a hash table for storage.
- No duplication and unordered
- In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically

Example : HashSet

Ref: Java Complete Reference Pg: 517

```
import java.util.HashSet;

public class HashSetDemo {

    public static void main(String args[]) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();

        // Add elements to the hash set.
        hs.add("Beta");
        hs.add("Alpha");
        hs.add("Eta");
        hs.add("Gamma");
        hs.add("Epsilon");
        hs.add("Omega");

        System.out.println(hs);
    }
}
```

HashSetDemo.java

Exercise 4 – Question 1

Problem

Think of a scenario where you store student details including ID and the marks.

What is the best approach you are suggesting?

Answer - TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface

- Objects are stored in sorted, ascending order

Exercise 4 – Question 2

Problem

Think of a cache, where in if new data comes in we overwrite the existing record using the key. So basically the cache would be used to store the most recent state.

What is the best approach you are suggesting?

Answer - Maps

- A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value.
- Both keys and values are objects. The keys must be unique, but the values may be duplicated.
- Some maps can accept a null key and null values, others cannot.
- Map is useful if you have to search, update or delete elements on the basis of key
- Maps are not part of the Collections Framework**

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterations.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

Ref: Java Complete Reference Pg: 537

HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map.

- HashMap is a generic class that has this declaration:
class HashMap<K, V>
K specifies the type of keys, and V specifies the type of values.

Example : HashMap

Ref: Java Complete Reference Pg: 537

```
public class HashMapDemo {  
    public static void main(String args[]) {  
        // Create a hash map.  
        HashMap<String, Double> hm = new HashMap<String, Double>();  
  
        // Put elements to the map  
        hm.put("John Doe", new Double(3434.34));  
        hm.put("Tom Smith", new Double(123.22));  
        hm.put("Jane Baker", new Double(1378.00));  
        hm.put("Tod Hall", new Double(99.22));  
        hm.put("Ralph Smith", new Double(-19.08));  
  
        // Get a set of the entries.  
        Set<Map.Entry<String, Double>> set = hm.entrySet();  
  
        // Display the set.  
        for (Map.Entry<String, Double> me : set) {  
            System.out.print(me.getKey() + ": ");  
            System.out.println(me.getValue());  
        }  
  
        // Deposit 1000 into John Doe's account.  
        double balance = hm.get("John Doe");  
        hm.put("John Doe", balance + 1000);  
  
        System.out.println("John Doe's new balance: " + hm.get("John Doe"));  
    } }
```

HashMapDemo.java

Hashmap

- Values in hash map are not ordered or not sorted
- Remove duplicates in the key

Exercise 5

LinkedHashMap

- HashMap with additional feature that it maintains **insertion order**
- contains values based on the key
- Remove duplicates in the key
- When Display, the order is not guaranteed

Exercise 6

TreeMap

- contains values based on the key
- contains only unique elements
- maintains ascending orders
- provides an efficient means of storing key-value pairs in sorted order

Exercise 7

Reference

- Java Complete Reference – 9th Edition

- Java T- Point:
<https://www.javatpoint.com/collections-in-java>

- Java Collections Tutorial - Jakob Jenkov
<http://tutorials.jenkov.com/java-collections/index.html>

Generics

Lecture – 7

Contents

- **Wrapper classes**
- **Introduction to Generics**
 - Generic classes - Declaration & Instantiation
 - Generic Methods – Implementation & Invocation
 - Bounded Type Parameters
 - Wildcards
 - Limitations in Generics

Wrapper class / Covering class

- Wrapper class in java provides the mechanism *to convert primitive datatype into object* and *object into primitive type*.
- For each primitive datatype, there exists a covering class in the *java.lang* package.

byte	→	Byte
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double
char	→	Character
Boolean	→	Boolean

Wrapper class cont.

- The automatic conversion of primitive type into object is known as *autoboxing* and vice-versa *unboxing*.
- Since J2SE 5.0, autoboxing and unboxing feature converts primitive into object and object into primitive *automatically*.

```
1 public class Test {  
2     public static void main(String args[]) {  
3         int a = 50;  
4  
5         Integer i =Integer.valueOf(a);    //converting an int into an Integer  
6         Integer j = a;      //auto boxing  
7  
8         System.out.println(a);  
9         System.out.println(i);  
10        System.out.println(j);  
11    }  
12 }  
13 }  
14 }
```

autoboxing.java

Wrapper class cont.

unboxing

```
1 public class Test {  
2     public static void main(String args[]) {  
3  
4         //Converting Integer to int  
5         Integer a=new Integer(3);  
6         int i=a.intValue(); //converting Integer to int  
7         int j=a;      //unboxing, now compiler will write a.intValue() internally  
8  
9         System.out.println(a);  
10        System.out.println(i);  
11        System.out.println(j);  
12  
13    }  
14 }  
15 }
```

unboxing.java

Generics in Java

- Concept of “generics” was introduced in JDK 5 to deal with type-safe objects.
- Introduction of generics has changed Java in 2 ways:
 - Added new syntax to Java language
 - Caused changes to many of the classes & methods in the core API
- With the use of generics, it is possible to create classes, interfaces & methods which works in type-safe manner.

Generics in Java cont.

- “Generics” means “parameterized types”.
- Many algorithms are same regardless of its data type that is applied. With generics, you can define the algorithm once independently of any specific type of data. Later, you can use the algorithm to a wide variety of data types without any additional effort.

Advantages of Generics

- 1. Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other types of objects.
- 2. Type casting is not required:** There is no need to typecast the object. All type conversions are implicit.
- 3. Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than at runtime.

Implementation

- When you are in need to store a list of values, you shall use an array.

e.g.: if you are storing marks of 5 students then you may create an array as shown in the code below

```
1
2 public class GenExample {
3     public static void main(String args[]){
4         int marks[] = new int[5];
5     }
6 }
```

Implementation cont.

Problem:

- *Size of the array is fixed.* If there is a change in the size of the array (in the number of elements that are stored), you have to modify the code manually.
- You may not be able to expand or shrink the array automatically as & when the element is being added.

Solution:

- Use Collection Interface

Implementation cont.

Shown below is an example of how Collection is used to store elements. Note that the list grows each & every time add() is called.

```
1 import java.util.ArrayList;
2 import java.util.Collection;
3
4 public class GenExample {
5     public static void main(String args[]){
6         Collection value = new ArrayList();
7         value.add(10);
8         value.add(20);
9         value.add(30);
10        value.add(40);
11        value.add(50);
12
13    }
14 }
```

GenericDemo2.java

Implementation cont.

Collection value = new ArrayList();

- This statement allows you to create a list of elements (list of marks).
- Note that Collection is an interface and cannot be instantiated directly.
- ArrayList is a class which implements the Interface List which extends the Interface Collection.

Implementation cont.

```
Collection value = new ArrayList();
```

Problem:

This list may contain any object as we have not specified the data type of the element to be added.

```
Collection value = new ArrayList();
value.add("SLIIT"); //String object
value.add(145); //integer
value.add(23.4578); //double
value.add(1.54f); //float
value.add('y'); //char
```

Implementation cont.

Collection value = new ArrayList();

Problem:

What if the requirement is to store only *integers*?

Solution:

Make use of **Generics**!

Collection <Integer> value = new ArrayList<>();

*Make use of Wrappers as primitive types do not support Generics.

Implementation cont.

Collection <Integer> value = new ArrayList<>();

- Now you may not be able to add elements other than Integer type. (Note the Compilation Error)

```
1 import java.util.ArrayList;
2 import java.util.Collection;
3
4 public class GenExample {
5     public static void main(String args[]){
6         Collection<Integer> value = new ArrayList<>();
7         value.add("SLIIT"); //String object
8         value.add(145); //integer
9         value.add(23.4578); //double
10        value.add(1.54f); //float
11        value.add('y'); //char
12
13    }
14 }
```

GenericDemo3.java

Question:

Write a program to store a list of names, retrieve the names & display on the screen.

Hint: Use an [ArrayList](#) class

Type Parameters

The type parameter naming conventions are as follows:

T - Type

E - Element

K - Key

N - Number

V - Value

Generic class

- A class that can refer to any type is known as generic class.
- Type T indicates that it can refer to any type (Integer, Double, Employee etc.)

```
1
2 public class MyGen<T> {
3     T obj;
4
5     void add(T val) {
6         this.obj = val;
7     }
8
9     T get() {
10        return obj;
11    }
12 }
```

test.java

Generic class cont.

- You may make use of the generic class as shown below:

```
1 public class Test {  
2     public static void main(String args[]) {  
3         MyGen<Integer> m= new MyGen<>();  
4         m.add(12);  
5         System.out.println(m.get());  
6     }  
7 }  
8 }
```

- Note that there will be a compilation error if you attempt to add any other type of data other than an integer.

test.java

Generic method

- Like generic class, we can create generic method that can accept any type of argument.

```
1 public class Test {  
2  
3     public static <E> void printArray(E[] elements) {  
4         for(E elem : elements) {  
5             System.out.print(elem+" ");  
6         }  
7         System.out.println();  
8     }  
9  
10    public static void main(String args[]) {  
11        Integer[] intArray = {12,15,45,78,92,56,72};  
12        Character[] charArray = {'S','L','I','I','T'};  
13  
14        printArray(intArray);  
15        printArray(charArray);  
16  
17    }  
18 }
```

12 15 45 78 92 56 72
S L I I T

test1.java

Multiple Parameters

- A generic class or method can have multiple type parameters.

```
1  class TwoGen<T,V>{
2      T ob1;
3      V ob2;
4
5      TwoGen(T o1, V o2){
6          ob1 = o1;
7          ob2 = o2;
8      }
9
10     void ShowType() {
11         System.out.println("Type of T is"+ob1.getClass().getName());
12         System.out.println("Type of V is"+ob2.getClass().getName());
13     }
14
15     T getob1() {
16         return ob1;
17     }
18
19     V getob2() {
20         return ob2;
21     }
22 }
```

test2.java

Multiple Parameters cont.

- This is how you will make use of the generic class you created:

```
25 public class GenExample {  
26     public static void main(String args[]){  
27         TwoGen<Integer, String> tg0b = new TwoGen<> (123, "Anne");  
28  
29         tg0b.ShowType();  
30  
31         int v = tg0b.getob1();  
32         System.out.println("Value is "+v);  
33  
34         String s = tg0b.getob2();  
35         System.out.println("Value is "+s);  
36     }  
37 }
```

Type of T isjava.lang.Integer
Type of V isjava.lang.String
Value is 123
Value is Anne

test2.java

Multiple Parameters cont.

- Although the two type argument differ in the example, it is possible for both types to be same.

e.g.:

```
TwoGen<String, String> tg0b = new TwoGen<> ("BM-1", "Anne");  
tg0b.ShowType();
```

```
String v = tg0b.getob1();  
System.out.println("Value is "+v);
```

```
Type of T is java.lang.String  
Type of V is java.lang.String  
Value is BM-1  
Value is Anne
```

```
String s = tg0b.getob2();  
System.out.println("Value is "+s);
```

- Also note, if both type arguments were always the same, then two type parameters would be unnecessary.

Bounded Type Parameters

- There may be times when you will want to *restrict* the kind of data types that are allowed to be passed to a type parameter.
e.g.: A method that operates on numbers might only want to accept instances of Number or its subclasses.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound in the following format.
`<T extends superclass>`
e.g.: `<T extends Number>`
- Note: T, can only be replaced by the superclass or its subclasses.

Bounded Type Parameters cont.

- This is an example of the class with a bounded type parameter:

```
1 public class Stats <T extends Number>{
2     T[] nums;
3
4     Stats(T[] ob){
5         nums = ob;
6     }
7     double average() {
8         double sum = 0.0;
9         for(int i= 0; i<nums.length; i++)
10             sum+= nums[i].doubleValue();
11         return sum /nums.length;
12     }
13 }
```

test3.java

Bounded Type Parameters cont.

- This is how you will make use of the class with a bounded type parameter.

```
1 public class BoundDemo {  
2     public static void main(String[] args) {  
3         Integer inums[] = {1,2,3,4,5};  
4         Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};  
5  
6         Stats<Integer> ob1 = new Stats<>(inums);  
7         Stats<Double> ob2 = new Stats<>(dnums);  
8  
9         double i = ob1.average();  
10        System.out.println("Average of Integer Array: "+i);  
11  
12        double d = ob2.average();  
13        System.out.println("Average of Double Array: "+d);  
14    }  
15 }
```

Average of Integer Array: 3.0
Average of Double Array: 3.3

test3.java

Bounded Type Parameters cont.

- Note that the following code segment will give compilation error as String is not a subclass of Number

```
String snum[] = {"1", "2", "3", "4", "5"};
Stats<String> ob3 = new Stats<>(snums);
```

Bounded Type Parameters cont.

- In addition to using a class type as a bound, you can also use an interface type too! In fact, you can specify multiple interfaces as bounds.
- A bound can include a class type & one or more interfaces. In this case, the class type needs to be specified first.
- When a bound include an interface type, only type arguments that implement that interface are legal.
- When specifying a bound that has a class & an interface or multiple interfaces, use the & operator to connect them

e.g:

```
1 public class MyClass <T extends TClass & TInterface> {  
2     //code  
3 }  
4  
5 class TClass{}  
6 interface TInterface{}
```

Wildcard in Java Generics

- The ? Symbol represents the wildcard element. It means any type can be matched with the ? symbol.
- If we write <? extends Number>, it means any child class of Number (e.g. Integer, Double, Float etc.) can be matched with the ?

```
1 abstract class Shape {  
2     abstract void draw();  
3 }  
4  
5 class Recangle extends Shape{  
6     void draw() {  
7         System.out.println("Drawing Rectangle");  
8     }  
9 }  
10  
11 class Circle extends Shape{  
12     void draw() {  
13         System.out.println("Drawing Circle");  
14     }  
15 }
```

test4.java

Wildcard in Java Generics cont.

```
2
3 public class Test {
4
5     //creating a method that accepts only the child class of Shape
6     public static void drawShapes(List <? extends Shape> lists) {
7         for(Shape s: lists)
8             s.draw();
9     }
10
11    public static void main(String args[]) {
12        List<Rectangle> list1 = new ArrayList<>();
13        list1.add(new Rectangle());
14
15        List<Circle> list2 = new ArrayList<>();
16        list2.add(new Circle());
17        list2.add(new Circle());
18
19        drawShapes(list1);
20        drawShapes(list2);
21    }
22 }
```

Drawing Rectangle
Drawing Circle
Drawing Circle

test4.java

Limitations in Generics

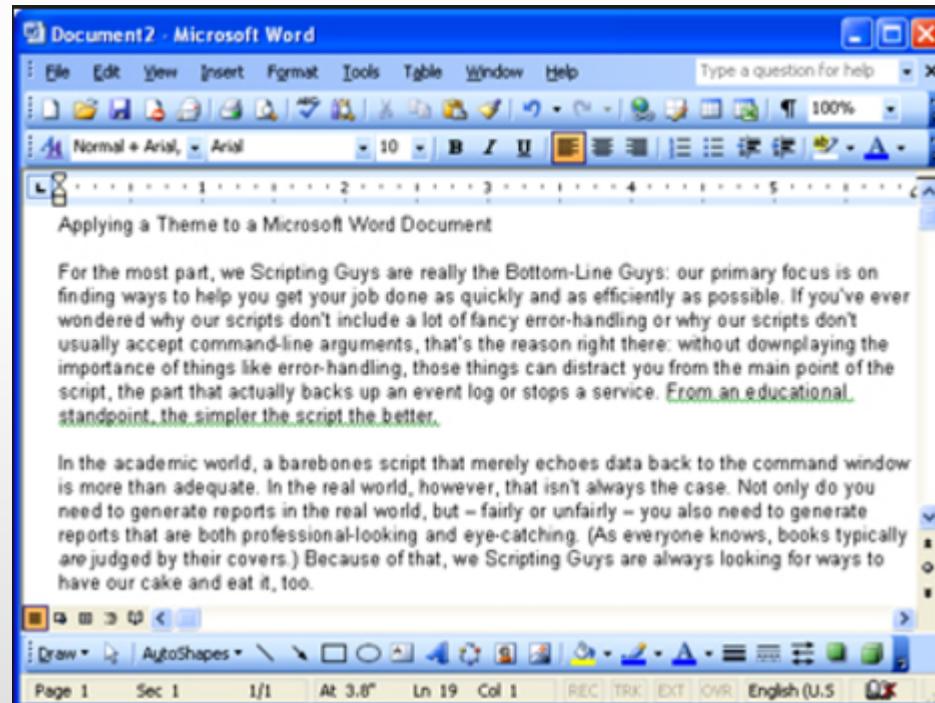
- **Type parameters cannot be instantiated** - It is not possible to create instances of a type parameter
- **Restriction on static members** - No static member can use a type parameter declared by the enclosing class. Note that you can declare static generic methods.
- **Generic array restriction** -
 - Cannot instantiate an array whose element type is a type parameter.
 - Cannot create an array of type specific references
- **Generic exception restriction** - A generic class cannot extend Throwable. This means we cannot create generic exception classes.

Threads

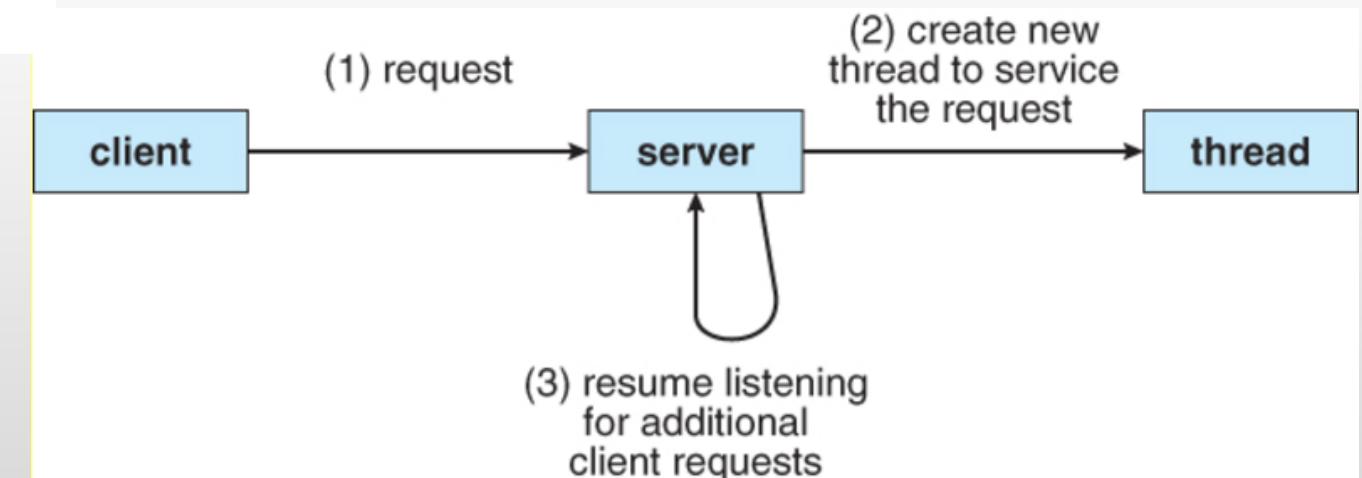
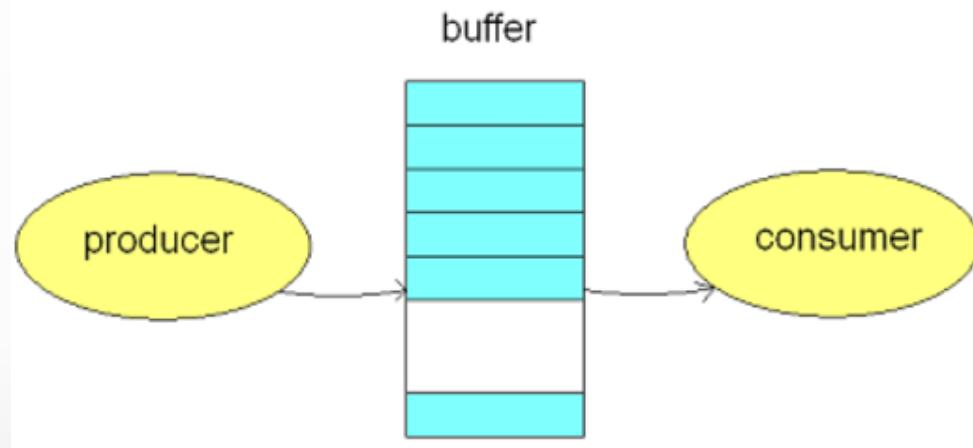
Object Oriented Programming (OOP)
2nd Year – Semester 1
By Udara Samaratunge

What is a Thread?

Examples:-



Other Scenarios Threads can be Applied



Thread Vs. Process

-
- **Threads** are easier to create than **processes** since they don't require a separate address space.
 - Threads are considered **lightweight** because they use far **less resources** than processes.
 - Processes are typically **independent**, while **threads** exist as **subsets** of a process
 - Processes have **separate address spaces**, whereas **threads** share their address space
 - Context switching between **threads** in the same process is typically **faster** than context switching between **processes**.

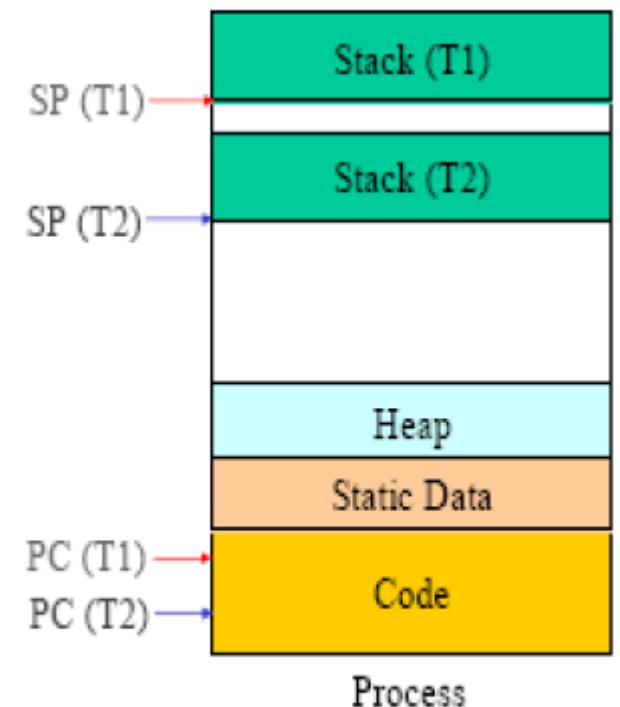
Thread Vs. Process

➤ A **Thread** in execution works with

- thread ID
- Registers (program counter and working register set)
- Stack (for procedure call parameters, local variables etc.)

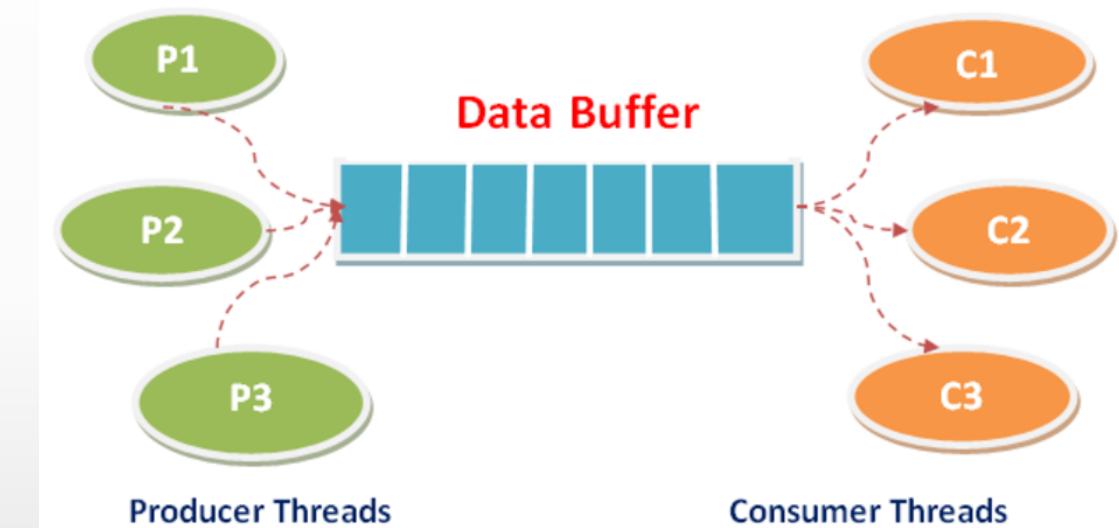
➤ A **thread** *shares* with other threads a process's (to which it belongs to)

- Code section
- Data section (static + heap)
- Permissions
- Other resources (e.g. files)



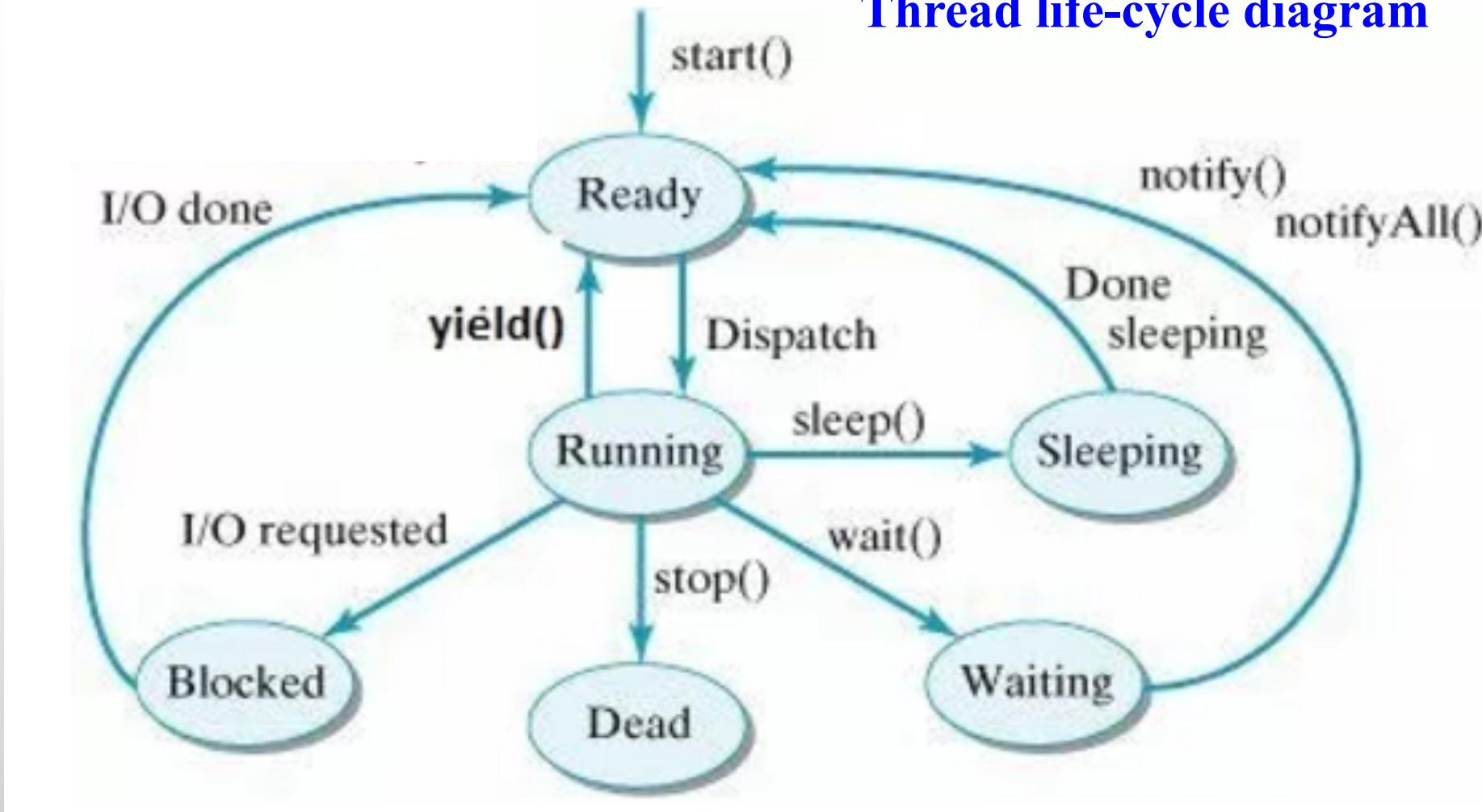
**Process with
2 threads**

Multi-threaded Environment

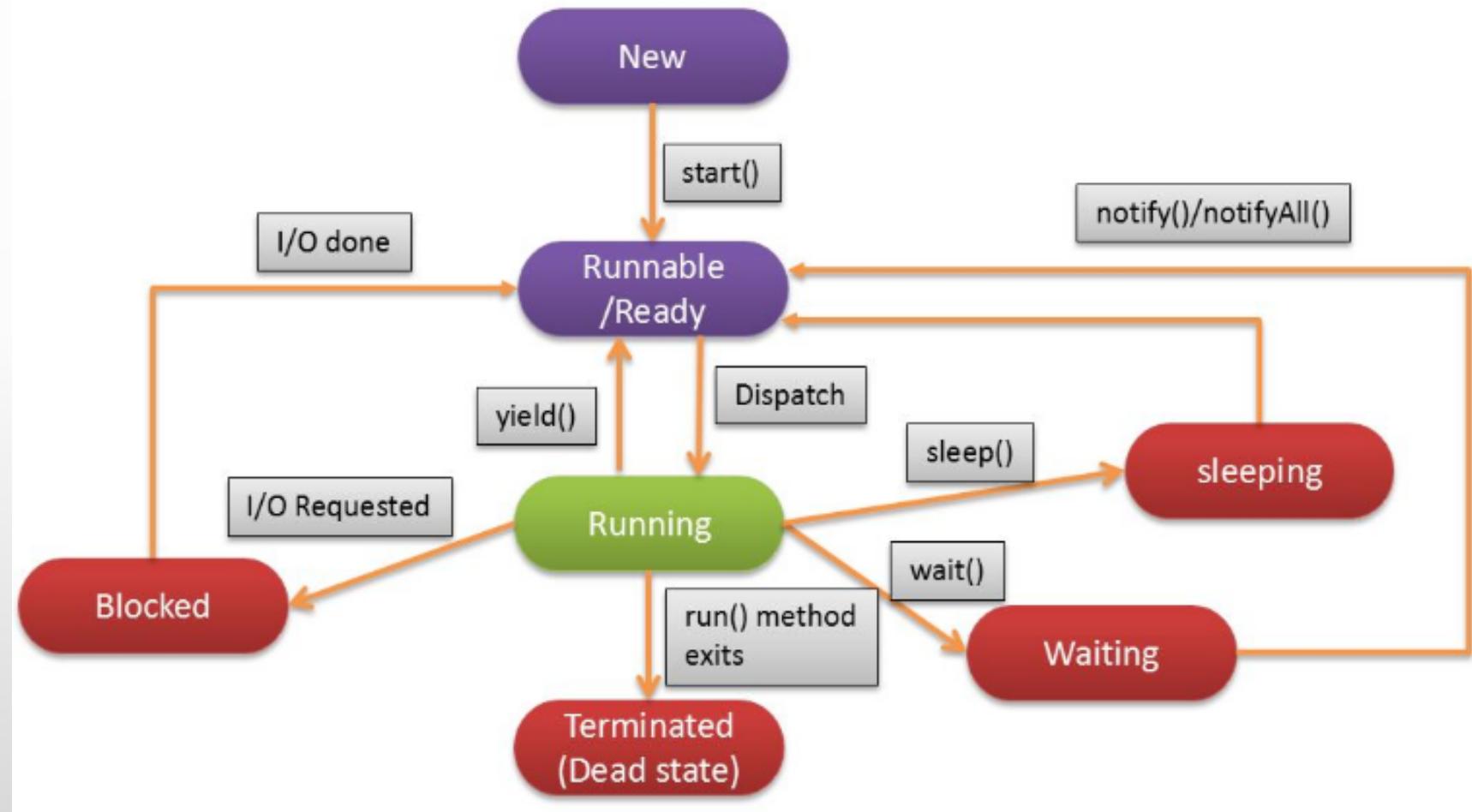


Thread Life Cycle

Thread life-cycle diagram

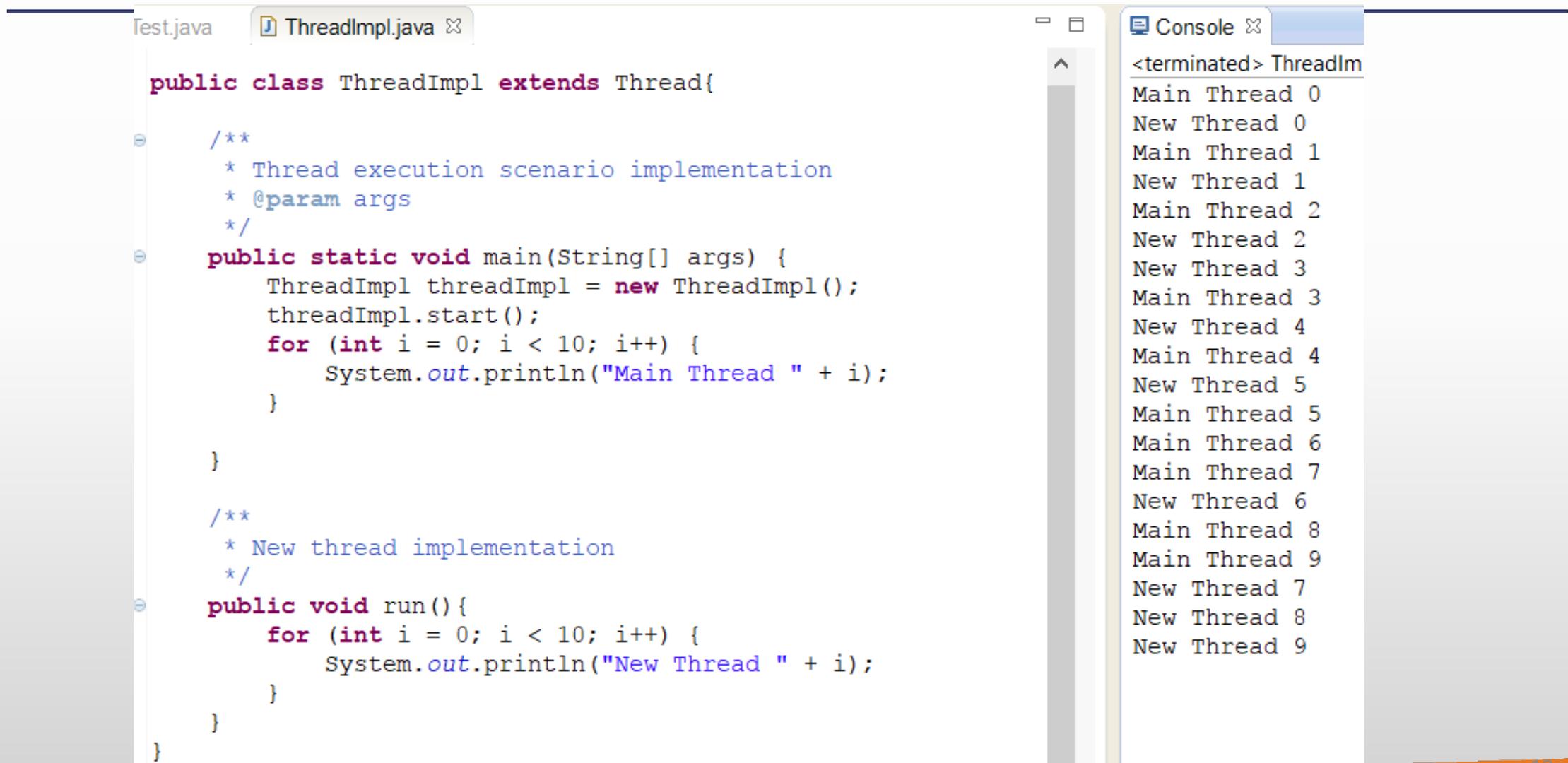


Thread States



Thread Implementation

Extends Thread class



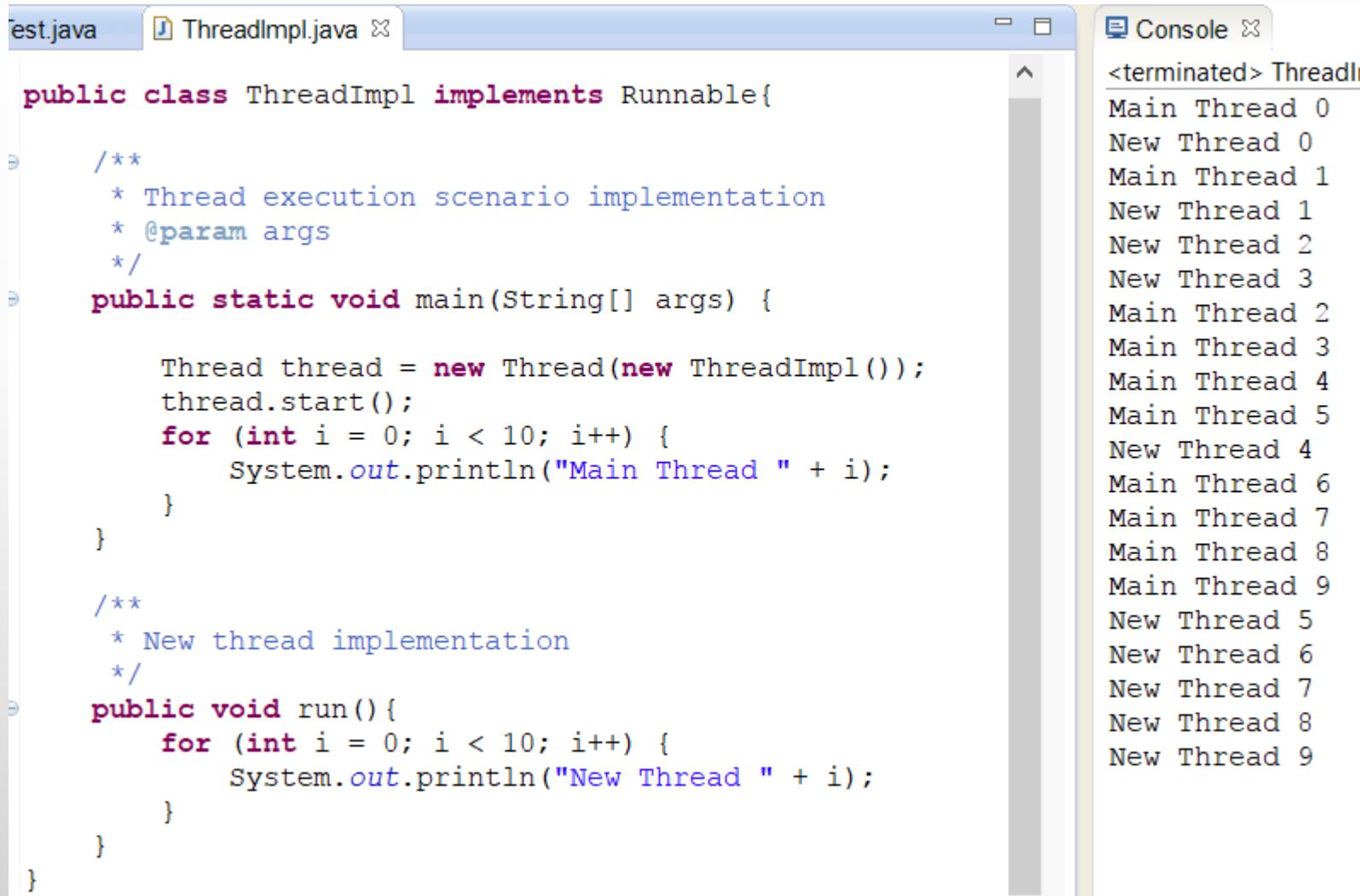
The screenshot shows an IDE interface with two files open: `Test.java` and `ThreadImpl.java`. The `Test.java` file contains a main method that creates a `ThreadImpl` object and starts it. The `ThreadImpl` class implements the `Runnable` interface and overrides the `run` method to print "New Thread" followed by an index from 0 to 9. The `main` method of `Test.java` prints "Main Thread" followed by indices from 0 to 9. The `Console` tab shows the execution output, alternating between "Main Thread" and "New Thread" for each index from 0 to 9.

```
public class ThreadImpl extends Thread{  
  
    /**  
     * Thread execution scenario implementation  
     * @param args  
     */  
    public static void main(String[] args) {  
        ThreadImpl threadImpl = new ThreadImpl();  
        threadImpl.start();  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Main Thread " + i);  
        }  
  
        /*  
         * New thread implementation  
         */  
        public void run(){  
            for (int i = 0; i < 10; i++) {  
                System.out.println("New Thread " + i);  
            }  
        }  
    }  
}
```

Console Output:

```
<terminated> ThreadImpl  
Main Thread 0  
New Thread 0  
Main Thread 1  
New Thread 1  
Main Thread 2  
New Thread 2  
New Thread 3  
Main Thread 3  
New Thread 4  
Main Thread 4  
New Thread 5  
Main Thread 5  
Main Thread 6  
Main Thread 7  
New Thread 6  
Main Thread 8  
Main Thread 9  
New Thread 7  
New Thread 8  
New Thread 9
```

Implements Runnable Interface



The screenshot shows an IDE interface with two tabs: 'test.java' and 'ThreadImpl.java'. The 'ThreadImpl.java' tab is active, displaying the following Java code:

```
public class ThreadImpl implements Runnable{

    /**
     * Thread execution scenario implementation
     * @param args
     */
    public static void main(String[] args) {

        Thread thread = new Thread(new ThreadImpl());
        thread.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("Main Thread " + i);
        }
    }

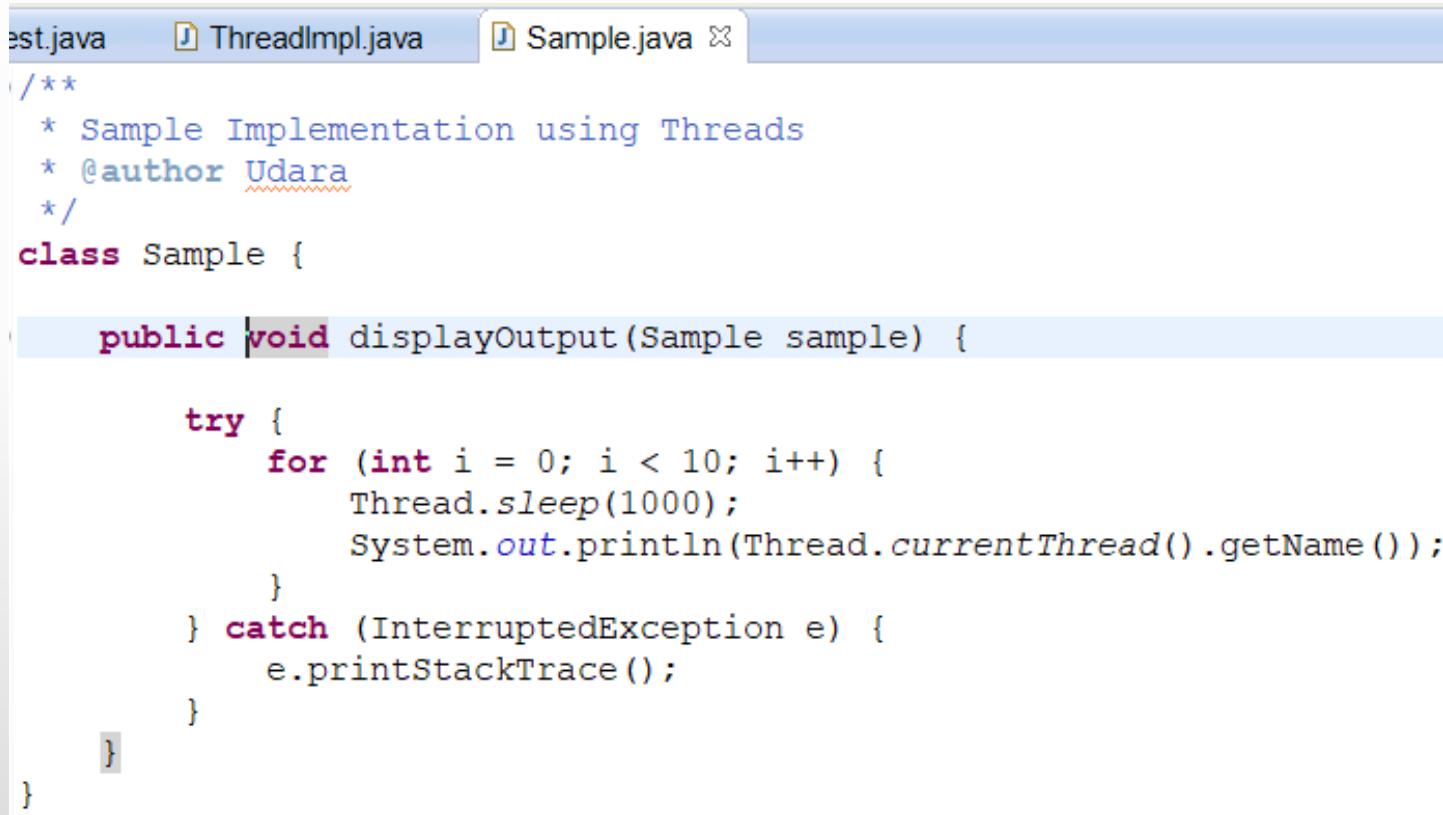
    /**
     * New thread implementation
     */
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println("New Thread " + i);
        }
    }
}
```

To the right of the code editor is a 'Console' window showing the output of the program. The output consists of two sets of ten lines, each representing either a 'Main Thread' or a 'New Thread'. The first set (Main Thread) starts at index 0 and ends at index 9. The second set (New Thread) starts at index 0 and ends at index 9.

```
<terminated> ThreadImpl
Main Thread 0
New Thread 0
Main Thread 1
New Thread 1
New Thread 2
New Thread 3
Main Thread 2
Main Thread 3
Main Thread 4
Main Thread 5
New Thread 4
Main Thread 6
Main Thread 7
Main Thread 8
Main Thread 9
New Thread 5
New Thread 6
New Thread 7
New Thread 8
New Thread 9
```

Thread Synchronization

Threads are not synchronized



```
test.java  ThreadImpl.java  Sample.java ✘
/*
 * Sample Implementation using Threads
 * @author Udara
 */
class Sample {

    public void displayOutput(Sample sample) {
        try {
            for (int i = 0; i < 10; i++) {
                Thread.sleep(1000);
                System.out.println(Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- When Thread sleeps it throws InterruptedException
- When Thread sleep it keeps the lock with it

Threads are not synchronized

The screenshot shows an IDE interface with two tabs open: `ThreadImpl.java` and `Sample.java`. The `ThreadImpl.java` tab contains the following code:

```
public class ThreadImpl extends Thread{
    Sample sample;
    String name;
    public static final String THREAD0 = "Thread 0";
    public static final String THREAD1 = "Thread 1";

    public ThreadImpl(Sample sample, String name) {
        this.sample = sample;
        this.name = name;
    }

    /**
     * Thread execution scenario implementation
     * @param args
     */
    public static void main(String[] args) {
        Sample sample = new Sample();
        ThreadImpl threadImpl1 = new ThreadImpl(sample, THREAD0);
        ThreadImpl threadImpl2 = new ThreadImpl(sample, THREAD1);
        threadImpl1.start();
        threadImpl2.start();
    }

    /**
     * New thread implementation
     */
    public void run(){
        sample.displayOutput(sample);
    }
}
```

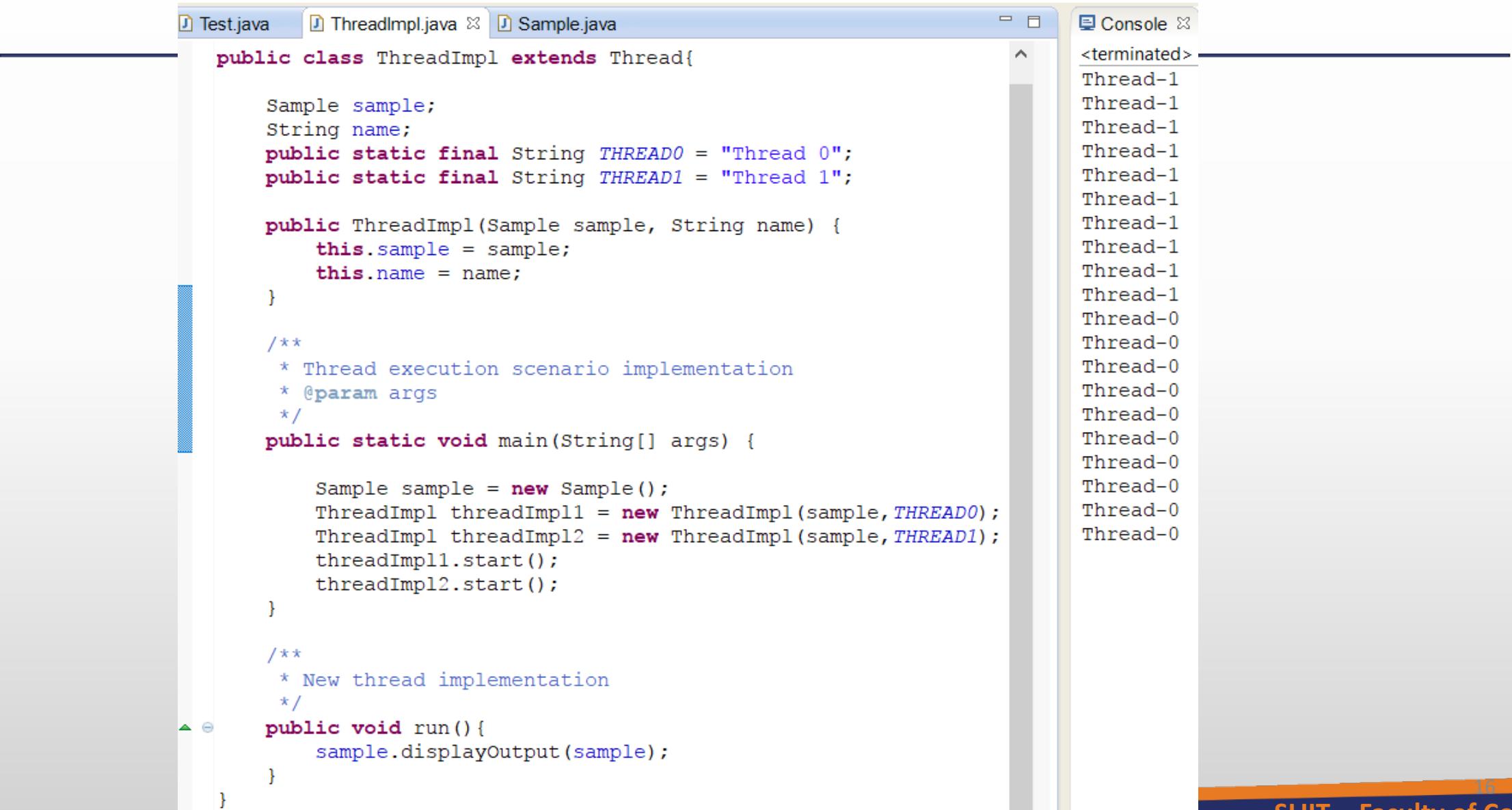
The `Sample.java` tab is currently selected. The `Console` tab on the right displays the output of the program, which consists of interleaved messages from two threads. The output is as follows:

```
<terminated> 1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
```

Thread Synchronized Method

```
/**  
 * Sample Implementation using Threads  
 * @author Udar  
 */  
class Sample {  
  
    public synchronized void displayOutput(Sample sample) {  
  
        try {  
            for (int i = 0; i < 10; i++) {  
                Thread.sleep(1000);  
                System.out.println(Thread.currentThread().getName());  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Thread Synchronization



The screenshot shows an IDE interface with three tabs: Test.java, ThreadImpl.java, and Sample.java. The Test.java tab is active, displaying the following code:

```
public class ThreadImpl extends Thread{  
  
    Sample sample;  
    String name;  
    public static final String THREAD0 = "Thread 0";  
    public static final String THREAD1 = "Thread 1";  
  
    public ThreadImpl(Sample sample, String name) {  
        this.sample = sample;  
        this.name = name;  
    }  
  
    /**  
     * Thread execution scenario implementation  
     * @param args  
     */  
    public static void main(String[] args) {  
  
        Sample sample = new Sample();  
        ThreadImpl threadImpl1 = new ThreadImpl(sample, THREAD0);  
        ThreadImpl threadImpl2 = new ThreadImpl(sample, THREAD1);  
        threadImpl1.start();  
        threadImpl2.start();  
    }  
  
    /**  
     * New thread implementation  
     */  
    public void run(){  
        sample.displayOutput(sample);  
    }  
}
```

The Console tab shows the output of the program, which consists of two columns of text. The left column contains 12 instances of "Thread-1" and the right column contains 12 instances of "Thread-0".

Thread-1	Thread-0
Thread-1	Thread-0

Thread Synchronization block

```
/**  
 * Sample Implementation using Threads  
 * @author Udara  
 */  
  
class Sample {  
  
    public void displayOutput(Sample sample) {  
  
        synchronized (sample) {  
            try {  
                for (int i = 0; i < 10; i++) {  
                    Thread.sleep(1000);  
                    System.out.println(Thread.currentThread().getName());  
                }  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Thread Synchronization block

```
est.java ThreadImpl.java Sample.java ^

public class ThreadImpl extends Thread{

    Sample sample;
    String name;
    public static final String THREAD0 = "Thread 0";
    public static final String THREAD1 = "Thread 1";

    public ThreadImpl(Sample sample, String name) {
        this.sample = sample;
        this.name = name;
    }

    /**
     * Thread execution scenario implementation
     * @param args
     */
    public static void main(String[] args) {

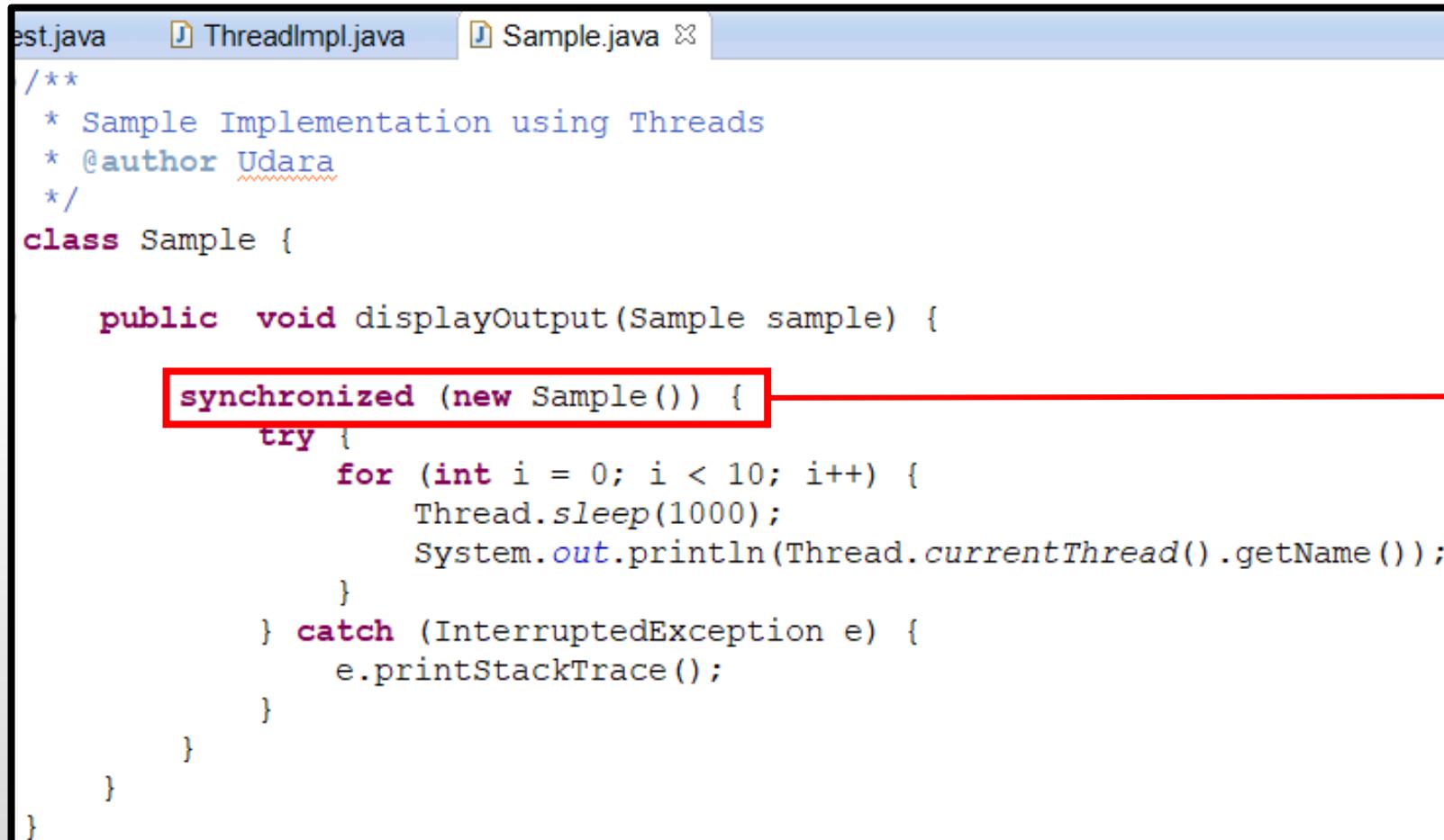
        Sample sample = new Sample();
        ThreadImpl threadImpl1 = new ThreadImpl(sample, THREAD0);
        ThreadImpl threadImpl2 = new ThreadImpl(sample, THREAD1);
        threadImpl1.start();
        threadImpl2.start();
    }

    /**
     * New thread implementation
     */
    public void run(){
        sample.displayOutput(sample);
    }
}
```

Console

```
<terminated>
Thread-0
Thread-1
```

Thread Synchronization block with lock change



```
test.java  ThreadImpl.java  Sample.java ✘
/*
 * Sample Implementation using Threads
 * @author Udar
 */
class Sample {

    public void displayOutput(Sample sample) {
        synchronized (new Sample()) {
            try {
                for (int i = 0; i < 10; i++) {
                    Thread.sleep(1000);
                    System.out.println(Thread.currentThread().getName());
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Each Thread get separate object to lock. So method not synchronized

Thread Synchronization block with lock change

```
est.java ThreadImpl.java Sample.java

public class ThreadImpl extends Thread{

    Sample sample;
    String name;
    public static final String THREAD0 = "Thread 0";
    public static final String THREAD1 = "Thread 1";

    public ThreadImpl(Sample sample, String name) {
        this.sample = sample;
        this.name = name;
    }

    /**
     * Thread execution scenario implementation
     * @param args
     */
    public static void main(String[] args) {

        Sample sample = new Sample();
        ThreadImpl threadImpl1 = new ThreadImpl(sample, THREAD0);
        ThreadImpl threadImpl2 = new ThreadImpl(sample, THREAD1);
        threadImpl1.start();
        threadImpl2.start();
    }

    /**
     * New thread implementation
     */
    public void run() {
        sample.displayOutput(sample);
    }
}
```

```
Console >
<terminated>
Thread-1
Thread-0
Thread-1
Thread-0
Thread-0
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-0
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
```

```

public class Singleton {
    private Singleton() {
    }

    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
            System.out.println("Instance Created "
                + Thread.currentThread().getName());
        }
        return instance;
    }
}

public class ThreadSafeSingleton {
    private ThreadSafeSingleton() {
    }

    private static ThreadSafeSingleton instance;

    public static ThreadSafeSingleton getInstance() {
        if (instance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if (instance == null) {
                    instance = new ThreadSafeSingleton();
                    System.out.println("Thread Safe Instance created "
                        + Thread.currentThread().getName());
                }
            }
        }
        return instance;
    }
}

```

Synchronized block importance

SingletonTest.java | Test.java | ThreadImpl.java | Sample.java

```

public class SingletonTest implements Runnable {
    /**
     * @param args
     */
    public static void main(String[] args) {
        new Thread(new SingletonTest()).start();

        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }
}

```

Console

```

<terminated> SingletonTest [Java Application] C:\Program Files\Java\jre7\bin\
Instance Created Thread-0
Instance Created main
Thread Safe Instance created Thread-0

```

Thread Join method

- The join() method waits for a thread to die.
- It causes the currently running threads to stop executing until the thread it joins with completes its task.

ThreadJoin.java

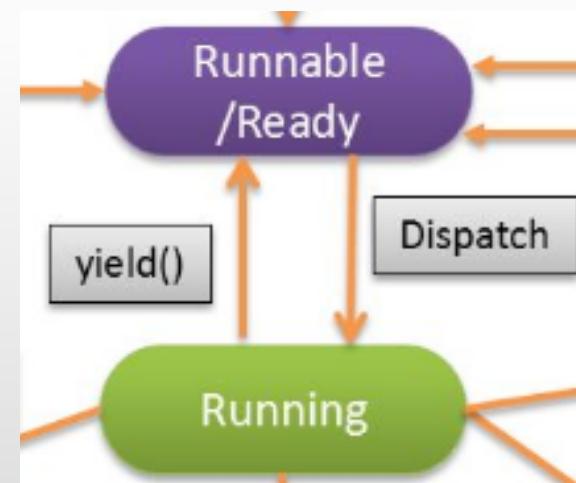
```
public class ThreadJoin extends Thread{  
  
    public void run() {  
        Thread t = Thread.currentThread();  
        System.out.println("Started executing " + t.getName());  
  
        for (int i = 0; i < 10; i++) {  
            System.out.println(t.getName() + i);  
        }  
        System.out.println("Finished executing " + t.getName());  
    }  
  
    public static void main(String args[]) throws Exception {  
  
        Thread t = new Thread(new ThreadJoin(),"New Thread ");  
        t.start();  
        System.out.println("Started executing main thread");  
  
        // waits for main thread to die and allow execute the other thread  
        t.join();  
  
        for (int i = 0; i < 10; i++) {  
            System.out.println(Thread.currentThread().getName() + i);  
        }  
        System.out.println("Finished executing " + Thread.currentThread().getName());  
    }  
}
```

Console

```
<terminated> ThreadJoin [Java Application]  
Started executing main thread  
Started executing New Thread  
New Thread 0  
New Thread 1  
New Thread 2  
New Thread 3  
New Thread 4  
New Thread 5  
New Thread 6  
New Thread 7  
New Thread 8  
New Thread 9  
Finished executing New Thread  
main0  
main1  
main2  
main3  
main4  
main5  
main6  
main7  
main8  
main9  
Finished executing main
```

Thread Yield method

- Yield() is used to give the other threads of the same priority a chance to execute
- This causes current running thread to move to runnable state. [running state to ready state]



ThreadYield.java

```
public class ThreadYield extends Thread{
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Started executing " + t.getName());

        for (int i = 0; i < 10; i++) {
            System.out.println(t.getName() + i);
        }
        System.out.println("Finished executing " + t.getName());
    }

    public static void main(String args[]) throws Exception {
        Thread t = new Thread(new ThreadYield(), "New Thread ");
        t.start();
        System.out.println("Started executing main thread");
        /*
         * temporarily stop executing main thread and give chance to
         * newly created thread.
         */
        t.yield();

        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + i);
        }
        System.out.println("Finished executing " + Thread.currentThread().getName());
    }
}
```

Console

```
<terminated> ThreadYield [Java Application]
Started executing main thread
Started executing New Thread
New Thread 0
main0
New Thread 1
New Thread 2
New Thread 3
New Thread 4
main1
main2
main3
main4
main5
main6
main7
main8
main9
Finished executing main
New Thread 5
New Thread 6
New Thread 7
New Thread 8
New Thread 9
Finished executing New Thread
```

Thread wait and notify

- Once thread executes **wait()** method it **releases the lock** and state changed from **Runnable** to **waiting state**.
- Other thread can acquire the **lock** and continue execution.
- Once **notify()** method get executed the **waited thread** move to **ready state** and resume its execution.
- **notifyAll()** This **wakes up all the threads** that called **wait()** on **the same object**.

Thread wait and notify

```
class Thread1 extends Thread {  
  
    ThreadNotifyTest object;  
  
    public Thread1(ThreadNotifyTest object, String name) {  
        super(object, name);  
        this.object = object;  
    }  
  
    public void run() {  
        synchronized (object) {  
            try {  
                System.out.println("Started "  
                    + Thread.currentThread().getName() + " wait");  
                object.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            for (int i = 0; i < 10; i++) {  
                System.out.println(Thread.currentThread().getName() + " " + i);  
            }  
        }  
    }  
}
```

- **Thread wait() method throw an InterruptedException**
- **But it releases the lock**

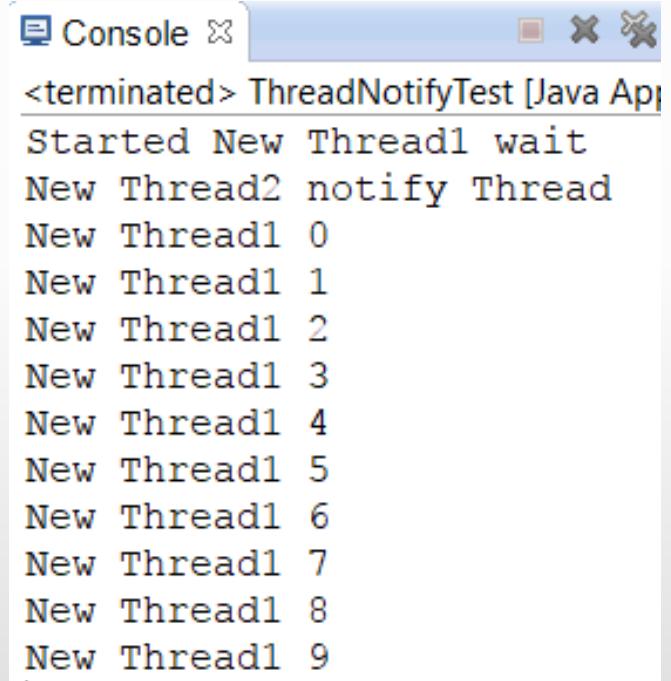
Thread wait and notify

```
class Thread2 extends Thread {  
  
    ThreadNotifyTest object;  
  
    public Thread2(ThreadNotifyTest object, String name) {  
        super(object, name);  
        this.object = object;  
    }  
  
    public void run() {  
        synchronized (object) {  
            System.out.println(Thread.currentThread().getName()  
                + " notify Thread");  
            object.notify();  
        }  
    }  
}
```

Thread wait and notify

```
public class ThreadNotifyTest extends Thread {  
  
    public static void main(String args[]) throws Exception {  
  
        ThreadNotifyTest threadNotify = new ThreadNotifyTest();  
        Thread1 t1 = new Thread1(threadNotify, "New Thread1");  
        Thread2 t2 = new Thread2(threadNotify, "New Thread2");  
  
        t1.start();  
        t2.start();  
    }  
}
```

Response



Console > <terminated> ThreadNotifyTest [Java API]
Started New Thread1 wait
New Thread2 notify Thread
New Thread1 0
New Thread1 1
New Thread1 2
New Thread1 3
New Thread1 4
New Thread1 5
New Thread1 6
New Thread1 7
New Thread1 8
New Thread1 9

Example for notifyAll with multi-threaded scenario

```
class Thread1 extends Thread {  
  
    ThreadNotifyTest object;  
  
    public Thread1(ThreadNotifyTest object, String name) {  
        super(object, name);  
        this.object = object;  
    }  
  
    public void run() {  
        synchronized (object) {  
            try {  
                System.out.println("Started " + Thread.currentThread().getName() + " wait");  
                object.wait();  
                System.out.println("Started " + Thread.currentThread().getName() + " notified");  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            for (int i = 0; i < 10; i++) {  
                System.out.println(Thread.currentThread().getName() + " " + i);  
            }  
        }  
    }  
}
```

by Udara Samaratunge

Example for notifyAll with multi-threaded scenario

```
class Thread2 extends Thread {  
  
    ThreadNotifyTest object;  
  
    public Thread2(ThreadNotifyTest object, String name) {  
        super(object, name);  
        this.object = object;  
    }  
  
    public void run() {  
        synchronized (object) {  
            try {  
                System.out.println("Started " + Thread.currentThread().getName() + " wait");  
                object.wait();  
                System.out.println("Started " + Thread.currentThread().getName() + " notified");  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            for (int i = 0; i < 10; i++) {  
                System.out.println(Thread.currentThread().getName() + " " + i);  
            }  
        }  
    }  
}
```

Thread notifyAll() method

```
class Thread3 extends Thread {  
  
    ThreadNotifyTest object;  
  
    public Thread3(ThreadNotifyTest object, String name) {  
        super(object, name);  
        this.object = object;  
    }  
  
    public void run() {  
        synchronized (object) {  
            System.out.println("notifyAll Thread executed");  
            object.notifyAll();  
        }  
    }  
}
```

This method awake all threads which are waiting with the same objects

Output of notifyAll Scenario

```
public class ThreadNotifyTest extends Thread {  
  
    public static void main(String args[]) throws Exception {  
  
        ThreadNotifyTest threadNotify = new ThreadNotifyTest();  
        Thread1 t1 = new Thread1(threadNotify, "New Thread1");  
        Thread2 t2 = new Thread2(threadNotify, "New Thread2");  
        Thread3 t3 = new Thread3(threadNotify, "New Thread3");  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```



The screenshot shows the Java IDE's console window with the title 'Console'. The output text is:
<terminated> ThreadNotifyTest [Java Application]
Started New Thread1 wait
Started New Thread2 wait
notifyAll Thread executed
Started New Thread2 notified
New Thread2 0
New Thread2 1
New Thread2 2
New Thread2 3
New Thread2 4
New Thread2 5
New Thread2 6
New Thread2 7
New Thread2 8
New Thread2 9
Started New Thread1 notified
New Thread1 0
New Thread1 1
New Thread1 2
New Thread1 3
New Thread1 4
New Thread1 5
New Thread1 6
New Thread1 7
New Thread1 8
New Thread1 9

Thread Priority

```
System.out.println(Thread.MIN_PRIORITY);      => 1
```

```
System.out.println(Thread.NORM_PRIORITY);     => 5
```

```
System.out.println(Thread.MAX_PRIORITY);      => 10
```

```
public class ThreadPriority {  
    public static void main(String[] args) {  
        System.out.println(Thread.MIN_PRIORITY);  
        System.out.println(Thread.NORM_PRIORITY);  
        System.out.println(Thread.MAX_PRIORITY);  
        System.out.println("Existing thread priority = "  
                         + Thread.currentThread().getPriority());  
    }  
}
```

```
<terminated> ThreadPriority [Java Application]  
1  
5  
10  
Existing thread priority = 5
```

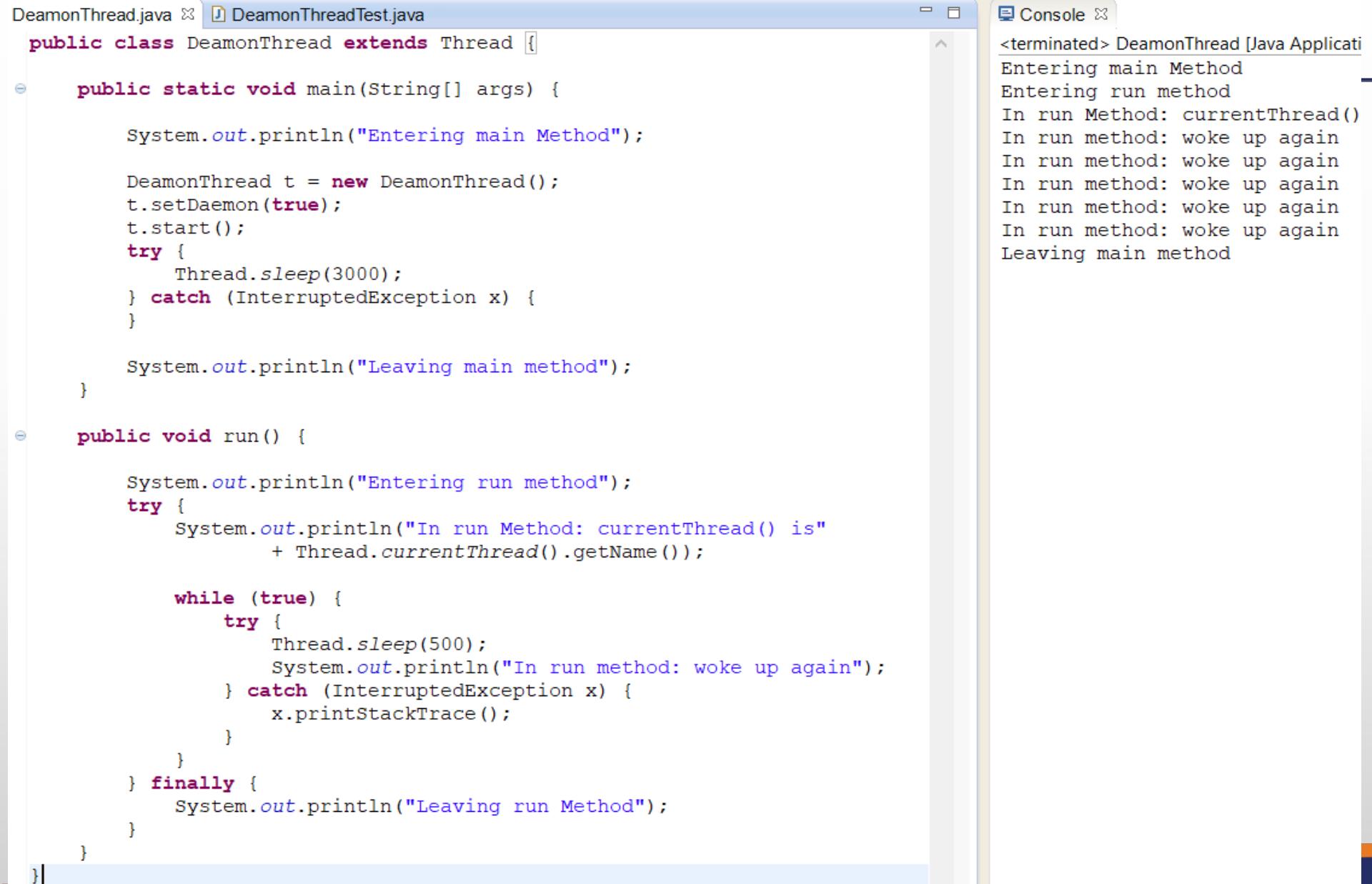
Daemon Threads

- Daemon Threads are “background threads”.
- That provides service to other threads, e.g. The garbage collection thread.
- The Java VM will not exit if non-daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon thread die when the Java VM exits.

Daemon Thread Example

- Since newly created thread is **daemon thread** when **main thread completes its execution JavaVM will not wait** until Daemon thread completes its execution.
- So it exit and Daemon thread automatically Die

Daemon Thread Example



The screenshot shows an IDE interface with two tabs: "DeamonThread.java" and "DeamonThreadTest.java". The "DeamonThread.java" tab is active, displaying the following Java code:

```
public class DeamonThread extends Thread {  
  
    public static void main(String[] args) {  
  
        System.out.println("Entering main Method");  
  
        DeamonThread t = new DeamonThread();  
        t.setDaemon(true);  
        t.start();  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException x) {  
        }  
  
        System.out.println("Leaving main method");  
    }  
  
    public void run() {  
  
        System.out.println("Entering run method");  
        try {  
            System.out.println("In run Method: currentThread() is"  
                + Thread.currentThread().getName());  
  
            while (true) {  
                try {  
                    Thread.sleep(500);  
                    System.out.println("In run method: woke up again");  
                } catch (InterruptedException x) {  
                    x.printStackTrace();  
                }  
            }  
        } finally {  
            System.out.println("Leaving run Method");  
        }  
    }  
}
```

The "Console" tab shows the output of the application:

```
<terminated> DeamonThread [Java Application]  
Entering main Method  
Entering run method  
In run Method: currentThread()  
In run method: woke up again  
Leaving main method
```

Commonly used methods for Thread class

1. **public void run()**: is used to perform action for a thread.
2. **public void start()**: starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds)**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join()**: waits for a thread to die.
5. **public void join(long miliseconds)**: waits for a thread to die for the specified milliseconds.
6. **public int getPriority()**: returns the priority of the thread.
7. **public int setPriority(int priority)**: changes the priority of the thread.
8. **public String getName()**: returns the name of the thread.
9. **public void setName(String name)**: changes the name of the thread.
10. **public Thread currentThread()**: returns the reference of currently executing thread.
11. **public int getId()**: returns the id of the thread.

Commonly used methods for Thread class

12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

The End



Object Oriented Programming

Week 09

Java Web Technologies

Learning Outcomes

At the end of the Lecture students should be able to get details of a few specific things related to the Java web technologies.

- MVC architecture
- JSP
- Servlet
- Maven Building tool

What is a Web Application Framework?

A web application framework is a piece of structural software that provides automation of common tasks of the domain as well as a built-in architectural solution that can be easily inherited by applications implemented on the framework.

Common Tasks of the Web Application

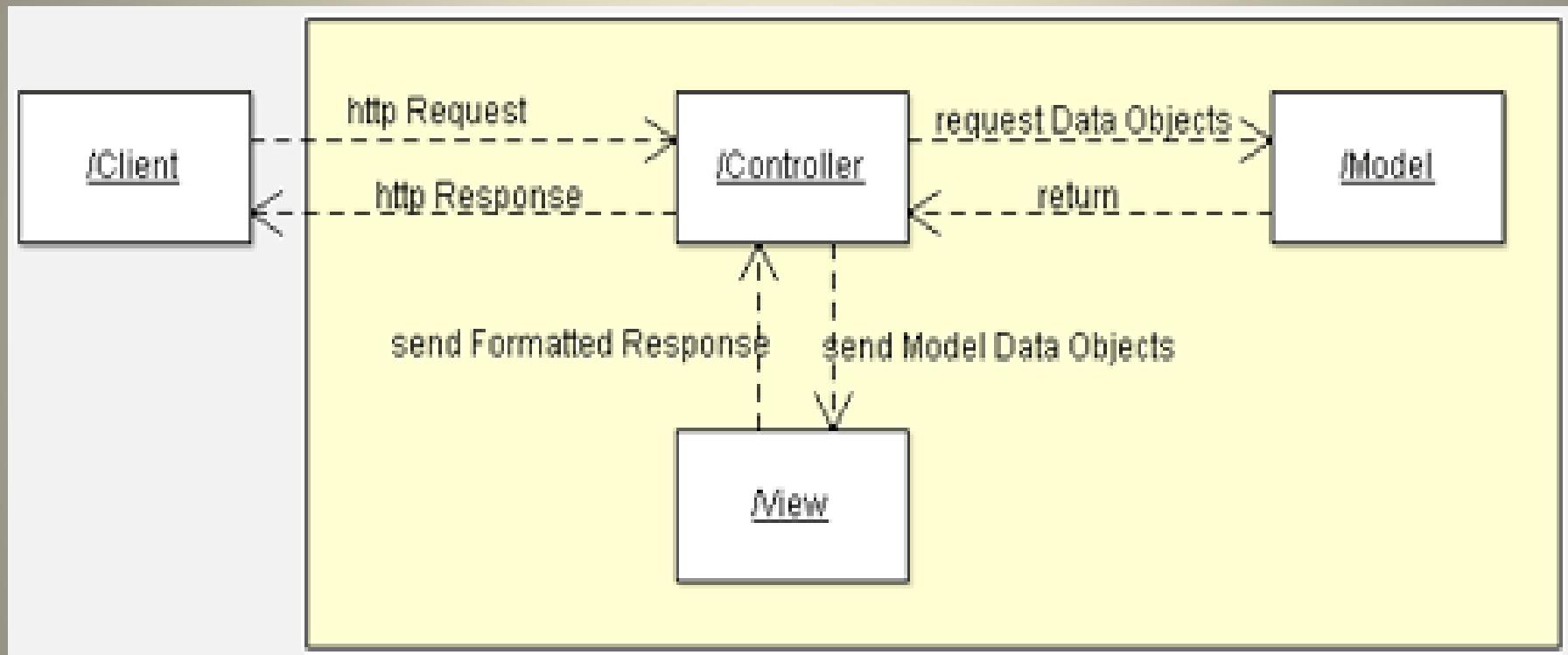
- Binding Request Parameters to Java Types
- Validating Data
- Making calls to business logic
- Making calls to the data layer
- Rendering presentation layer (HTML, ...)
- Providing internationalization and localization
- Keep Presentation Layer Separate from Data Layer

MVC

- The model view controller pattern is the most used pattern for today's world web applications
- It has been used for the first time in Smalltalk and then adopted and popularized by Java
- At present there are more than a dozen PHP web frameworks based on MVC pattern

MVC Cont..

- The **model** does all the computational work
 - It is input/output free
 - All communication with the model is via methods
- The **controller** tells the model what to do
 - User input goes to the controller
- The **view** shows results; it is a “window” into the model
 - The view can get results from the controller, or
 - The view can get results directly from the model



Applying MVC to Web Applications

- View:
 - HTML form; native Java interface; client-side script; applet
- Controller:
 - Java servlet; session Bean
- Model:
 - Entity Bean or other business logic object

Interchangeable Elements

- View:
 - HTML form becomes touch-screen
- Controller:
 - JSP becomes session bean
- Model:
 - Entity Bean
- Views and Controllers closely interact (HTML/JSP)
- If HTML code is written out entirely through JSP, the Controller and View (conceptually) merge
- A Controller-View pair works with one Model
- One Model may have multiple Controller-View pairs

MVC Advantages

- Single point of entry to Model object
- Multiple-client support
- Design clarity
- Modularity
- Controlled growth
- Portable
- *separation of concerns*

What is a Java Server Page (JSP)

- Java Server Pages (JSP)
 - A simplified, fast way to create **dynamic** web content
 - HTML or XML pages with embedded Java Code or Java Beans
 - Can be a mix of template data in HTML/XML with some dynamic content
- Java Server Pages are HTML pages embedded with snippets of Java code.
 - It is an inverse of a Java Servlet
- Four different elements are used in constructing JSPs
 - Scripting Elements
 - Implicit Objects
 - Directives
 - Actions

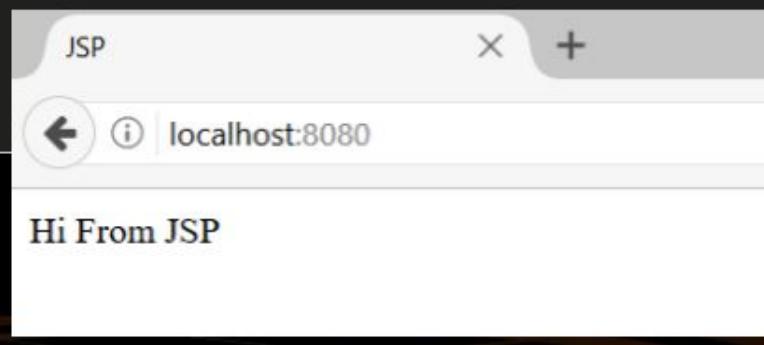
JSP Example

index.jsp

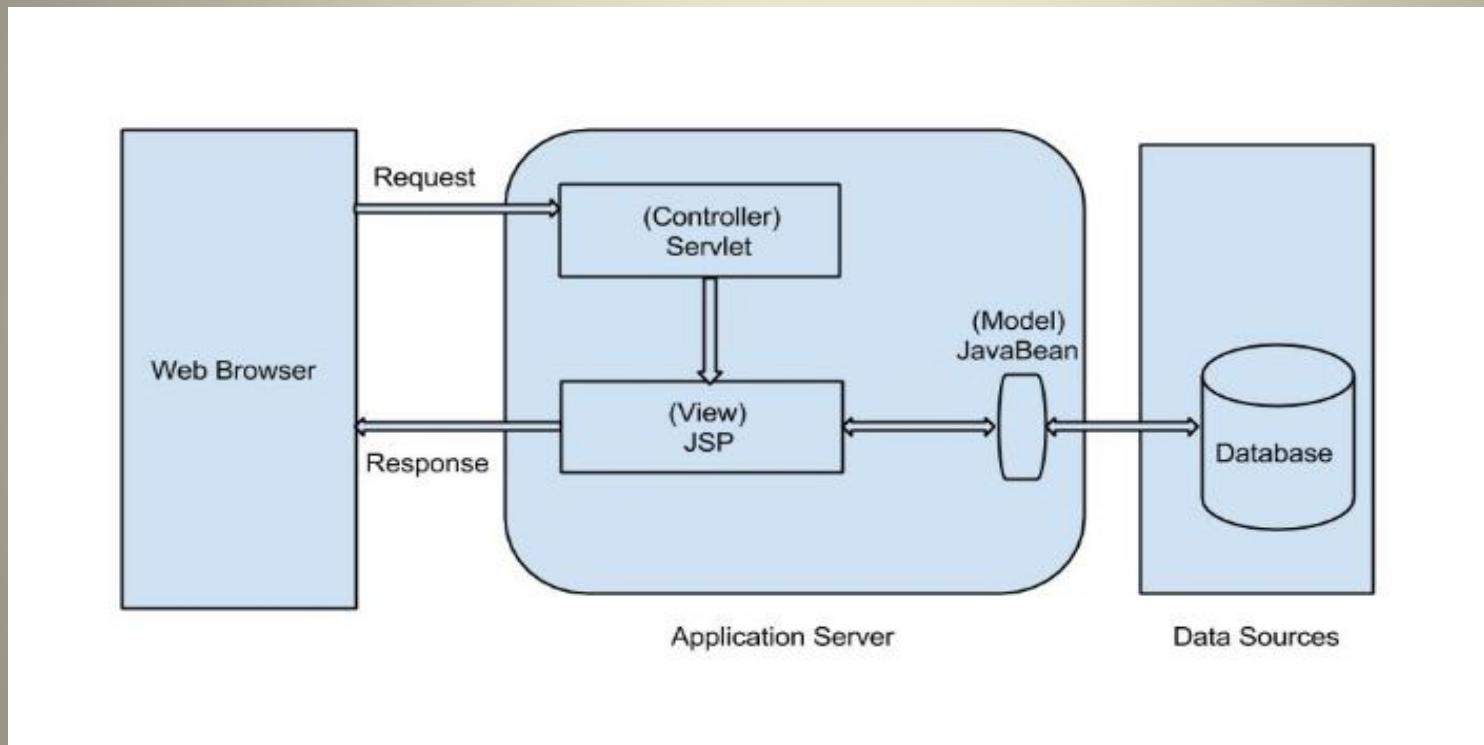
```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
  <head>
    <title>JSP</title>
  </head>
  <body>
    <%= new String("Hi From JSP")%>
  </body>
</html>
```

Page Directive

Java Code



How JSP works



Why to use JSP?

- High Performance
- Has access to all the powerful Enterprise Java APIs, including **JDBC, JNDI, EJB, JAXP**
- can be used in combination with servlets

Scripting Elements

- There are three kinds of scripting elements
 - Declarations
 - Scriptlets
 - Expressions

Declarations

- Declarations are used to define methods & instance variables
 - Do not produce any output that is sent to client
 - Embedded in <%! and %> delimiters

Example:

<%!

```
    Public void jspDestroy() {  
        System.out.println("JSP Destroyed");  
    }
```

```
    Public void jspInit() {  
        System.out.println("JSP Loaded");  
    }
```

```
    int myVar = 123;
```

%>

- The functions and variables defined are available to the JSP Page as well as to the servlet in which it is compiled

Scriptlets

- Used to embed java code in JSP pages.
- Contents of JSP go into _JSPpageservice() method
- Code should comply with syntactical and semantic construct of java
- Embedded in <% and %> delimiters
 - Example:

```
<%
    int x = 5;
    int y = 7;
    int z = x + y;
%>
```

Expressions

- Used to write dynamic content back to the browser.
 - If the output of expression is Java primitive the value is printed back to the browser
 - If the output is an object then the result of calling `toString` on the object is output to the browser
 - Embedded in `<%=` and `%>` delimiters

Example:

- `<%="Fred"+ " " + "Flintstone %>`
prints “Fred Flintstone” to the browser
- `<%=Math.sqrt(100)%>`
prints 10 to the browser

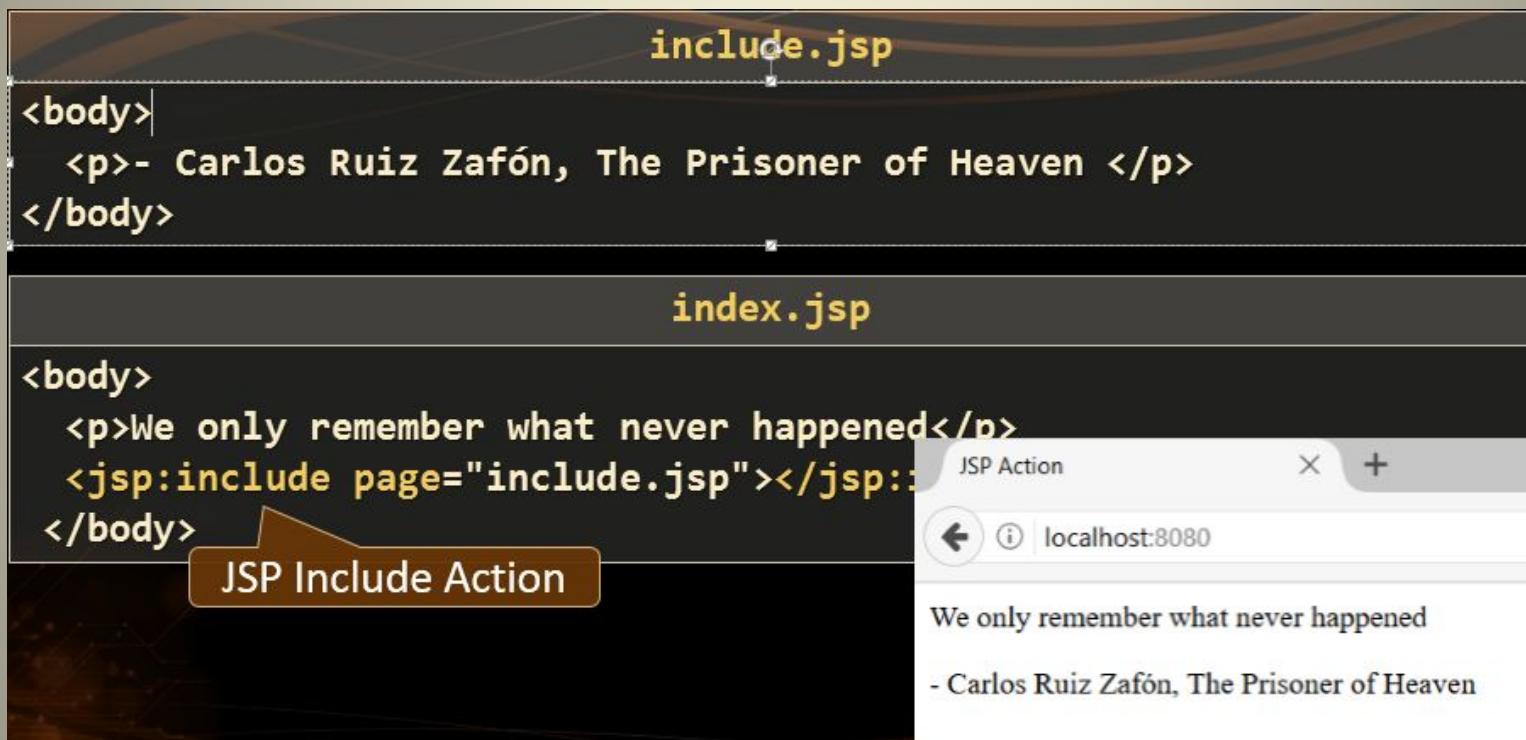
JSP Actions

- Processed during the request processing phase.
 - As opposed to JSP directives which are processed during translation
- Standard actions should be supported by J2EE compliant web servers
- Custom actions can be created using tag libraries
- The different actions are
 - Include action
 - Forward action
 - Param action
 - useBean action
 - getProperty action
 - setProperty action
 - plugIn action

JSP Actions - Include

- Include action used for including resources in a JSP page
 - Include directive includes resources in a JSP page at translation time
 - Include action includes response of a resource into the response of the JSP page
 - Same as including resources using RequestDispatcher interface
 - Changes in the included resource reflected while accessing the page.
 - Normally used for including dynamic resources
- Example
 - <jsp:include page="inlcudedPage.jsp">
 - Includes the the output of includedPage.jsp into the page where this is included.

JSP Actions



The screenshot illustrates the use of the JSP `<jsp:include>` action. In the top window, titled "include.jsp", the code includes a quote from Carlos Ruiz Zafón's "The Prisoner of Heaven". In the bottom window, titled "index.jsp", the code uses the `<jsp:include>` action to embed the content of "include.jsp" into the page. A callout bubble labeled "JSP Include Action" points to the include tag in the "index.jsp" code. The browser preview shows the final rendered output: "We only remember what never happened" followed by the quote from Carlos Ruiz Zafón.

```
include.jsp
<body>
  <p>- Carlos Ruiz Zafón, The Prisoner of Heaven </p>
</body>

index.jsp
<body>
  <p>We only remember what never happened</p>
  <jsp:include page="include.jsp"></jsp:include>
</body>
```

JSP Action × +

localhost:8080

We only remember what never happened

- Carlos Ruiz Zafón, The Prisoner of Heaven

JSP Actions - Param

- Used in conjunction with Include & Forward actions to include additional request parameters to the included or forwarded resource
- Example

```
<jsp:forward page="Param2.jsp">  
    <jsp:param name="FirstName" value="Sanjay">  
</jsp:forward>
```

- This will result in the forwarded resource having an additional parameter FirstName with a value of Sanjay

Syntax

- Scriptlet – nest Java code

```
<% System.out.println("Message in the Console"); %>
```

- Declaration Tag – define variables

```
<%! int x = 5; %>
```

- Expression Tag – print java variables

```
<%= x%>
```

- Comment Tag – add comments

```
<%-- This is JSP comment --%>
```

Simple JSP Loop

index.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <body>
    <%! int elements = 10;%>
    <% for (int i = 0; i < elements; i++) {%
      <div style="color: green">
        <%=i%>
      </div>
    }%>
  </body>
</html>
```

Declare Variable Prepare Loop Print Value

JSP Loop localhost:8080

0
1
2
3
4
5
6
7
8
9

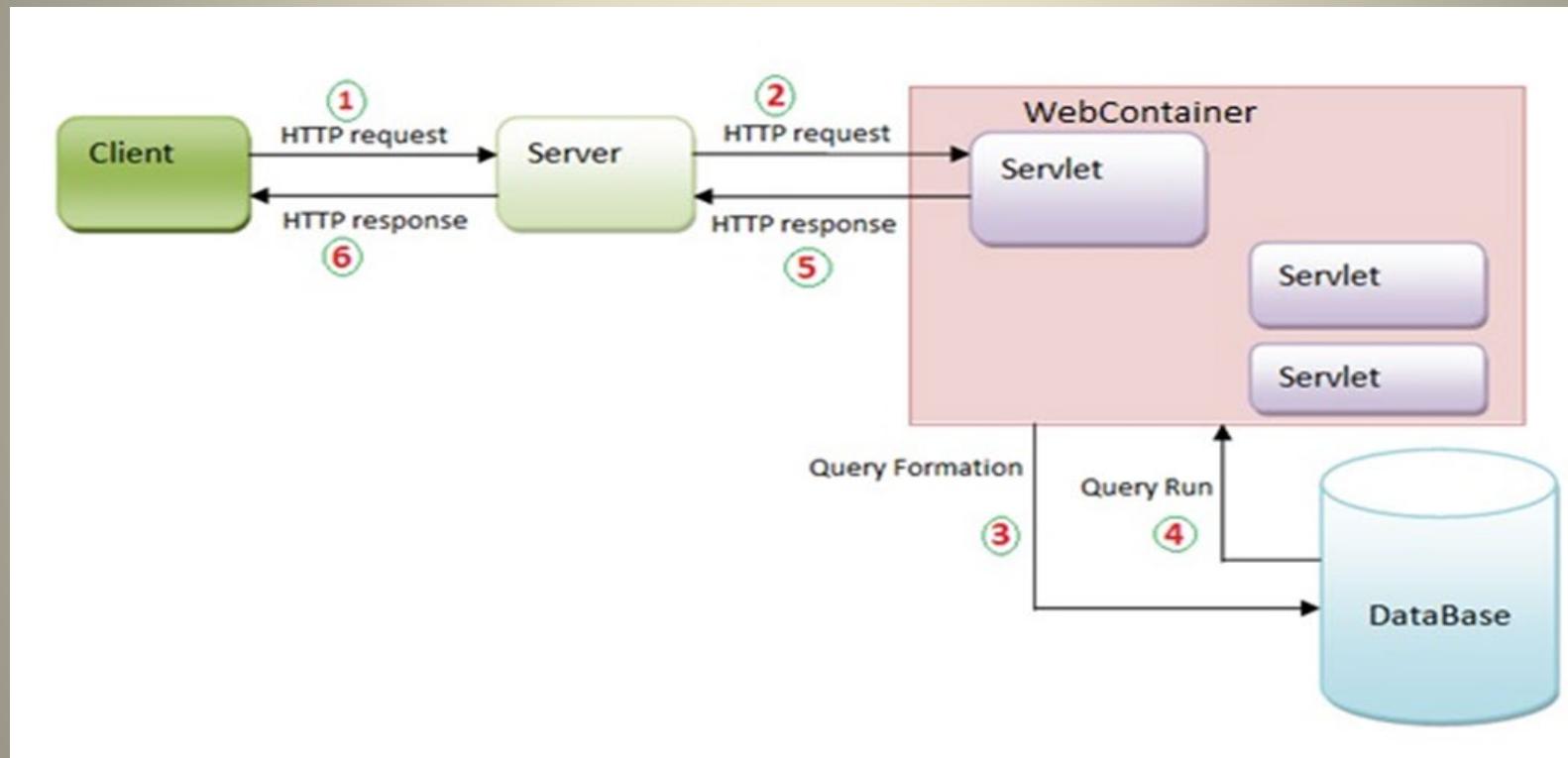
What is a Servlet?

- Servlets are small programs that execute on the server side of a web connection
- Servlets dynamically extend the functionality of a web server
- Java Servlets/JSP are part of the Sun's J2EE Enterprise Architecture
 - The web development part
- Java Servlet
 - is a simple, consistent mechanism for extending the functionality of a web server
 - Are precompiled Java programs that are executed on the server side.
 - Require a Servlet container to run in
 - Portable to any java application server

Why to Use Servlets?

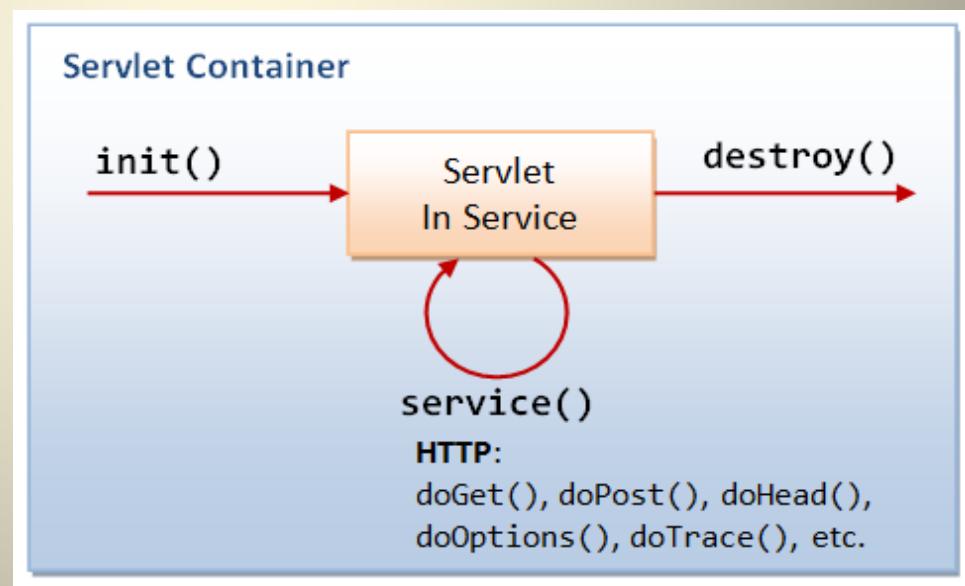
- Work well in a Heterogeneous Environments
 - OS and platform neutral
 - Work with all major web servers (IIS, Apache,etc..)
- Well defined Web Architecture framework
 - Standard built in services such as:
 - Standard Approach to Authentication using declarative security
 - vice programmatic security
 - Database connection pooling
 - Complete support for sessions via cookies and/or URL re-writing
 - Has automatic fallback to URL re-writing

How servlet works



The Life Cycle of a Servlet

- A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet
- init()
- service()
- destroy()
- doGet()
- doPost()



The init() Method

- The init method is called only once when the servlet is created
- When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.

```
public void init(ServletConfig config) throws ServletException
```

The service() Method

- The service() method is the main method to perform the actual task
- The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service
- The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

```
public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {
}
```

The destroy() Method

- The destroy() method is called only once at the end of the life cycle of a servlet.
- method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities
- After the destroy() method is called, the servlet object is marked for garbage collection.

```
public void destroy() {  
    // Finalization code...  
}
```

The HttpServlet class

- HttpServlet class servers client's HTTP request
- For each of the HTTP methods, GET, POST and other corresponding methods
 - doGet().... – Serves HTTP GET request
 - doPost()..- Servers HTTP POST request
 - The servelet usually must implement one of the first two methods or the service().. method

The HttpServletRequest Object

- Contains the request data from the client
 - HTTP request headers
 - Form data and Queary parameters
 - Other client data(cookies, path, etc.)

The HTTPServletRespond Object

- Encapsulate data and send back to the client
 - HTTP response headers(Content type)
 - Response Body

The most important servlet functionalities

- Retrieve the HTML Form parameters from the request(Both get and post parameters)

```
HttpServletRequest.getParameter(String)
```

- Retrieve a servlet initialization parameter

```
ServletConfig.getInitParameter()
```

- Retrieve HTTP request header information

```
HttpServletRequest.getHeader(String)
```

Building tool- Maven

Maven is really a process of applying patterns to a build infrastructure in order to provide a coherent view of software projects.

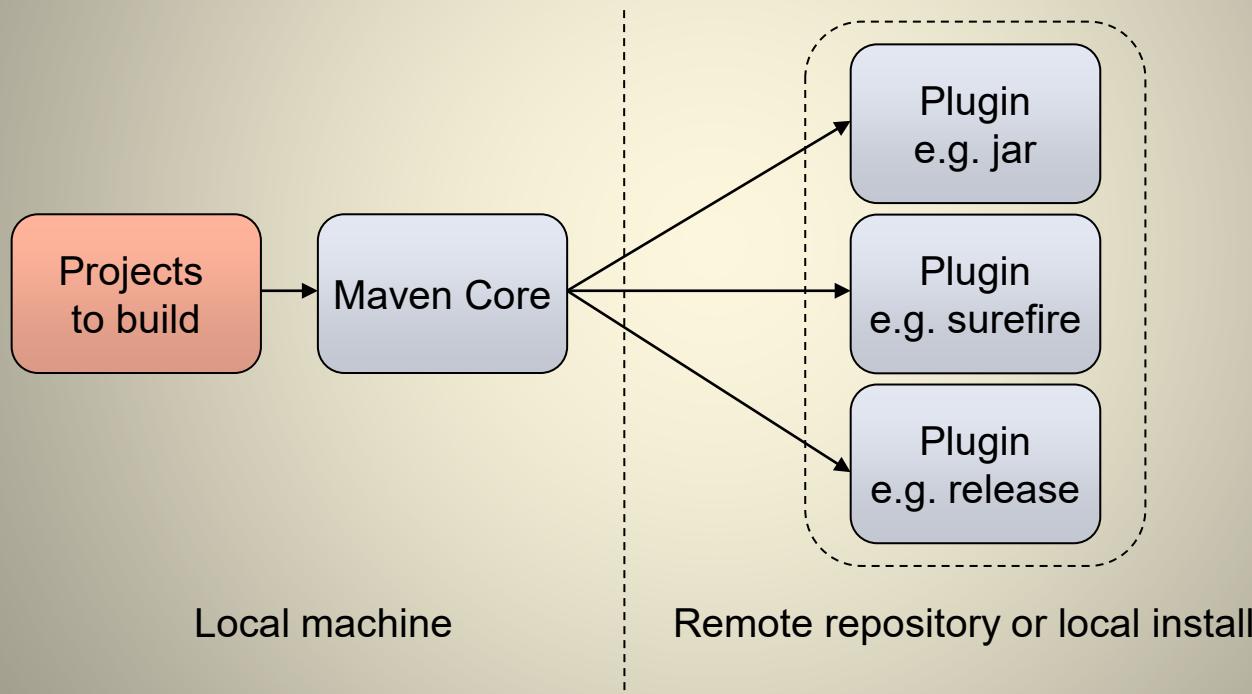
Provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies
- Software Configuration Management
- Releases
- Distribution

Why to use maven

- Make the development process visible or transparent
- Provide an easy way to see the health and status of a project
- Decreasing training time for new developers
- Bringing together the tools required in a uniform way
- Preventing inconsistent setups
- Providing a standard development infrastructure across projects
- Focus energy on writing applications

Maven Architecture



Common project metadata format

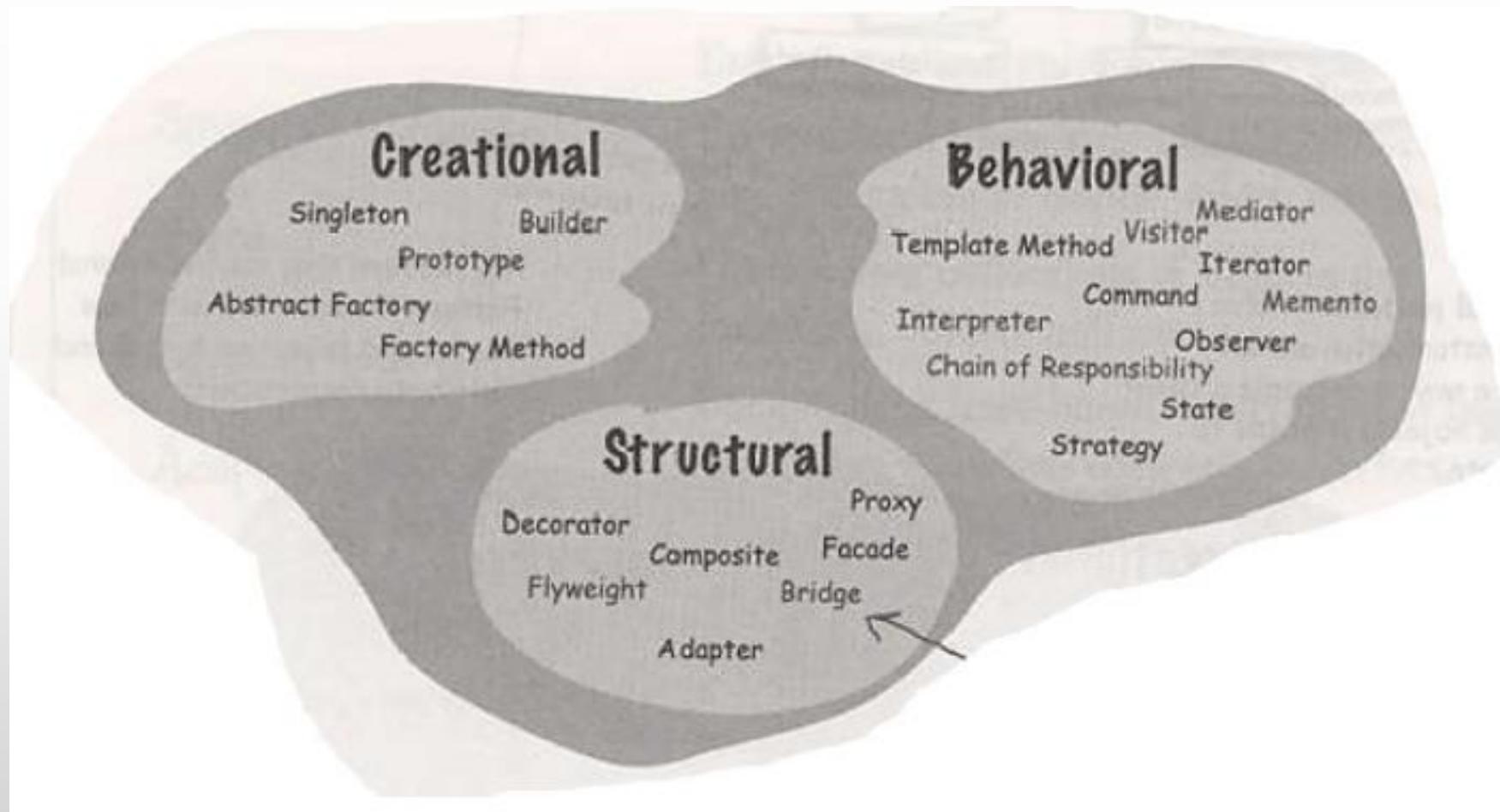
- POM = Project Object Model = pom.xml
- Contains metadata about the project
 - Location of directories, Developers/Contributors, Issue tracking system, Dependencies, Repositories to use, etc
- Example:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-core-api-container</artifactId>
  <name>Cargo Core Container API</name>
  <version>0.7-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies/>
  <build/>
  [...]
```

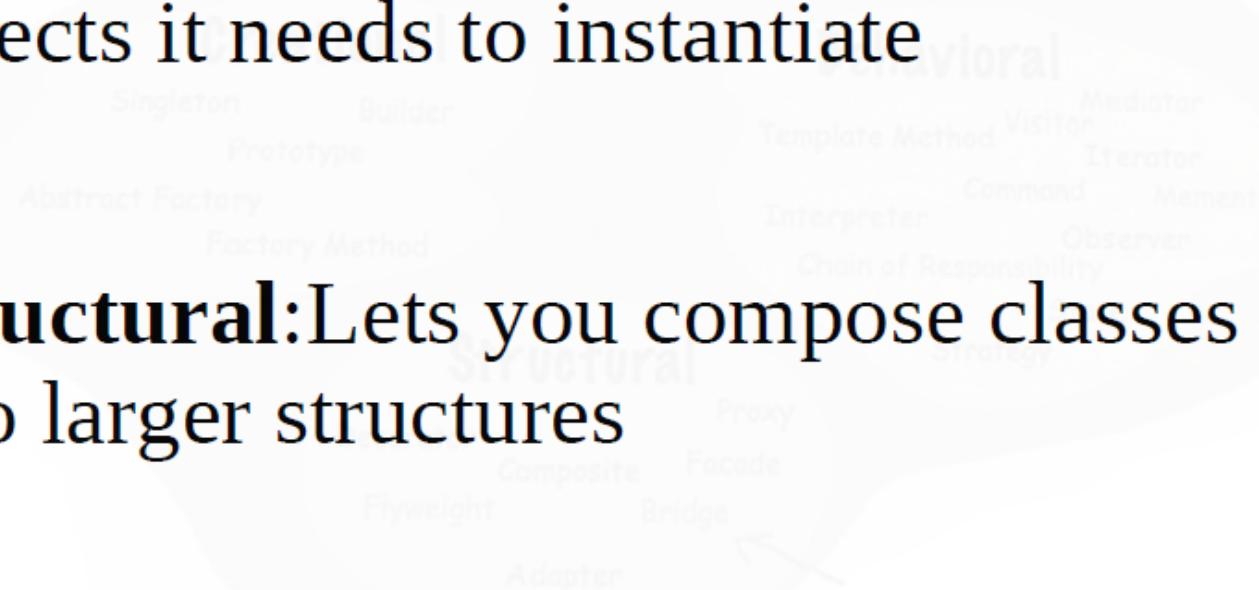
Design Patterns

Object Oriented Programming (OOP)
2nd Year – Semester 1
By Udara Samaratunge

Fundamental Design Patterns (Gang Of Four - GOF)



- **Creational:** Involve object initialization and provide a way to decouple client from the objects it needs to instantiate



- **Structural:** Lets you compose classes or objects into larger structures
- **Behavioral:** Concerned with how classes and objects interact or distribute responsibility

Gang of Four Patterns

Creational :

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

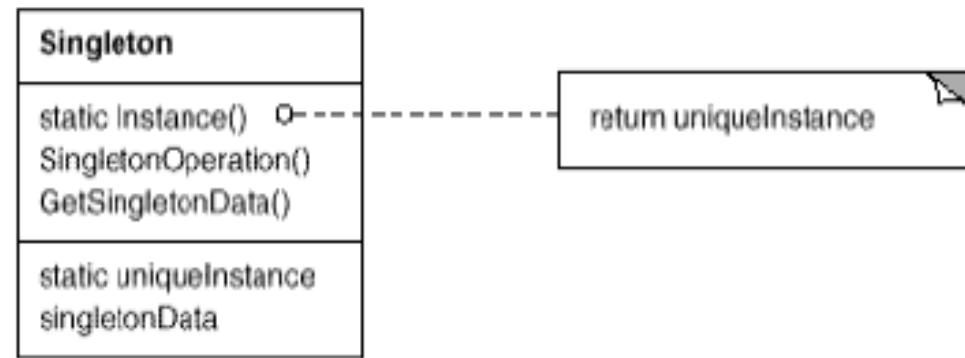
Behavioral :

- Strategy
- Observer
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Chain of Responsibility
- State
- Template Method
- Visitor

Structural :

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Singleton Pattern



Ensure a class only has one instance, and provide a global point of access to it

Singleton

```
public class Singleton {  
    private static Singleton uniqueinstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
  
        if (uniqueinstance == null) {  
            uniqueinstance = new Singleton();  
        }  
        return uniqueinstance;  
    }  
}
```

A static instance

Need to have a private constructor to block multiple instances

Lazy Loading

This ensures only one object instance is ever created.

However, this is good only for a single threaded application

Singleton

```
public class Singleton {  
    private static Singleton uniqueinstance;  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
  
        if (uniqueinstance == null) {  
            uniqueinstance = new Singleton();  
        }  
        return uniqueinstance;  
    }  
}
```

*This overcomes
the multiple
threading issue.*

However, the synchronization is bit expensive

This way is good if the performance is not an issue
By Udara Samaratunge

Singleton – with Double Check Lock

```
public class Singleton {  
    private volatile static Singleton uniqueinstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueinstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueinstance == null) {  
                    uniqueinstance = new Singleton();  
                }  
            }  
        }  
        return uniqueinstance;  
    }  
}
```

Double
Check
Locking

Here the object is created and synchronized at the first time only. If the Double Check Locking is not there, two threads can get into the synchronized block one after the other

This way is good if the application is keen on its performance

Thread-safe singleton output

```
public class TestThreadSingleton implements Runnable{  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
  
        new Thread(new TestThreadSingleton()).start();  
  
        for (int i = 0; i < 10; i++) {  
            Singleton.getInstance();  
            ThreadSafeSingleton.getInstance();  
        }  
    }  
  
    /**  
     * Invoke thread  
     */  
    public void run(){  
        for (int i = 0; i < 10; i++) {  
            Singleton.getInstance();  
            ThreadSafeSingleton.getInstance();  
        }  
    }  
}
```

```
<terminated> TestThreadSingleton [Java Application]  
singleton invocation  
Singleton invocation  
Object created for ThreadSafeSingleton.
```

Factory Pattern

The Factory Method

The factory method pattern encapsulates the object creation by letting subclasses to decide what objects to create

PizzaStore is now abstract (see why below).

```
public abstract class PizzaStore {  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
  
        return pizza;  
    }  
  
    abstract createPizza(String type);  
}
```

Our "factory method" is now abstract in PizzaStore.

Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

The Creator Classes

```
public abstract class PizzaStore {  
    abstract Pizza createPizza(String item);  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza = createPizza(type);  
        System.out.println("--- Making a " + pizza.getName() + " ---");  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}  
  
public class ChicagoPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new ChicagoStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new ChicagoStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new ChicagoStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new ChicagoStylePepperoniPizza();  
        } else return null;  
    }  
}  
  
public class NYPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new NYStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new NYStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new NYStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new NYStylePepperoniPizza();  
        } else return null;  
    }  
}
```

Factory objects are created through INHERITANCE

This is the "Factory Method"

Simple Factory Vs Factory Method

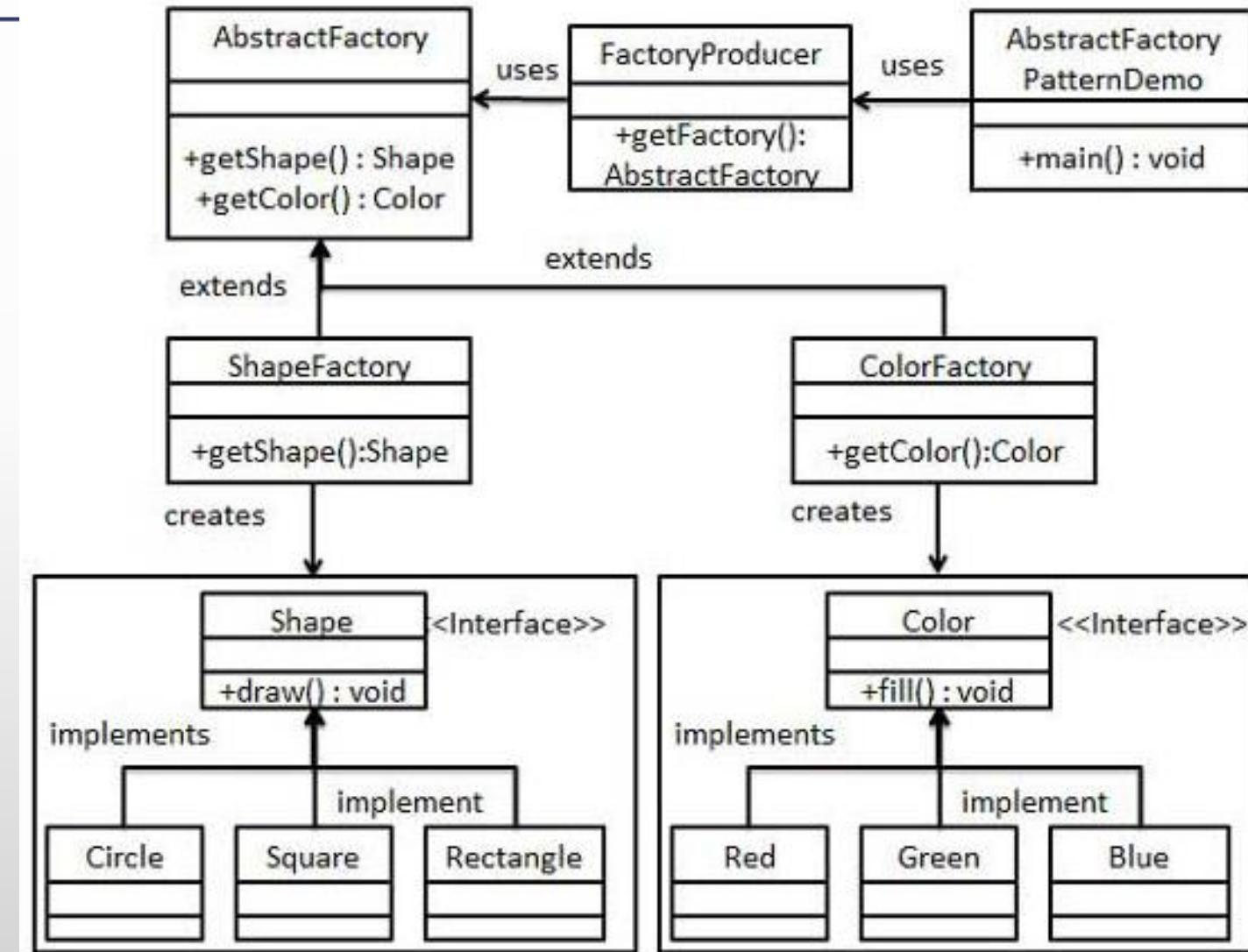
Simple Factory

-  Does not let you vary the product implementations being created

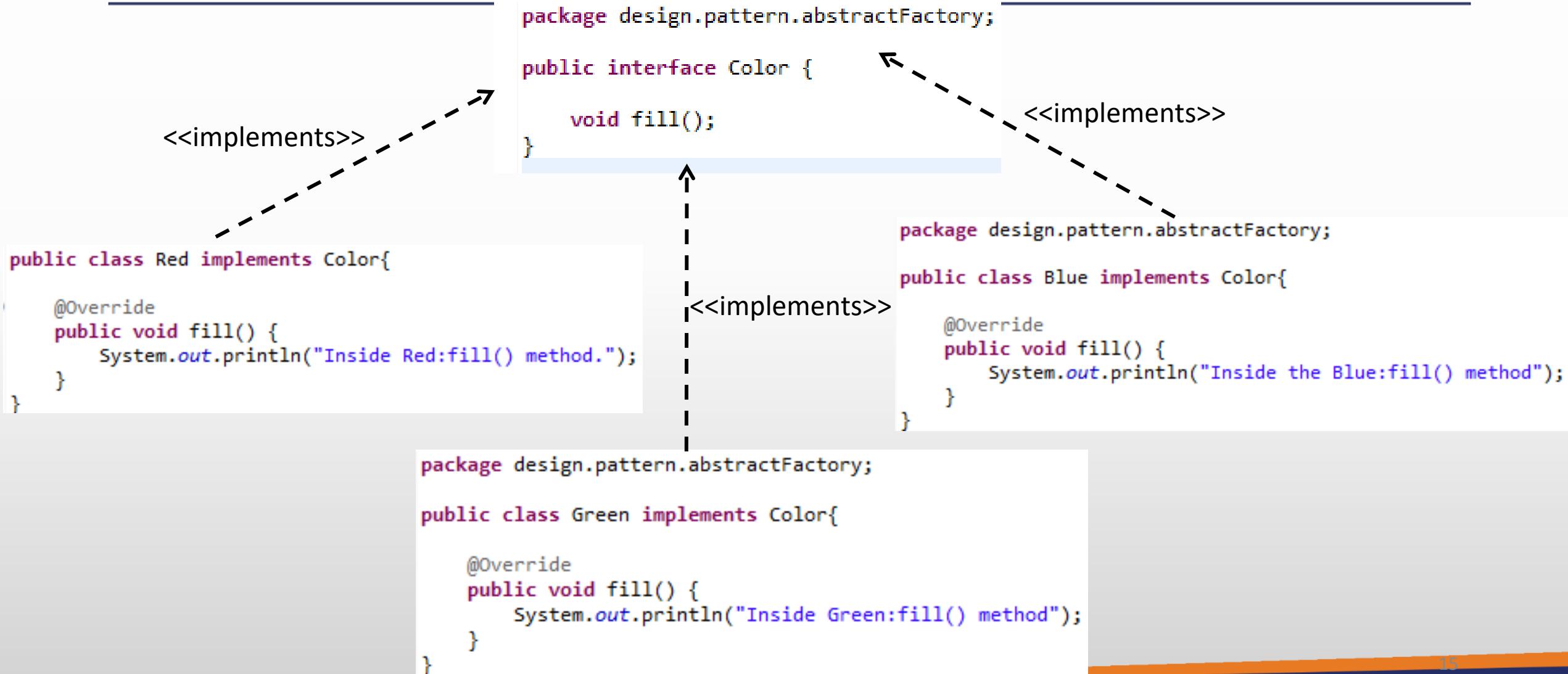
Factory Method

-  Creates a framework that lets the sub classes decides which product implementation will be used

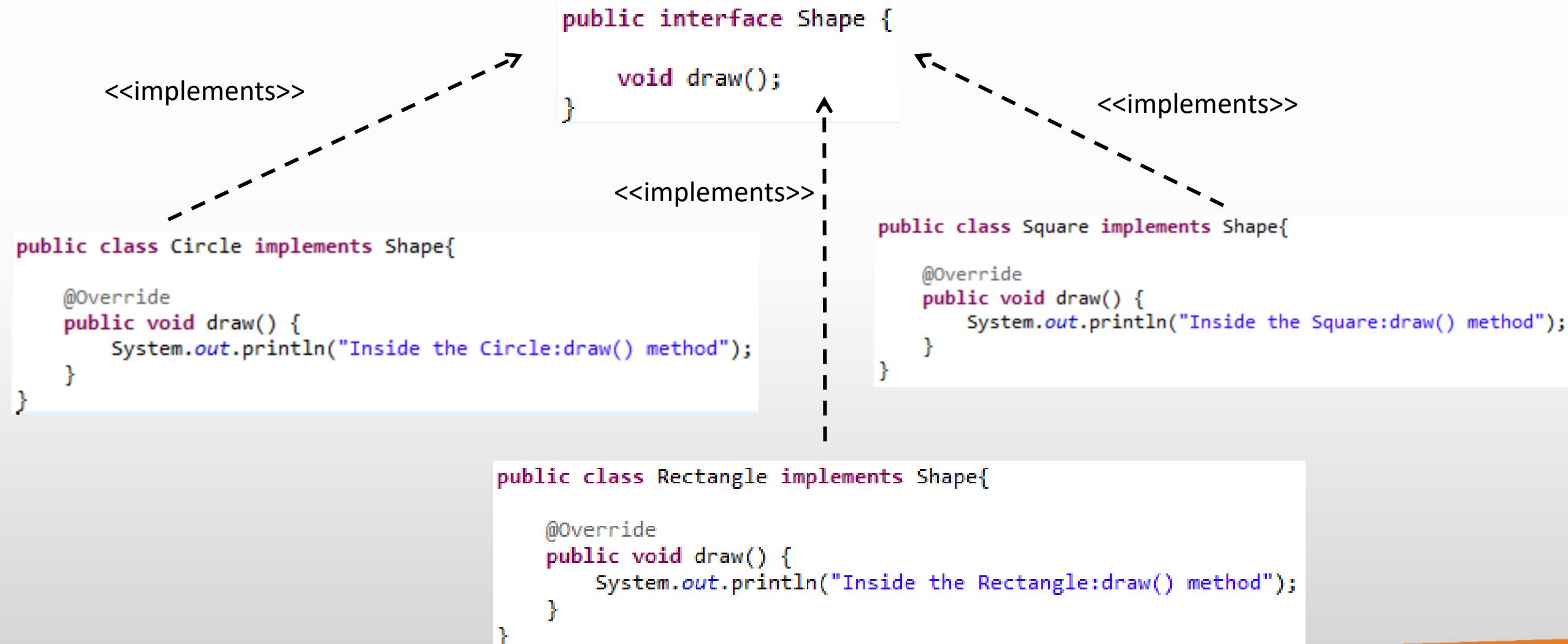
Abstract Factory Pattern



Abstract Factory Pattern



Abstract Factory Pattern



Abstract Factory Pattern

```
public class ColorFactory extends AbstractFactory{  
  
    @Override  
    public Color getColor(String color) {  
  
        if(color.equalsIgnoreCase("RED")){  
            return new Red();  
        }  
        else if(color.equalsIgnoreCase("GREEN")){  
            return new Green();  
        }  
        else if(color.equalsIgnoreCase("BLUE")){  
            return new Blue();  
        }  
        else{  
            return null;  
        }  
    }  
  
    @Override  
    public Shape getShape(String type) {  
        return null;  
    }  
}
```

```
public class FactoryProducer {  
  
    public static AbstractFactory getFactory(String choice){  
  
        if(choice.equalsIgnoreCase("SHAPE")){  
            return new ShapeFactory();  
        }  
        else if(choice.equalsIgnoreCase("COLOR")){  
            return new ColorFactory();  
        }  
        else{  
            return null;  
        }  
    }  
}
```

```
public class ShapeFactory extends AbstractFactory{  
  
    @Override  
    public Shape getShape(String shapeType) {  
  
        if(shapeType == null){  
            return null;  
        }  
        else if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
        }  
        else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
        }  
        else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        else{  
            return null;  
        }  
    }  
  
    @Override  
    public Color getColor(String type) {  
        return null;  
    }  
}
```

Abstract Factory Pattern

```
public class ColorFactory extends AbstractFactory{
    @Override
    public Color getColor(String color) {
        if(color.equalsIgnoreCase("RED")){
            return new Red();
        }
        else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }
        else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();
        }
        else{
            return null;
        }
    }

    @Override
    public Shape getShape(String type) {
        return null;
    }
}
```

```
public abstract class AbstractFactory {
    public abstract Color getColor(String type);
    public abstract Shape getShape(String type);
}
```

uses

```
class FactoryProducer {
    public static AbstractFactory getFactory(String choice){
        if(choice.equalsIgnoreCase("SHAPE")){
            return new ShapeFactory();
        }
        else if(choice.equalsIgnoreCase("COLOR")){
            return new ColorFactory();
        }
        else{
            return null;
        }
    }
}
```

```
public class ShapeFactory extends AbstractFactory{
    @Override
    public Shape getShape(String shapeType) {
        if(shapeType == null){
            return null;
        }
        else if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        }
        else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        }
        else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        else{
            return null;
        }
    }

    @Override
    public Color getColor(String type) {
        return null;
    }
}
```

Abstract Factory Pattern

```
package design.pattern.abstractFactory;

public class AbstractFactoryPatternDemo {

    private static final String SHAPE = "SHAPE";
    private static final String CIRCLE = "CIRCLE";
    private static final String RECTANGLE = "RECTANGLE";
    private static final String SQUARE = "SQUARE";

    private static final String COLOR = "COLOR";
    private static final String RED = "RED";
    private static final String GREEN = "GREEN";
    private static final String BLUE = "BLUE";

    public static void main(String[] args) {

        AbstractFactory shapeFactory = FactoryProducer.getFactory(SHAPE);
        Shape shape = shapeFactory.getShape(CIRCLE);
        shape.draw();

        FactoryProducer.getFactory(SHAPE).getShape(RECTANGLE).draw();
        FactoryProducer.getFactory(SHAPE).getShape(SQUARE).draw();

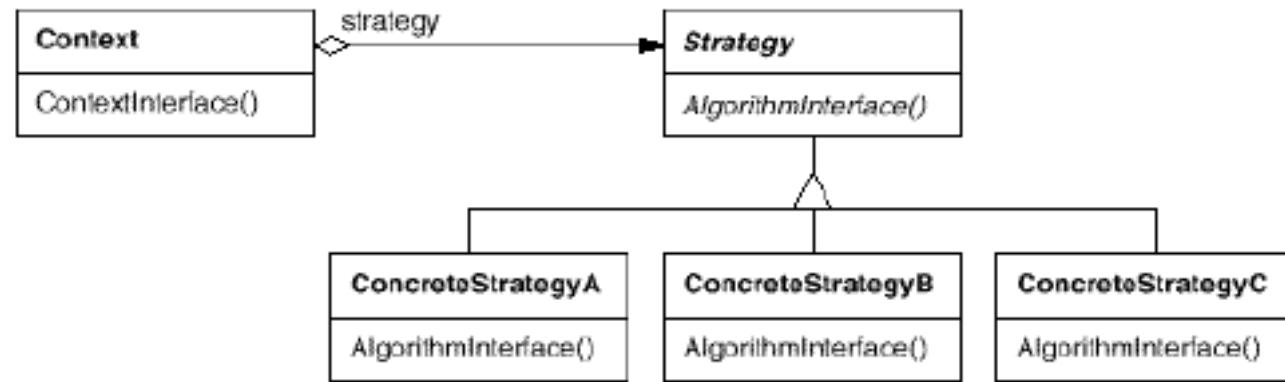
        FactoryProducer.getFactory(COLOR).getColor(RED).fill();
        FactoryProducer.getFactory(COLOR).getColor(GREEN).fill();
        FactoryProducer.getFactory(COLOR).getColor(BLUE).fill();
    }
}
```

Problems Console @ Javadoc Declaration Search Progress Cross Ref

```
<terminated> AbstractFactoryPatternDemo [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\ja
Inside the Circle:draw() method
Inside the Rectangle:draw() method
Inside the Square:draw() method
Inside Red:fill() method.
Inside Green:fill() method
Inside the Blue:fill() method
```

Strategy Pattern

Strategy Pattern



Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

By Udara Samaratunge

21

Design Principles covered - (3)

① *Design Principle 1*

Identify the aspects of your application that vary and separate them from what stays the same

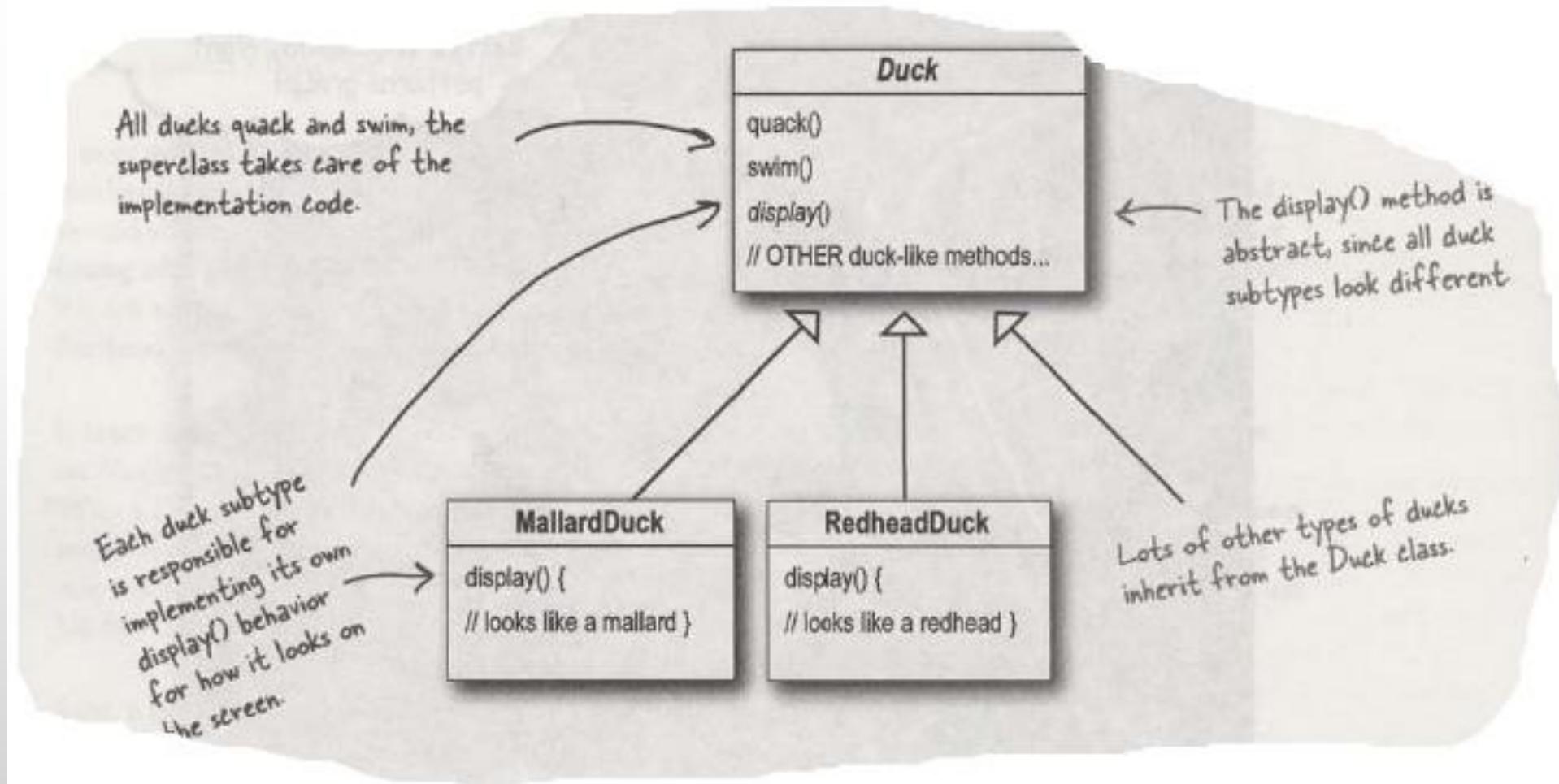
② *Design Principle 2*

Program to an interface not to an implementation

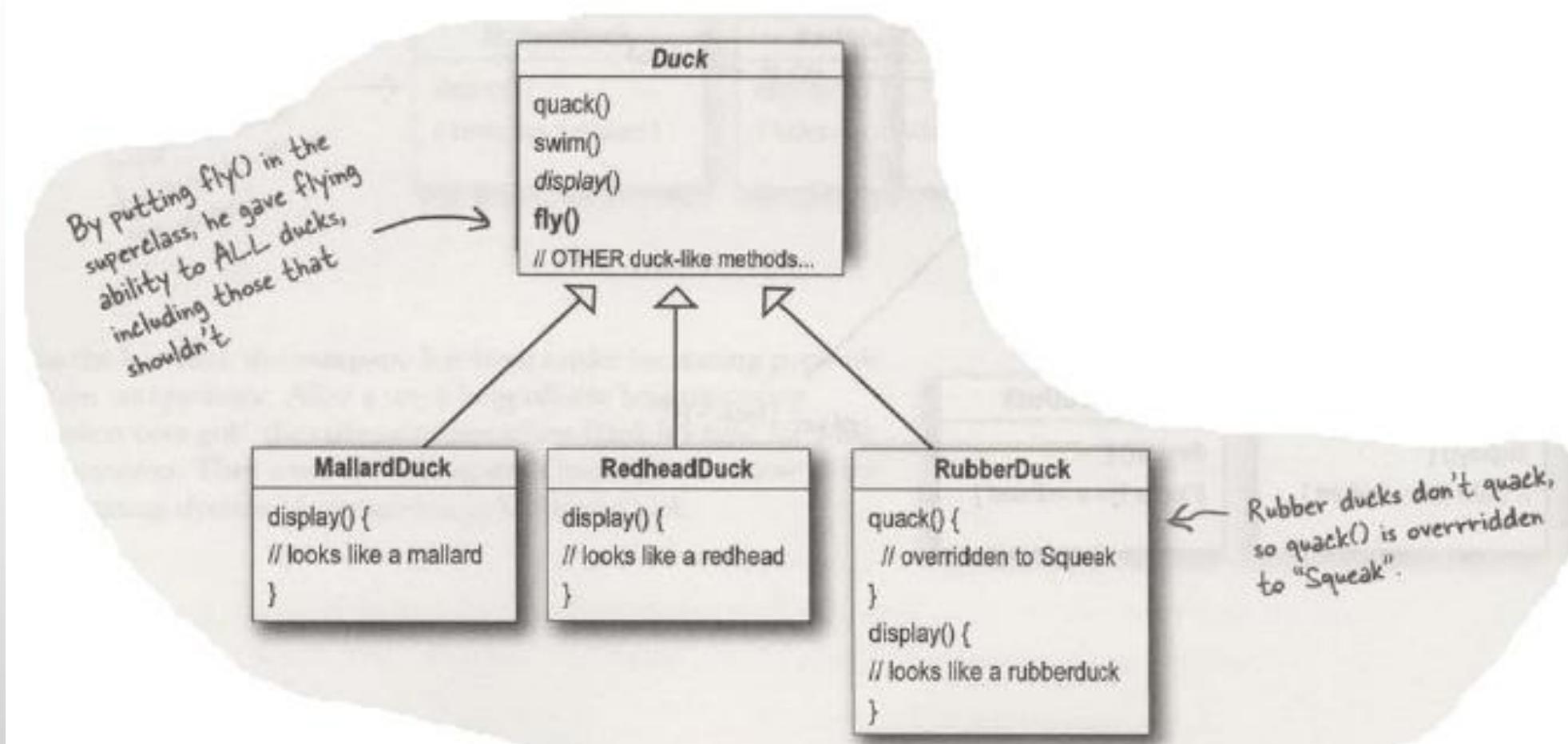
③ *Design Principle 3*

Favor composition over inheritance

The Duck Simulation

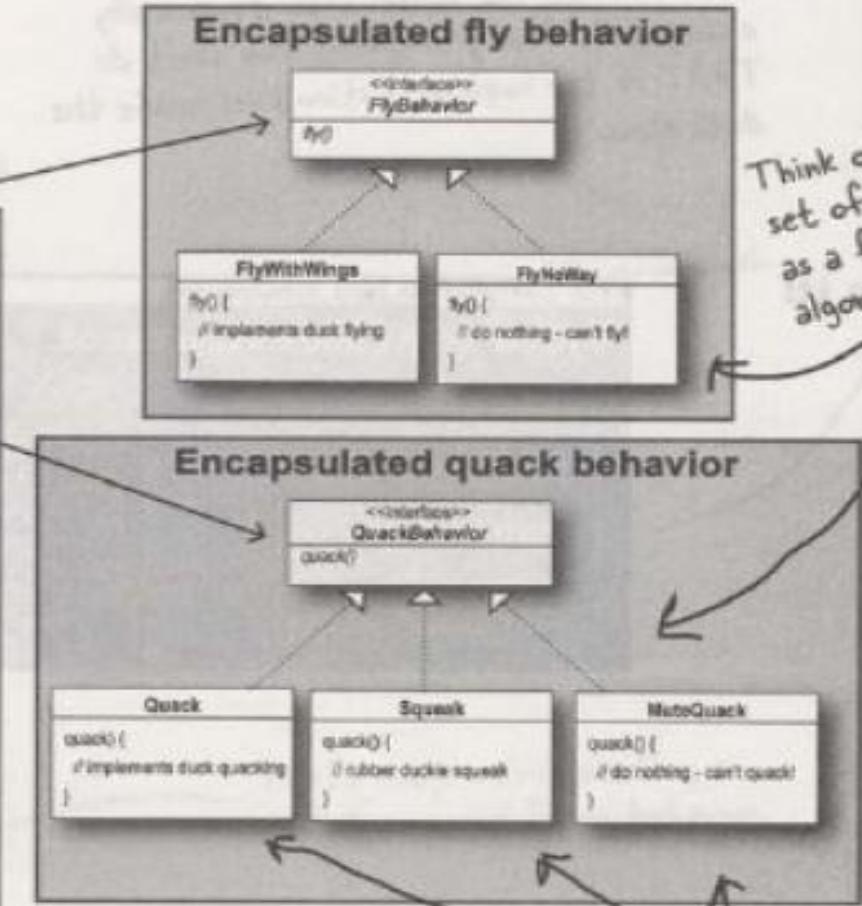
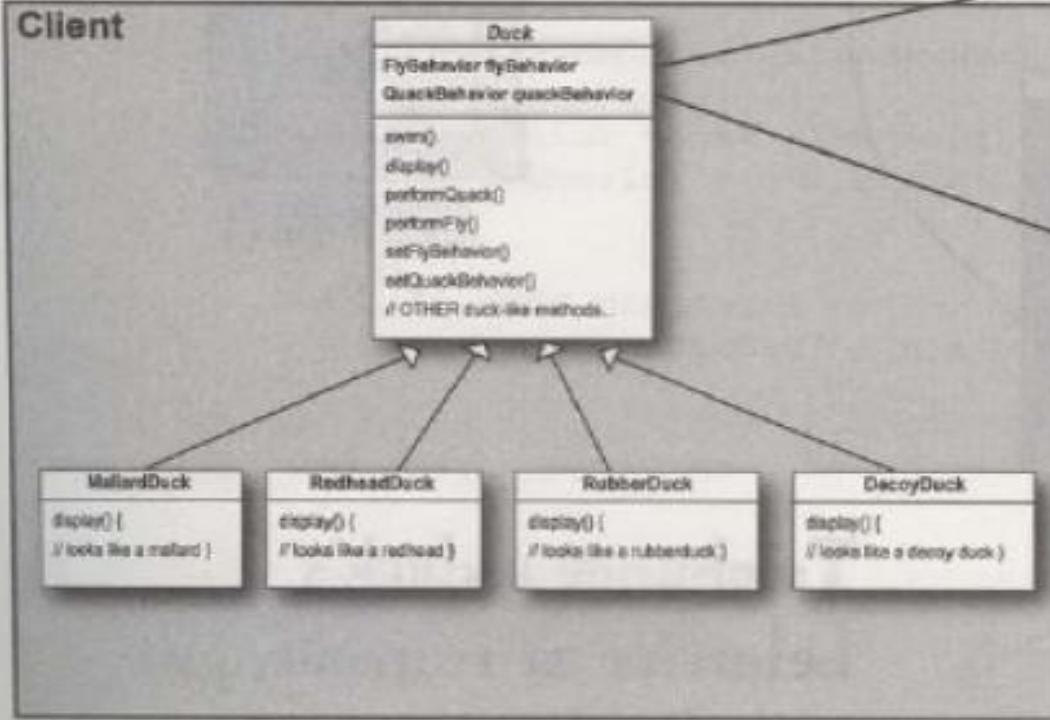


How about Inheritance?

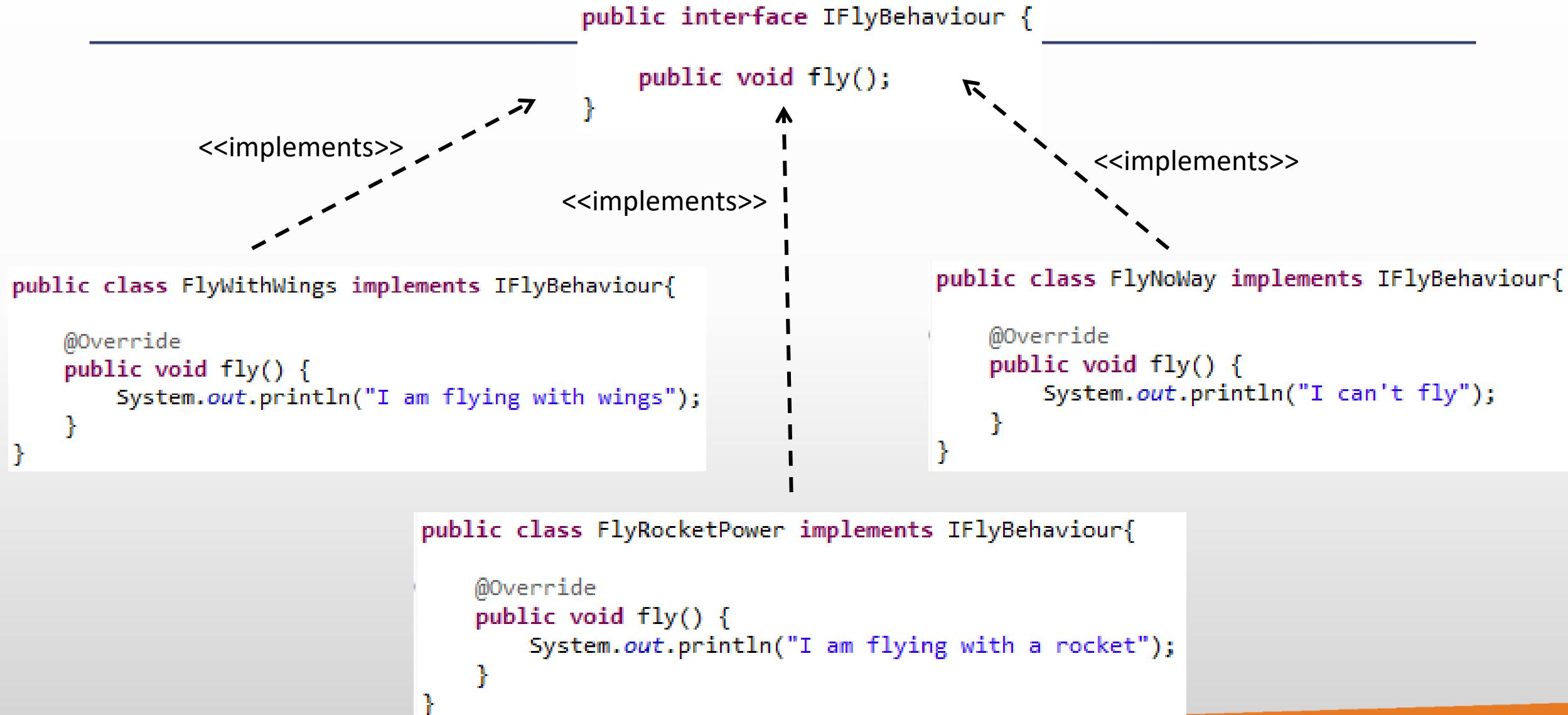


Setting the behavior dynamically

Client makes use of an encapsulated family of algorithms for both flying and quacking.



Strategy Pattern Implementation



Strategy Pattern Implementation

```
public interface IQuackBehaviour {  
    public void quack();  
}
```

<<implements>>

```
public class Quack implements IQuackBehaviour{  
    @Override  
    public void quack() {  
        System.out.println("Quack..Quack...");  
    }  
}
```

<<implements>>

```
public class ModelQuack implements IQuackBehaviour{  
    @Override  
    public void quack() {  
        System.out.println("Quack Model duck");  
    }  
}
```

```
public abstract class Duck {
    IFlyBehaviour flyBehaviour;
    IQuackBehaviour quackBehaviour;

    public abstract void display();

    public void performFly(){
        flyBehaviour.fly();
    }

    public void performQuack(){
        quackBehaviour.quack();
    }

    public void swim(){
        System.out.println("All ducks float even Decoy");
    }

    public void setFlyBehaviour(IFlyBehaviour flyBehaviour) {
        this.flyBehaviour = flyBehaviour;
    }

    public void setQuackBehaviour(IQuackBehaviour quackBehaviour) {
        this.quackBehaviour = quackBehaviour;
    }
}
```

```
public interface IFlyBehaviour {
    public void fly();
}

public interface IQuackBehaviour {
    public void quack();
}
```

```
public class ModelDuck extends Duck{
    public ModelDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyNoWay();
    }

    @Override
    public void display() {
        System.out.println("I am a model Duck");
    }
}
```

```
public class MollardDuck extends Duck{
    public MollardDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyWithWings();
    }

    @Override
    public void display() {
        System.out.println("I am a real Mollard Duck.");
    }
}
```

By Udara Samaratunge

Strategy Pattern Implementation

```
package design.pattern.strategy;

public class TestDuck {

    /**
     * @param args
     */
    public static void main(String[] args) {

        System.out.println("Start Mollard Duck");
        System.out.println("=====");
        Duck mollard = new MollardDuck();
        mollard.performFly();
        mollard.performQuack();

        System.out.println("Start Model Duck");
        System.out.println("=====");
        Duck model = new ModelDuck();

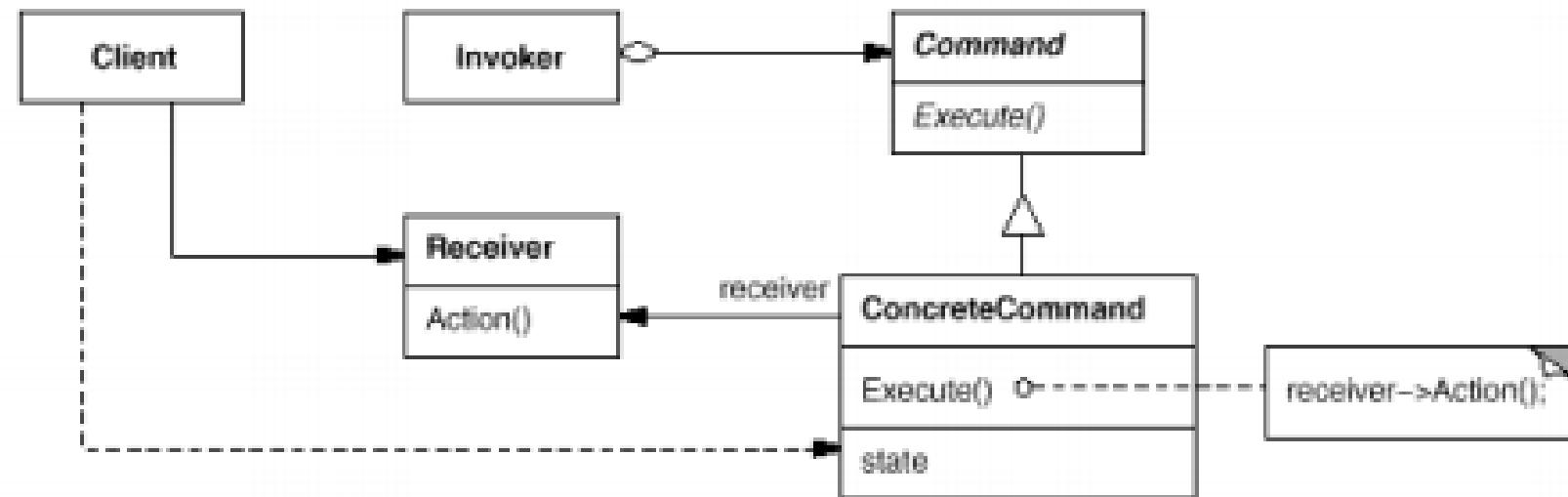
        model.performFly();
        model.setFlyBehaviour(new FlyRocketPower());
        model.performFly();

        model.performQuack();
        model.setQuackBehaviour(new ModelQuack());
        model.performQuack();

    }
}
```

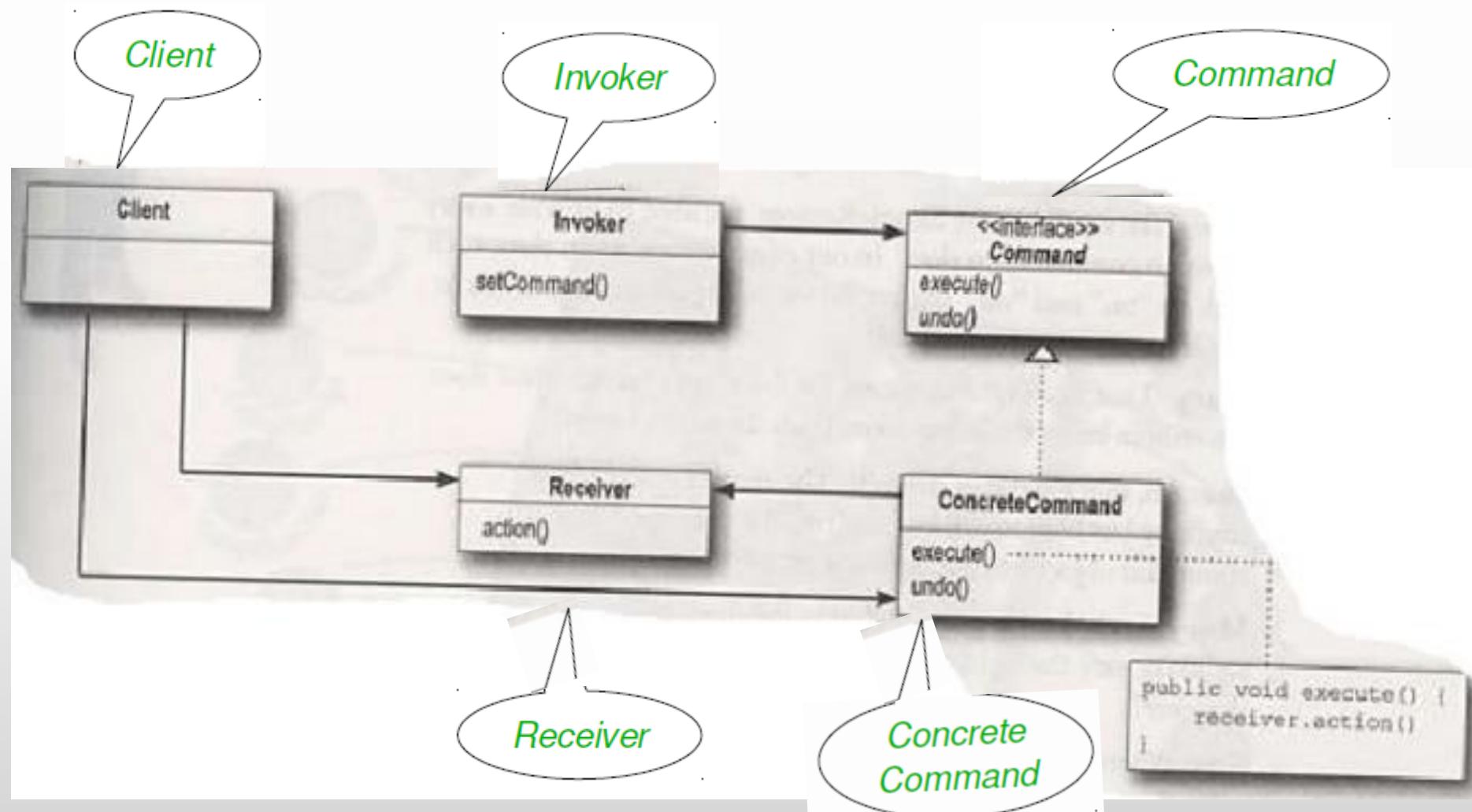
```
<terminated> TestDuck [Java Application] C:\P
Start Mollard Duck
=====
I am flying with wings
Quack..Quack...
Start Model Duck
=====
I can't fly
I am flying with a rocket
Quack..Quack...
Quack Model duck
```

Command Pattern

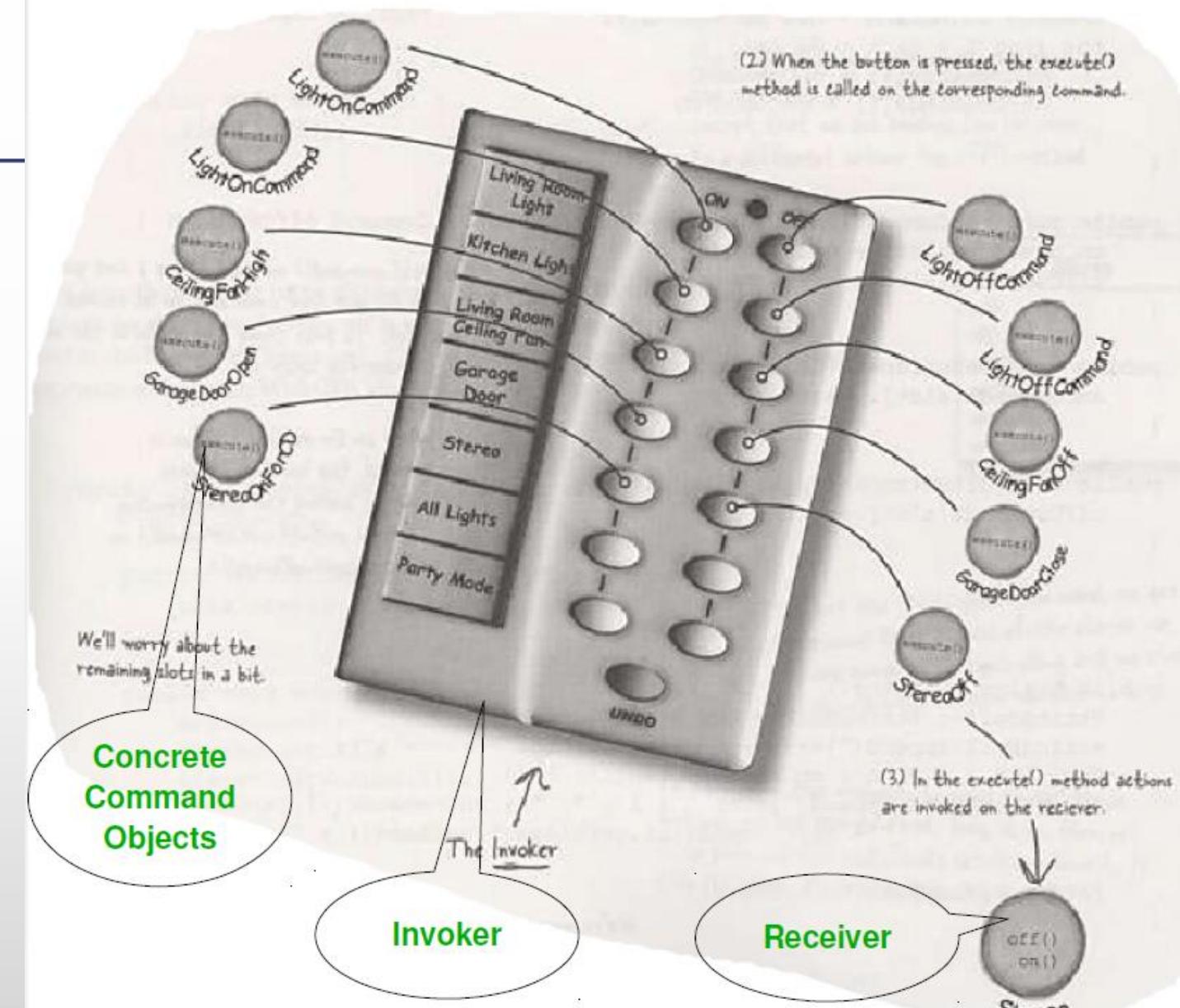


Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

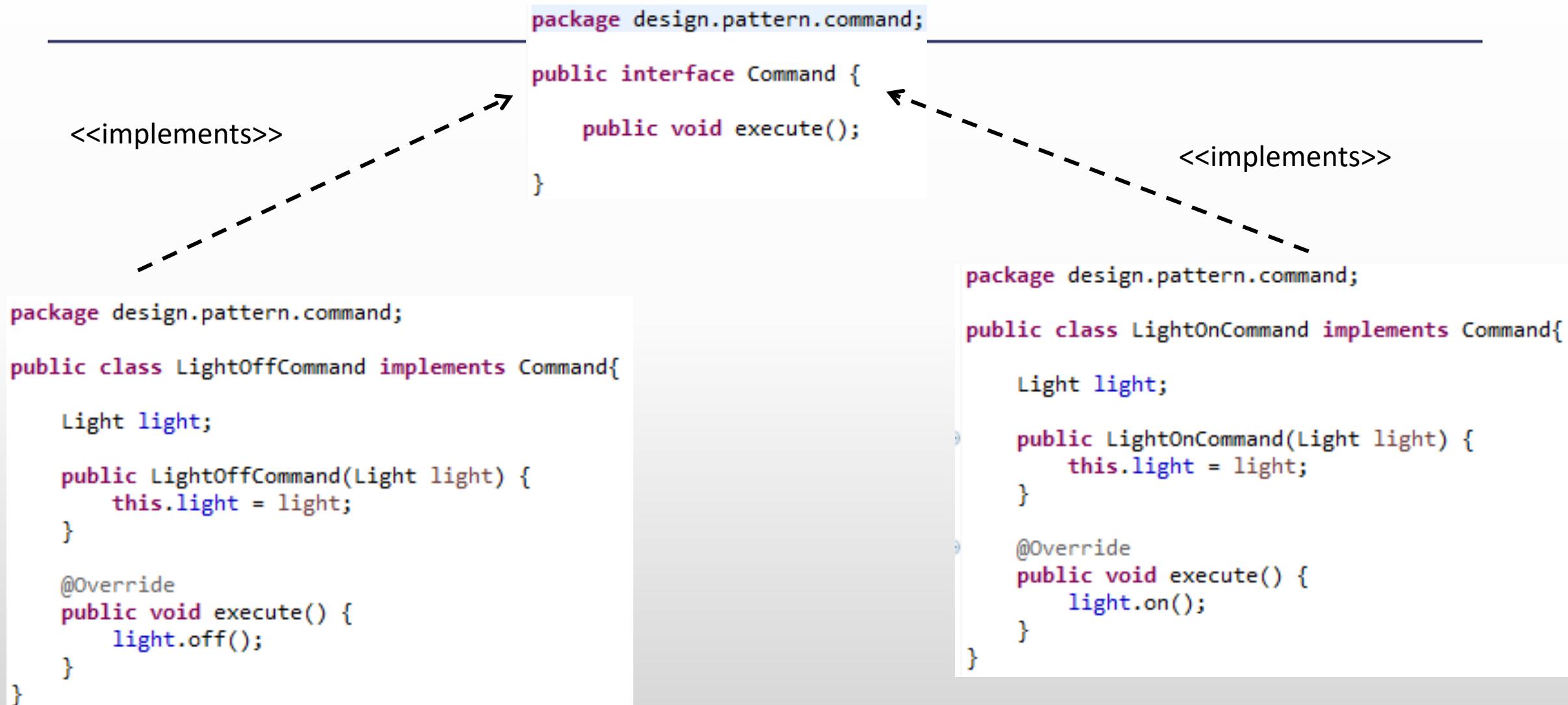
Command Pattern



Command Pattern



Command & Concrete Command



Receiver

```
public class Light {  
  
    private String location;  
  
    public Light(String location) {  
        this.location = location;  
    }  
  
    public void on(){  
        System.out.println(location + " light is on.");  
    }  
  
    public void off(){  
        System.out.println(location + " light is off.");  
    }  
}
```

Command Pattern

```

package design.pattern.command;

public class RemoteController {
    Command [] onCommands;
    Command [] offCommands;

    public RemoteController() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        for (int i = 0; i < 7; i++) {
            onCommands[i] = null;
            offCommands[i] = null;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand){
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot){
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot){
        offCommands[slot].execute();
    }
}

```

```

package design.pattern.command;

public interface Command {
    public void execute();
}

public class Light {
    private String location;

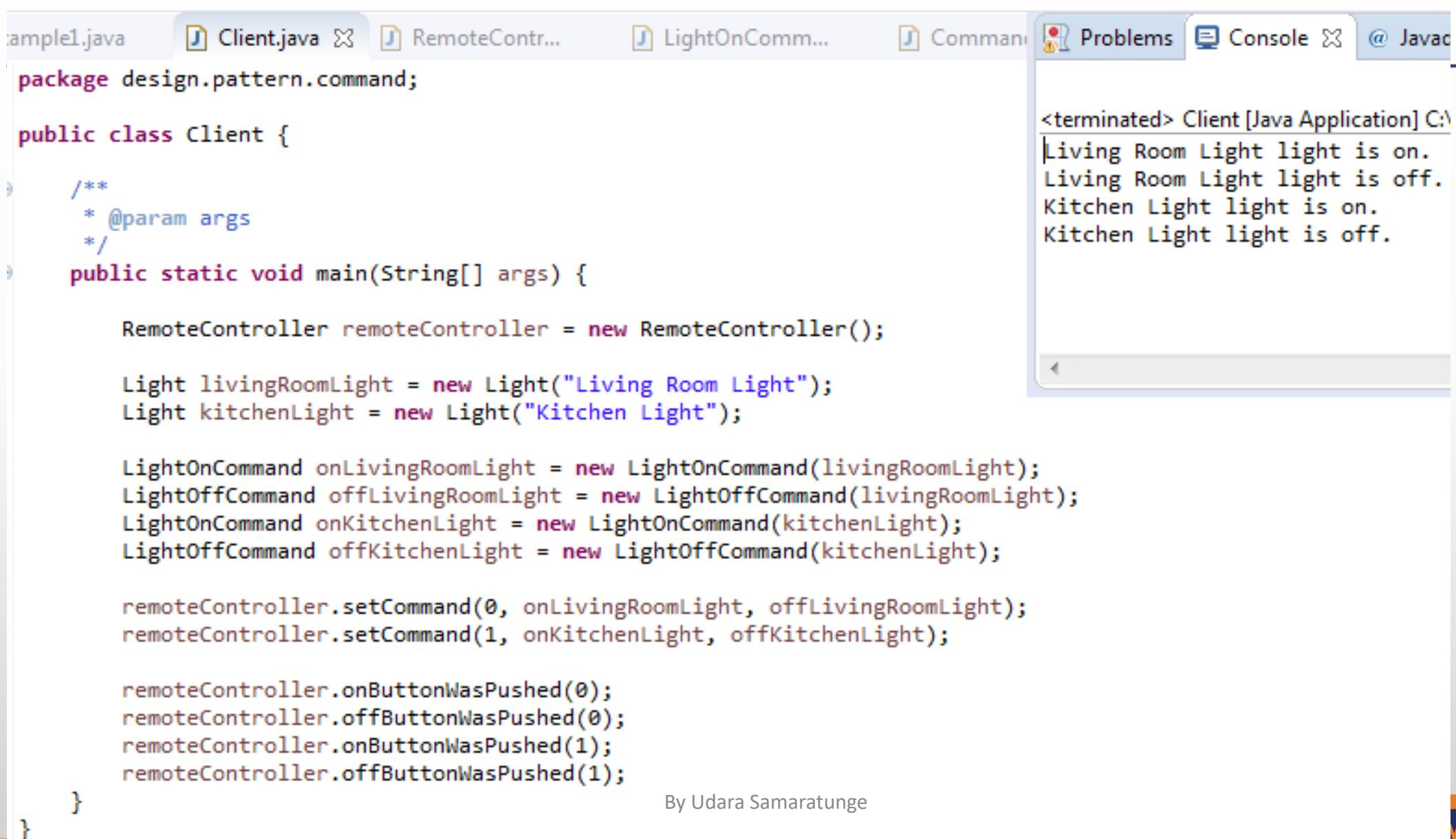
    public Light(String location) {
        this.location = location;
    }

    public void on(){
        System.out.println(location + " light is on.");
    }

    public void off(){
        System.out.println(location + " light is off.");
    }
}

```

Invoker



The screenshot shows the Eclipse IDE interface with several tabs at the top: sample1.java, Client.java, RemoteContr..., LightOnComm..., Command..., Problems (selected), Console, and JavaDoc.

The code in Client.java demonstrates the Invoker pattern:

```
package design.pattern.command;

public class Client {
    /**
     * @param args
     */
    public static void main(String[] args) {
        RemoteController remoteController = new RemoteController();

        Light livingRoomLight = new Light("Living Room Light");
        Light kitchenLight = new Light("Kitchen Light");

        LightOnCommand onLivingRoomLight = new LightOnCommand(livingRoomLight);
        LightOffCommand offLivingRoomLight = new LightOffCommand(livingRoomLight);
        LightOnCommand onKitchenLight = new LightOnCommand(kitchenLight);
        LightOffCommand offKitchenLight = new LightOffCommand(kitchenLight);

        remoteController.setCommand(0, onLivingRoomLight, offLivingRoomLight);
        remoteController.setCommand(1, onKitchenLight, offKitchenLight);

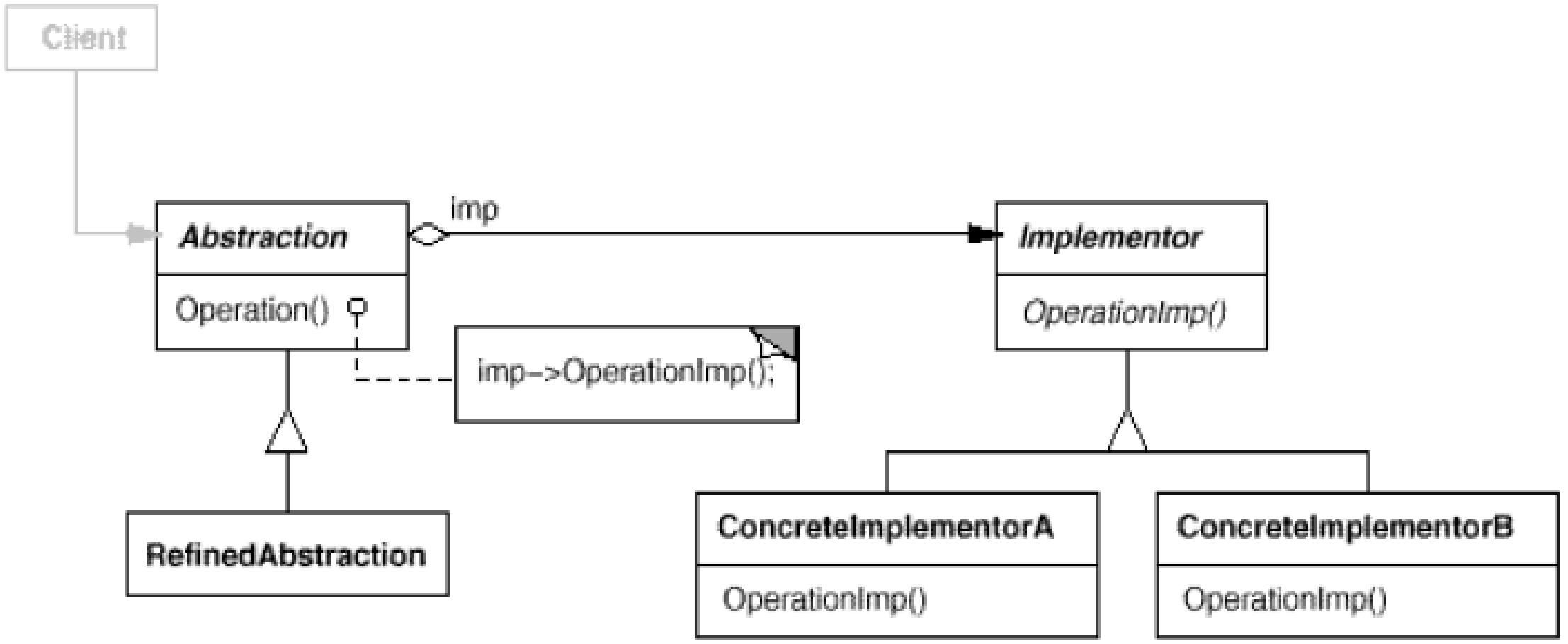
        remoteController.onButtonWasPushed(0);
        remoteController.offButtonWasPushed(0);
        remoteController.onButtonWasPushed(1);
        remoteController.offButtonWasPushed(1);
    }
}
```

The output in the Console tab shows the execution results:

```
<terminated> Client [Java Application] C:\Java\workspace\design.pattern.command\bin
Living Room Light light is on.
Living Room Light light is off.
Kitchen Light light is on.
Kitchen Light light is off.
```

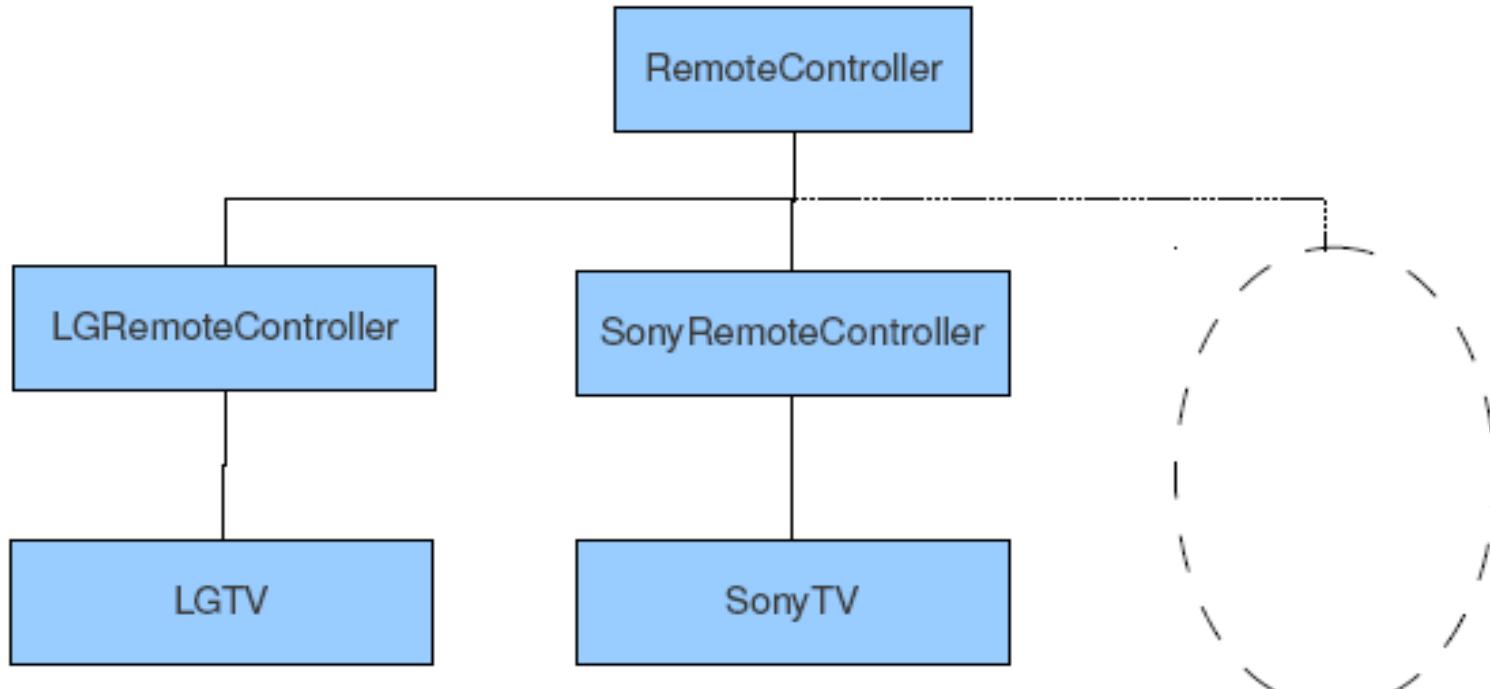
Bridge Pattern

Bridge Pattern



Example for Bridge Pattern

- There are two brands of TVs (Sony and LG) in your living room. So there are two remote controllers for each one. (See below diagram) Just assume a single remote controller can be used to **switch on, switch off** and **tune channels** of both TVs. Think of a design pattern that can solve this.



Bridge Pattern

```
public interface TV {  
    void on();  
    void off();  
    void tune(int channel);      <-- -----> <<implements>>  
}  
  
public class LGTV implements TV{  
  
    @Override  
    public void on() {  
        System.out.println("Switch on LG TV");  
    }  
  
    @Override  
    public void off() {  
        System.out.println("Switch off LG TV");  
    }  
  
    @Override  
    public void tune(int channel) {  
        System.out.println("Switch on channel in LG TV is:  
    }  
}
```

```
public class SonyTV implements TV{  
  
    @Override  
    public void on() {  
        System.out.println("Switch on Sony TV");  
    }  
  
    @Override  
    public void off() {  
        System.out.println("Switch off Sony TV");  
    }  
  
    @Override  
    public void tune(int channel) {  
        System.out.println("Switch on channel in Sony TV is: " + channel);  
    }  
}
```

```

public interface RemoteController {
    void on();
    void off();
    void tune(int channel);
}

public class RemoteControllerImpl implements RemoteController{
    TV tv;

    public RemoteControllerImpl(TV tv) {
        this.tv = tv;
    }

    @Override
    public void on() {
        tv.on();
    }

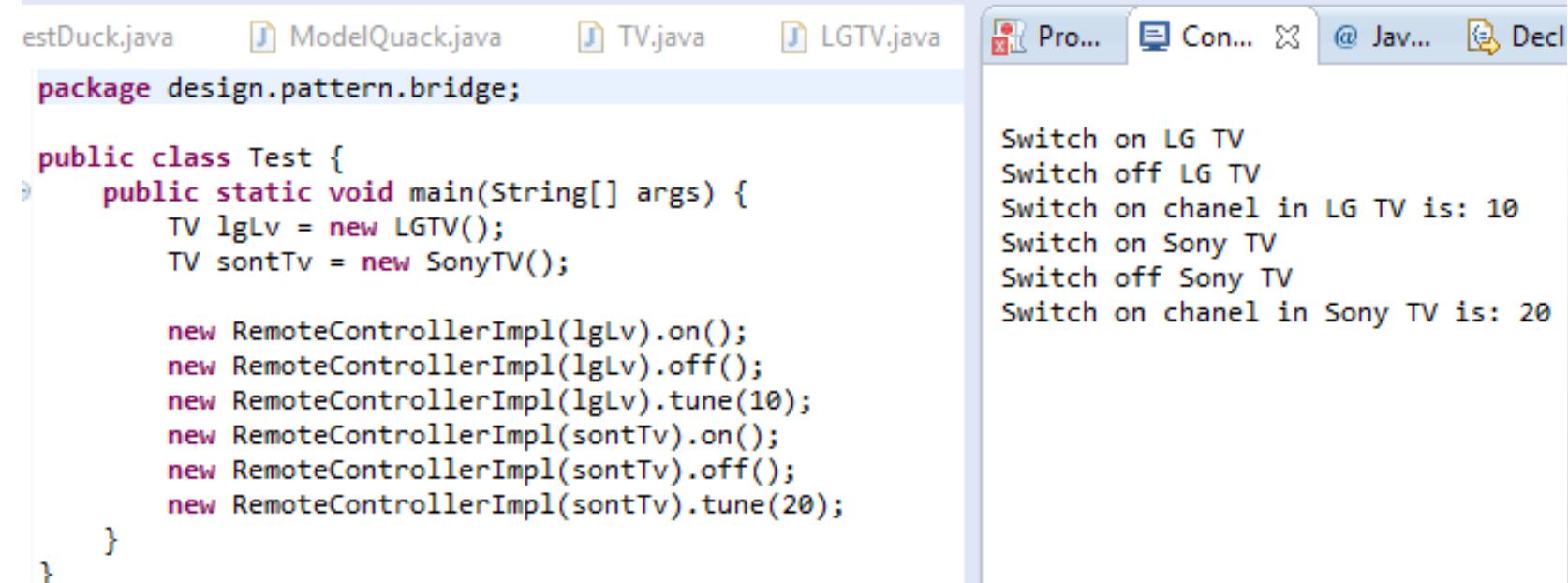
    @Override
    public void off() {
        tv.off();
    }

    @Override
    public void tune(int channel) {
        tv.tune(channel);
    }
}

```

↑
| <<implements>>
|

Bridge Pattern



```

package design.pattern.bridge;

public class Test {
    public static void main(String[] args) {
        TV lgLv = new LGTV();
        TV sonyTv = new SonyTV();

        new RemoteControllerImpl(lgLv).on();
        new RemoteControllerImpl(lgLv).off();
        new RemoteControllerImpl(lgLv).tune(10);
        new RemoteControllerImpl(sonyTv).on();
        new RemoteControllerImpl(sonyTv).off();
        new RemoteControllerImpl(sonyTv).tune(20);
    }
}

```

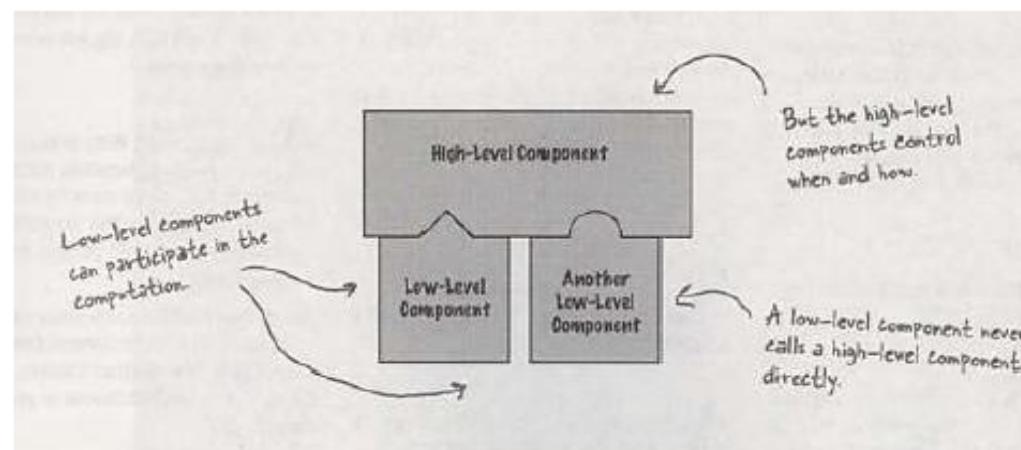
Template method pattern

Design Principles covered - (1)

④ *Design Principle*

“The Hollywood Principle” - Don’t call us, we will call you

Allows low level components to hook themselves into a system. But the high-level components determine when they are needed and how.



 The *Template Method*,

Is a method, which serves as a **template** for an algorithm

-  In the template,
 -  Each step of the algorithm is represented by a method
(These are called as “hooks”)
 -  Some methods are handled by this class.
 -  Some methods are handled by the sub class.
 -  The methods, that need to be supplied by a subclass are declared *abstract*

Template method pattern

The template method defines the steps of an algorithm and follows subclasses to provide the implementation for one more steps

Template method pattern

An Example: Servlets

- ⌚ The servlet container invokes our servlet code
- ⌚ HttpServlet defines a *Template Method service()*, which takes care of general purpose handling of HTTP requests by calling **doGet()** and **doPost()** methods
- ⌚ We can extend the HttpServlet by overriding the steps of the algorithm, **doGet()** and **doPost()** methods to provide meaningful results

An Example: Servlets

Servlet Containers Hollywood Principle

Don't call me I will call you (servlet), whenever I hear from a browser

Servlet's Template Method

*Let me have the control of the algorithm and let me deal with HTTP.
You (Developer) just respond with some meaningful action when I
call your methods*

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < (lastModified / 1000 * 1000)) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }
    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);

    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);

    } else if (method.equals(METHOD_PUT)) {
        doPut(req, resp);
```

Template
Method

```
public abstract class CaffeineBeverage {  
  
    void final prepareRecipe() {  
  
        boilWater();  
  
        brew();  
  
        pourInCup();  
  
        addCondiments();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        // implementation  
    }  
  
    void pourInCup() {  
        // implementation  
    }  
}
```

prepareRecipe() is our template method
Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

Template method pattern

```
public abstract class Beverage {  
  
    final void prepareRecepie(){  
        boilWater();  
        brew();  
        addCondiments();  
        pourInCup();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater(){  
        System.out.println("Boiling water.");  
    }  
  
    void pourInCup(){  
        System.out.println("Pour into cup.");  
    }  
}
```

```
public class Tea extends Beverage {  
  
    @Override  
    void brew() {  
        System.out.println("Steeping the Tea.");  
    }  
  
    @Override  
    void addCondiments() {  
        System.out.println("Adding Lemon.");  
    }  
}
```

```
public class Coffie extends Beverage {  
  
    @Override  
    void addCondiments() {  
        System.out.println("Add sugar and milk.");  
    }  
  
    @Override  
    void brew() {  
        System.out.println("Stripping coffee through filter.");  
    }  
}
```

Template method pattern

```
package design.pattern.templateMethod;

public class TestTemplateMethod {

    static Beverage beverage = null;

    public static void main(String[] args) {
        System.out.println("=====Tea===== \n");
        Beverage tea = new Tea();
        tea.prepareRecepie();

        System.out.println("=====Coffie===== \n");
        Beverage coffie = new Coffie();
        coffie.prepareRecepie();
    }
}
```

```
<terminated> TestTemplateMethod [Java A]
=====
Tea=====
```

Boiling water.
Steeping the Tea.
Adding Lemon.
Pour into cup.
=====Coffie=====

Boiling water.
Stripping coffie through filter.
Add suger and milk.
Pour into cup.

References

- Head First Design Patterns: *by Eric Freeman & Elisabeth Freeman*
- Design Patterns: Elements of Reusable Object Oriented Software: *Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (GOF)*
<http://www.hillside.net>
- Core Security Patterns: Best Practices and Strategies J2EE Web Services and Identity Management: *Chris Steel, Ramesh Nagappan, Ray Lai, Sun Microsystems*
- Core J2EE Patterns, Best Practices and Design Strategies: *Deepak Alur, John Crupi, Dan Malks, 2nd Edition, Prentice Hall/ Sun Microsystems, 2003*
- <http://www.javaworld.com/jw-11-1998/jw-11-techniques.html>

The End

