

# *Advanced Programming*

## CSE 201

Instructor: Sambuddho

(Semester: Monsoon 2024)

Week 1 - Introduction and Basics

# Introduction

- What is this course about ? - Introduction to OOP (using Java)
- Course outline:
  - Basics of Java.
  - Classes and objects.
  - Inheritance.
  - Polymorphism.
  - Abstract classes.
  - Interfaces.
  - Exceptions and exception handling.
  - Collections.
  - I/O operations.
  - Multithread programming and synchronizations
  - Basics of design patterns.

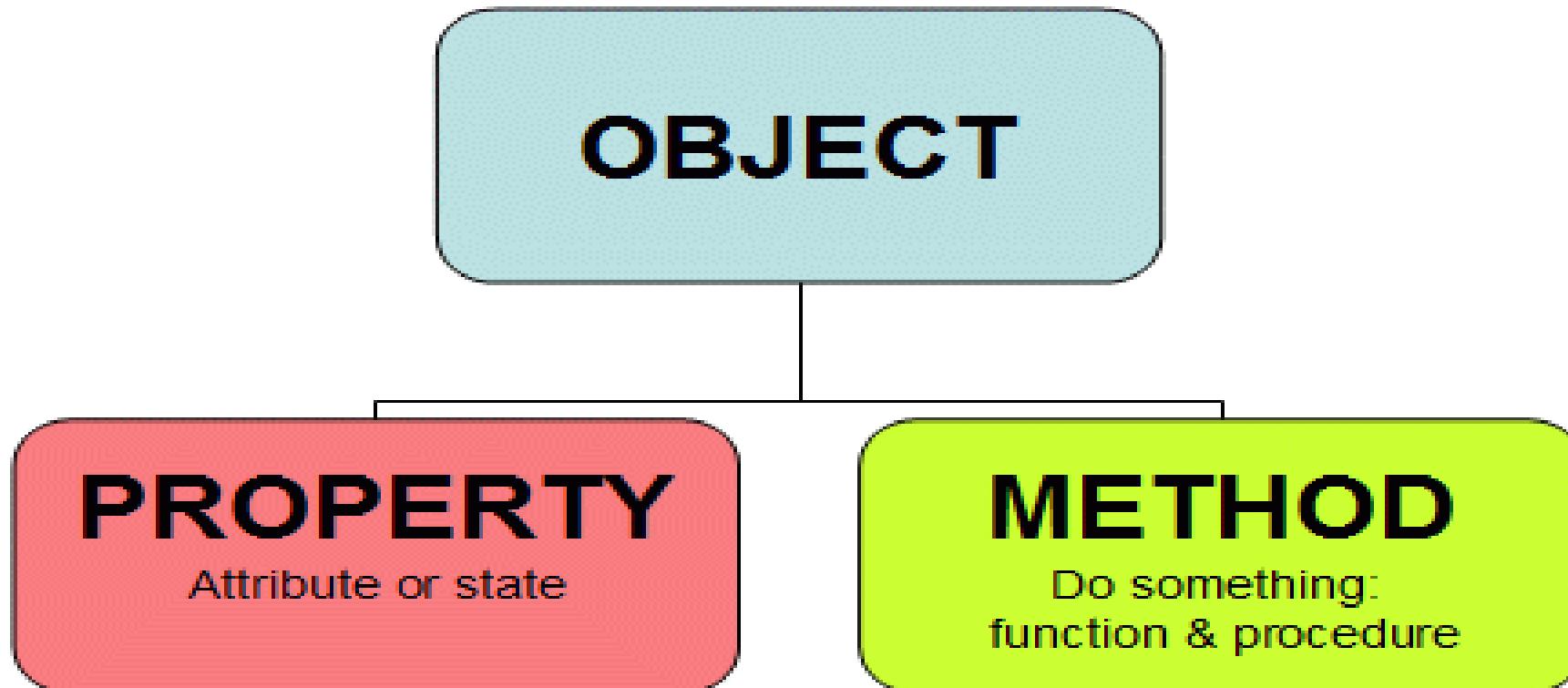
# Introduction

- Evaluations:
  - Assignments: 20% (total 4 assignments)
  - Quizzes: 10% (best 5 out of 6; 3 before midsem and 3 after midsem)
  - Midsem exam: 25%
  - Final exam: 25%
  - Project: 20% (in pairs)
- Note about quizzes: Must do attempt 2 before midsem and 2 after midsem.
- Textbook(s): (1) Core Java : Fundamentals, Volume I by Horstmann. Pearson (2) Core Java : Advanced Topics, Volume II by Horstmann. Pearson
- Office hours: TBA.

# Procedural Programming

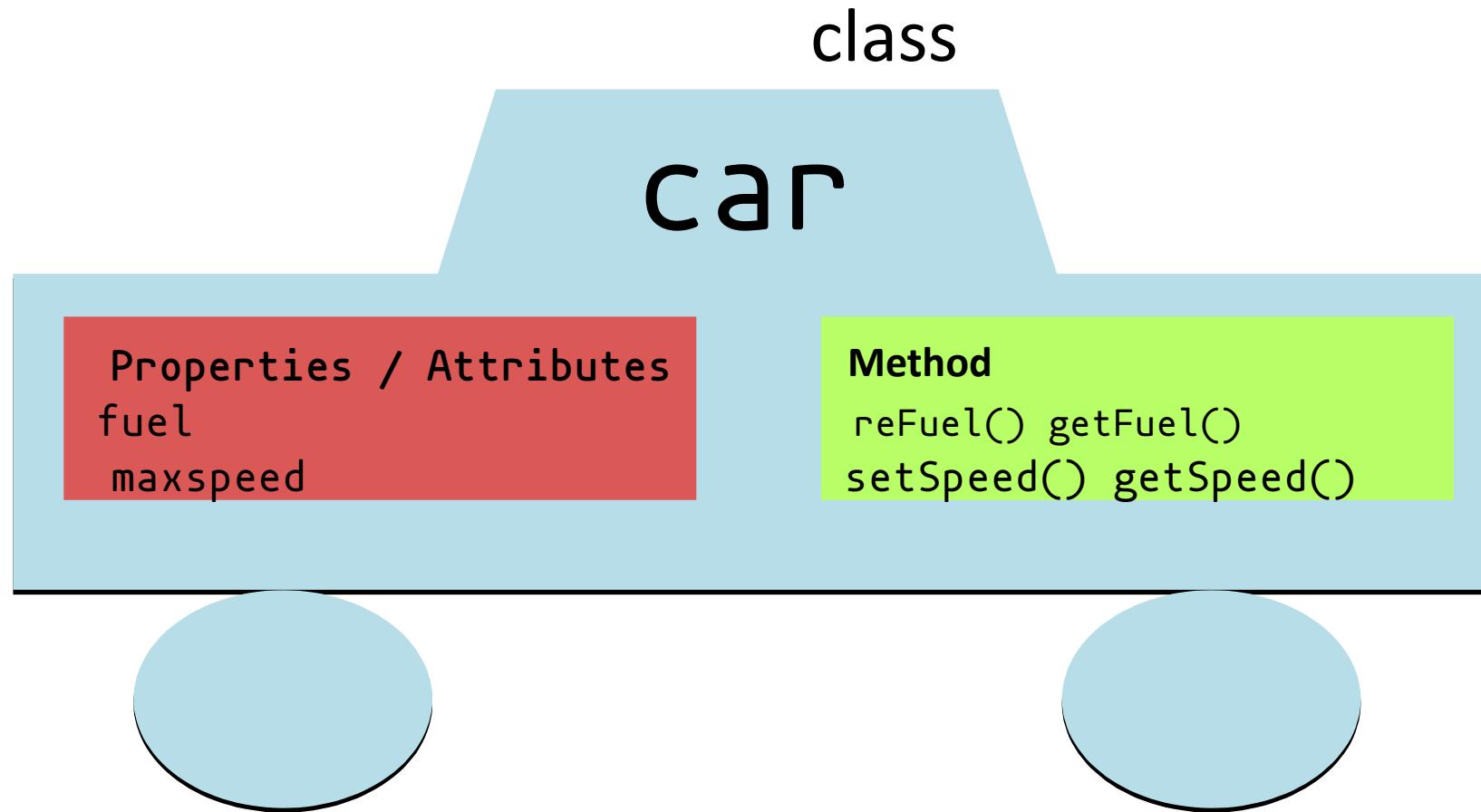
- Most natural progression from algorithms to code.
- Program: set of functions, one calls another in a fixed sequence.
- Abstract information is mapped to its storage.
  - Variables are actual memory locations based on types.
- Convenient for simple procedures - best choice for code e.g. device drivers, operating system etc.
- Not convenient for large application programs where abstractions are a must.

# What is OOP?



It is a programming paradigm based on the concept of “**objects**”, which may contain **data** in the form of **fields**, often known as **attributes**; and **code**, in the form of procedures, often known as **methods** (Wikipedia)

# What is OOP?



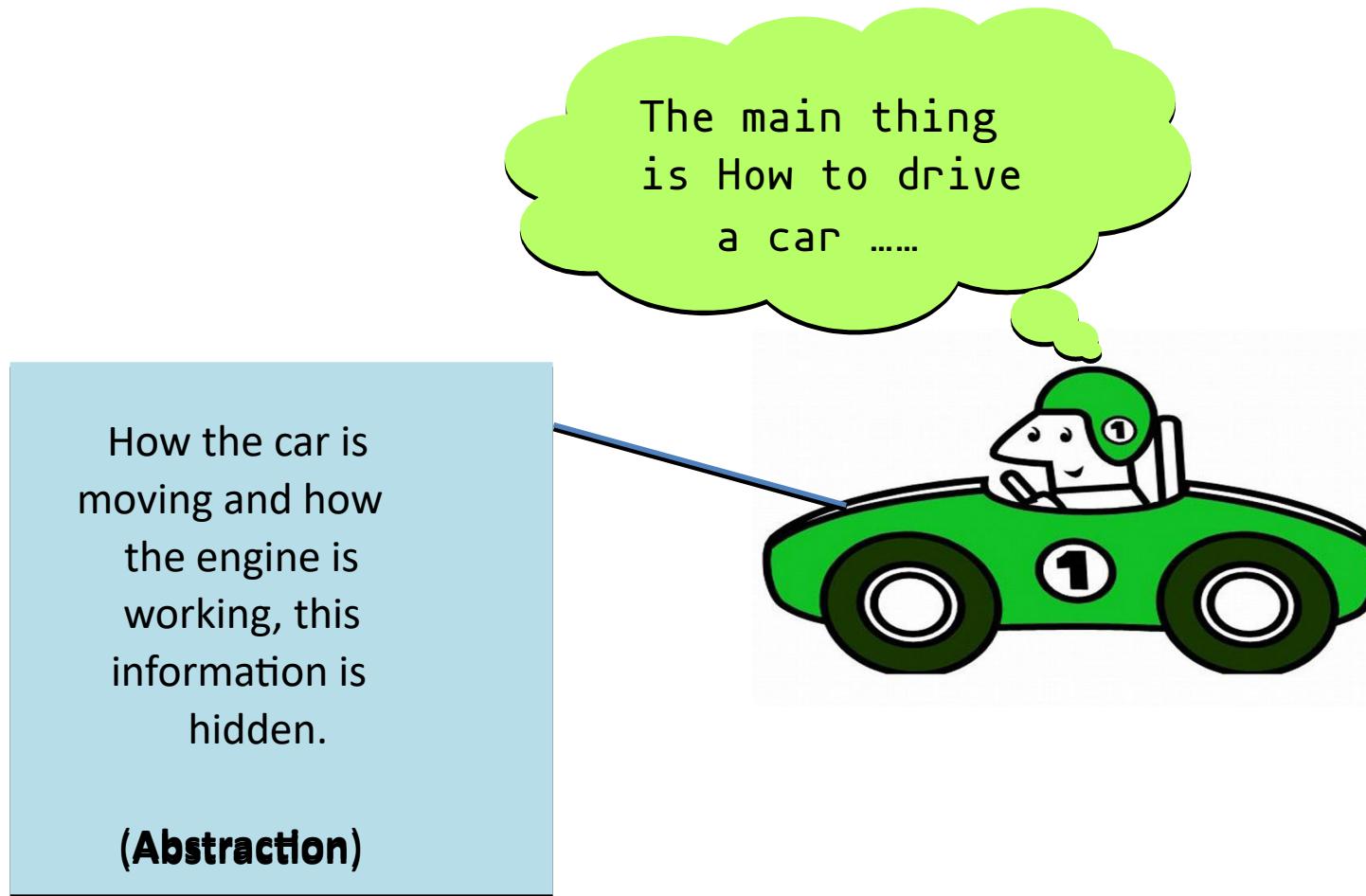
# OOP Features

- Encapsulation
- Method overloading
- Inheritance
- Abstraction
- Method overriding
- Polymorphism

# Advantages of OOP

- Code reuse and recycling
  - Objects can easily be reused
- Design benefits
  - Extensive planning phase results better design and lesser flaws
- Software maintenance
  - Easy to incorporate changes in legacy code (e.g., supporting a new hardware)
- Simplicity

# Abstraction



# Basics of Java Programming Language

- [Installing Java on Windows/Linux/MacOS/\\*nix](#)
  - Oracle/Sun Java JDK (proprietary - you only get the compiler and runtime)
  - OpenJDK/OpenJRE (open source - you can compile from the source)

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Server JRE	—	The software for running Java programs on servers
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on small devices
JavaFX	—	An alternate toolkit for graphical user interfaces that is included with certain Java SE distributions prior to Java 11
OpenJDK	—	A free and open source implementation of Java SE
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Oracle's term for a bug fix release up to Java 8
NetBeans	—	Oracle's integrated development environment

# Basics of Java Programming Language

---

- My first program:
- – Everything is a “class”.
- – The file name and classname must be the same.
- – Welcome.java
- – Java compilation:  
•   \$ java Welcome.java
- – Program runs in the JVM:  
•   \$ java Welcome
- Syntax very similar to C/C++.

```
1 /**
2  * This program displays a greeting for the reader.
3  * @version 1.30 2014-02-27
4  * @author Cay Horstmann
5  */
6 public class Welcome
7 {
8     public static void main(String[] args)
9     {
10         String greeting = "Welcome to Core Java!";
11         System.out.println(greeting);
12         for (int i = 0; i < greeting.length(); i++)
13             System.out.print("=");
14         System.out.println();
15     }
16 }
```

# Java vs C/C++

- – Machine independent.
- – Runs on a Java Virtual Machine (JRE) which understands a special byte-code format.
- – Doesn't run on bare metal.
- – High portability (Java/Python) vs performance (C/C++).
- – Is Java a compiled or an interpreted language ?

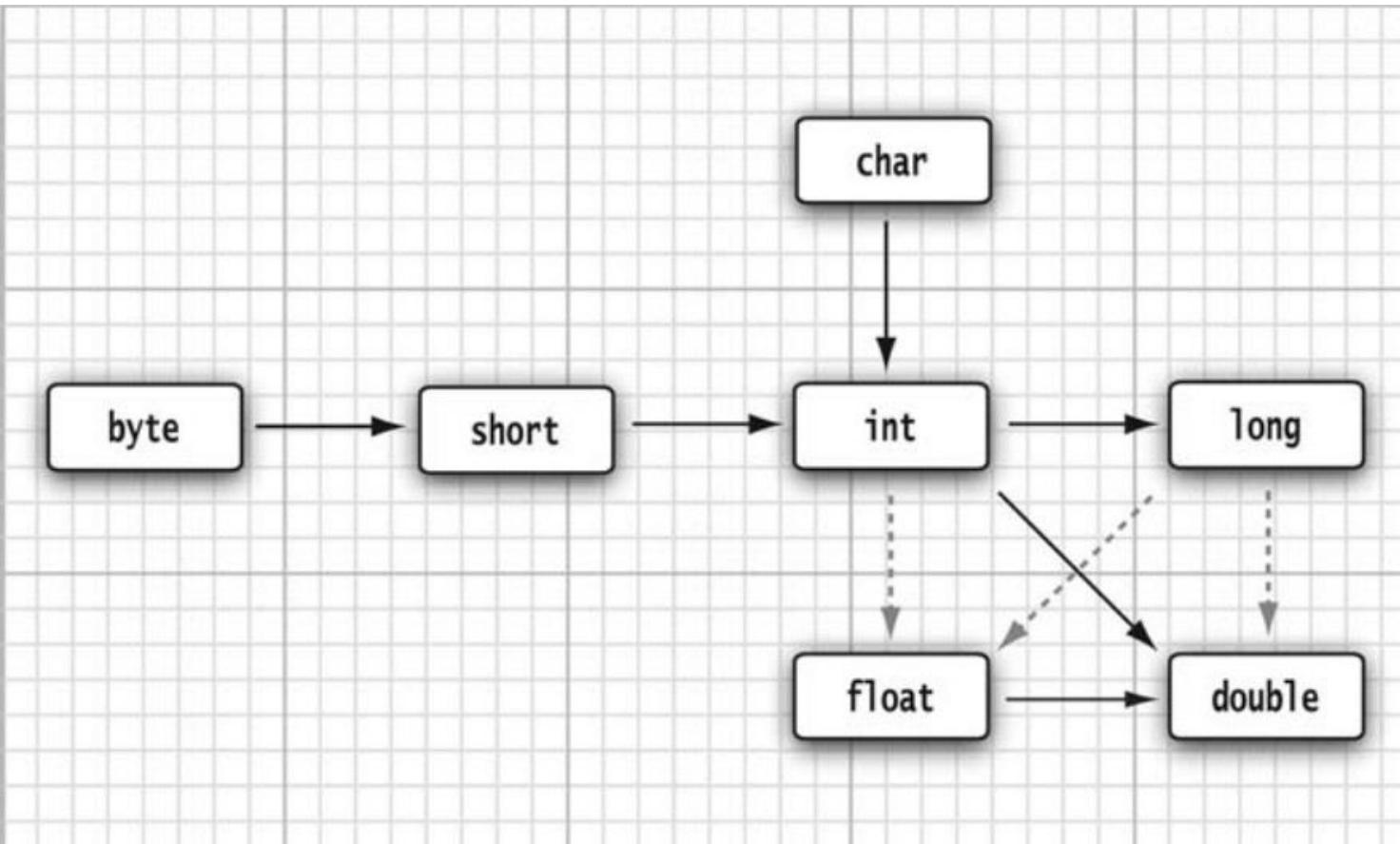
# Java: Types (Integer)

Type	Storage Requirement	Range (Inclusive)
int	4 bytes	-2,147,483,648 to 2,147,483,647 (just over 2 billion)
short	2 bytes	-32,768 to 32,767
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	1 byte	-128 to 127

# Java: Types (Floating point)

Type	Storage Requirement	Range
float	4 bytes	Approximately $\pm 3.40282347E+38F$ (6–7 significant decimal digits)
double	8 bytes	Approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)

# Java: Type casts/conversions



- Conversions without information loss  
 $\text{double} \rightarrow \text{int}$  [allowed by there wold be information loss]

# Java: Bitwise Operators

- & ('and')
- | ('or')
- ^ ('xor')
- ~ ('not')
- << ('left shift')
- >> ('right shift')

# Java: Operator Precedence and Hierarchy.

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= &=  = ^= <<= >>= >>>=	Right to left

# Java: Abstract Data Types (Strings)

- – ‘String’ abstract type.
- – Behaves like an *immutable* C++ string.
- Substrings:
  - String greetings = “Hello”;
  - String s = greeting.substring(0,3);
- Concatenation:
  - String greeting1 = “Hello”;
  - String greeting2 = “World”;
  - String message = greeting1 + greeting2;
- Testing string equality:
  - equals() method of String class used for testing equality of two strings.
  - <string object1>.equals(<string object2>)
  - Output: boolean (true or false).

# String Builder

```
String builder = new StringBuilder();
builder.append(ch);
builder.append(str);
String longString = builder.toString();
```

# Basic I/O

- Read input from stdin and write to stdout.
- Class ‘Scanner’

```
import java.util.*;  
  
Scanner in = new Scanner(System.in);  
  
in.next();  
in.nextLine();  
in.nextInt();  
in.nextDouble();  
in.hasNextInt() <-- boolean  
in.hasNextDouble() <-- boolean
```

# Basic I/O

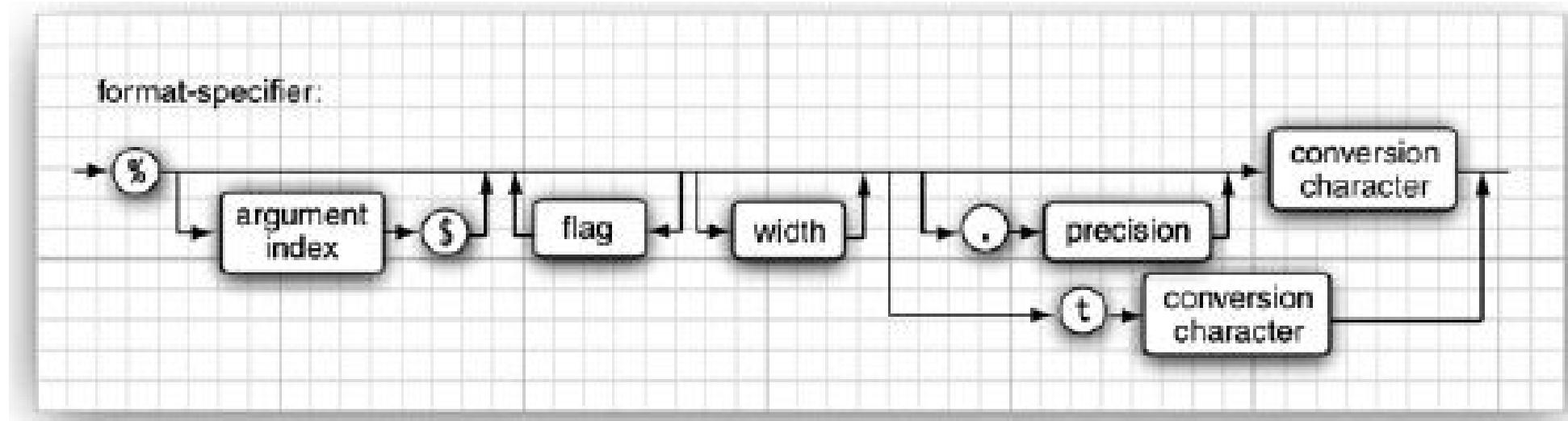
- Read input from stdin and write to stdout.
- Class ‘Scanner’

```
import java.util.*;  
  
Scanner in = new Scanner(System.in);  
  
in.next();  
in.nextLine();  
in.nextInt();  
in.nextDouble();  
in.hasNextInt() <-- boolean  
in.hasNextDouble() <-- boolean  
  
System.out.printf() <-- Java adaptation to C/C++ stdio printf()
```

# Basic I/O

Conversion Character	Type	Example
d	Decimal integer	159
x	Hexadecimal integer	9f
o	Octal integer	237
f	Fixed-point floating-point	15.9
e	Exponential floating-point	1.59e+01
g	General floating-point (the shorter of e and f)	—
a	Hexadecimal floating-point	0x1.fccdp3
s	String	Hello
c	Character	H
b	boolean	true
h	Hash code	42628b2
tx or Tx	Date and time (T forces uppercase)	Obsolete, use the <code>java.time</code> classes instead—see Chapter 6 of Volume II
%	The percent symbol	%
n	The platform-dependent line separator	—

# Basic I/O (printf format specifier)



# Control Flows

- if else
- while()
- for()
- switch case
- do{ }while()
- break and continue

Semantics same as C/C++

# Scope

- Determines the point in the code upto which a variable/function name etc. are valid.
- - Global / local
- - Also associated with functions inside a class.
- - Java is more conservative about scopes than C/C++.

```
public static void main(String[] args)
{
    int n;
    . .
    {
        int k;
        .
    } // k is only defined up to here
}
```

```
public static void main(String[] args)
{
    int n;
    {
        int k;
        int n; // ERROR--can't redefine n in inner block
        .
    }
}
```

# Arrays and References

- – Contiguous ``allocate memory''
- `int[] arr = new int[100]; //new keyword mandatory.`
- – Reference: much like pointers in C/C++ – Name of the object or array.
- `int[] arr = new int[100];`
- `int[] myref;`
- `myref = arr; // myref → arr : reference.`

# Multidimensional Arrays

- `<type>[][] <var-name>;`
- `int[][] marray;`
- `marray = new int [100];`
- `for (int i=0;i<100;i++){`
- `marray[i] = new int[100];`
- `...`
- `}`

# *Advanced Programming*

## CSE 201

Instructor: Sambuddho

(Semester: Monsoon 2024)

Week 2 - Classes and Objects – Basics

# Classes and Objects.

- Defining your own class:

```
class ClassName
{
    field1
    field2
    .
    .
    .
    constructor1
    constructor2
    .
    .
    .
    method1
    method2
    .
    .
}
```

```
class Employee
{
    // instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;

    // constructor
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    // a method
    public String getName()
    {
        return name;
    }

    // more methods
    .
}
```

# Classes and Objects.

- Defining your own class:

- 

- Array of objects:

- 

```
Employee[] staff = new Employee[3];
```

```
staff[0] = new Employee("Carl Cracker", . . .);
```

```
staff[1] = new Employee("Harry Hacker", . . .);
```

```
staff[2] = new Employee("Tony Tester", . . .);
```

Class with `main()` method should *ideally* be different from the class whose objects it uses.

# Classes and Objects.

```
import java.time.*;  
  
/**  
 * This program tests the Employee class.  
 * @version 1.13 2018-04-10  
 * @author Cay Horstmann  
 */  
public class EmployeeTest  
{  
    public static void main(String[] args)  
    {  
        // fill the staff array with three Employee objects  
        Employee[] staff = new Employee[3];  
  
        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);  
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);  
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);  
  
        // raise everyone's salary by 5%  
        for (Employee e : staff)  
            e.raiseSalary(5);  
  
        // print out information about all Employee objects  
        for (Employee e : staff)  
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay=" +  
                + e.getHireDay());  
    }  
  
    class Employee  
    {  
        private String name;  
        private double salary;  
        private LocalDate hireDay;  
  
        public Employee(String n, double s, int year, int month, int day)  
        {  
            name = n;  
            salary = s;  
            hireDay = LocalDate.of(year, month, day);  
        }  
  
        public String getName()  
        {  
            return name;  
        }  
    }  
}
```

# Private/Public Methods and Variables

- - Unlike C/C++ all methods and variables are public.
- - To make a class visible to the JVM and other files you must declare it as public.
- - All public classes need their respective files.
- - Non-public (private) class need not have their own files.

# Private/Public Methods and Variables

- **Final keyword**
- – WORM type field (define and initialize in the beginning and use as many times as you like without reassining or changing it).
- – Can be used for normal variable as well as class object variables.
- **Private keyword**
- – Private keyword for class members - cannot be accessed through objects. Can only be accessed by class methods.

# Static Methods and Variables

- Static keyword
- - Philosophy: Have a single instance of a class object, variable, method etc.
- - Class variable with ‘static’ keyword - Can be used without the class object only with the class name only.
- - Only one variable shared across all objects of the class.

# Static Methods and Variables

- Static constants
- static final

```
public class Math
{
    .
    .
    public static final double PI = 3.14159265358979323846;
    .
    .
}
```

# Constructors and Constructor Overloading

- - Called when the class object is created.
- - Explicit constructor vs implicit.
- Implicit: Default when there are no explicit constructors.
- Explicit: Function with same name as class and no return values. Can have any number of arguments.
- - Constructor overloading: Multiple constructors with the same name, differing only wrt the args.

# Initializations

- Constructor(s).
- Value initialization at declaration time.
- Value initialization block.

```
class Employee
{
    private static int nextId;           id = nextId;
                                         nextId++;
                                         }

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public Employee()
    {
        name = "";
        salary = 0;
    }

    ...
}

// object initialization block
```

- New feature
- Initialization.

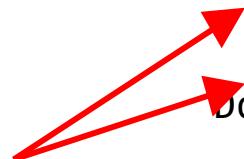
# Packages and Imports

- - Collection of classes - *package*.
- - *Packaged in a directory*.
- - Packages are clustered in a directory structure.
- - Classes with same name can reside inside their respective *packages*. No conflict.
- - Classname can be a fully qualified classname:
  - `java.time.LocalDate`
  - OR
  - Implicit based on the imported packages:
    - `import java.time.*;`
    - `LocalDate today = LocalDate.now();`

# What is Memory?

- Memory (system memory, not disk or other peripheral devices) is the hardware in which computers store information, both temporary and permanent
- Think of memory as a list of slots; each slot holds information (e.g., a local `int` variable, or a reference to an instance of a class)
- Here, two references are stored in memory: one to a `Dog` instance, and one to a `DogGroomer` instance

```
//Elsewhere in the program  
Petshop petSmart = new Petshop();  
  
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new  
        DogGroomer();  
        groomer.groom(django);  
    }  
}
```

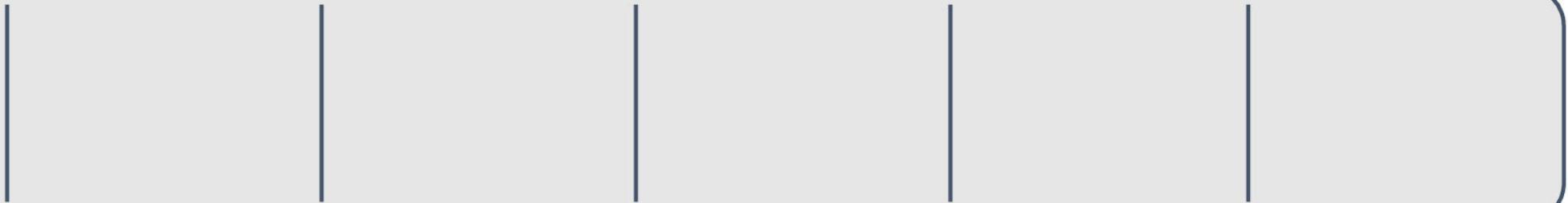


# Objects as Parameters: Under the Hood (1/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here  
    }  
}
```

Somewhere in memory...



# Objects as Parameters: Under the Hood (2/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



When we instantiate a

Dog , he's stored somewhere in memory. Our

PetShop will use the

# Objects as Parameters: Under the Hood (3/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



The same goes for the

DogGroomer

—we store a particular

DogGroomer

somewhere in

# Objects as Parameters: Under the Hood (4/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



We call the **groom** method on our **DogGroomer**, **groomer**. We need to tell her which **Dog** to

# Objects as Parameters: Under the Hood (5/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



# Objects as Parameters: Under the Hood (6/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...

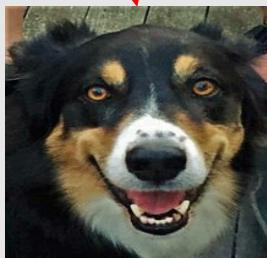


The `groom` method doesn't really care which

Dog it's told to groom—no matter what another

# Variable Reassignment: Under the Hood (1/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



# Variable Reassignment: Under the Hood (2/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



# Variable Reassignment: Under the Hood (3/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



# Local Variables (1/2)

- All variables we've seen so far have been **local variables**: variables declared *within a method*
- Problem: the **scope** of a local variable (where it is known and can be accessed) is limited to its own method—it cannot be accessed from anywhere else
  - the same is true of method parameters

```
public class PetShop {  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

**local variables**



# Local Variables (2/2)

- We created `groomer` and `django` in our `PetShop`'s helper method, but as far as the rest of the class is concerned, they don't exist
- What happens to `django` after the method is executed?
  - “Garbage Collection”

```
public class PetShop {  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

**local variables**



# Variable Reassignment: Under the Hood (4/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog(); //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```



# Variable Reassignment: Under the Hood (5/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();      //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```



*Advanced Programming*  
**CSE 201**  
**Instructor: Sambuddho**

(Semester: Monsoon 2024)  
Week 2 – Classes and Objects - Basics

# Classes and Objects.

- Defining your own class:

```
class ClassName
{
    field1
    field2
    .
    .
    .
    constructor1
    constructor2
    .
    .
    .
    method1
    method2
    .
    .
}
```

```
class Employee
{
    // instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;

    // constructor
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    // a method
    public String getName()
    {
        return name;
    }

    // more methods
    .
}
```

# Classes and Objects.

- Defining your own class:

- 

- Array of objects:

- 

```
Employee[] staff = new Employee[3];
```

```
staff[0] = new Employee("Carl Cracker", . . .);
```

```
staff[1] = new Employee("Harry Hacker", . . .);
```

```
staff[2] = new Employee("Tony Tester", . . .);
```

Class with `main()` method should *ideally* be different from the class whose objects it uses.

# Classes and Objects.

```
import java.time.*;

/**
 * This program tests the Employee class.
 * @version 1.13 2018-04-10
 * @author Cay Horstmann
 */
public class EmployeeTest
{
    public static void main(String[] args)
    {
        // fill the staff array with three Employee objects
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

        // raise everyone's salary by 5%
        for (Employee e : staff)
            e.raiseSalary(5);

        // print out information about all Employee objects
        for (Employee e : staff)
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
                + e.getHireDay());
    }
}

class Employee
{
    private String name;
    private double salary;
    private LocalDate hireDay;

    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    public String getName()
    {
        return name;
    }
}
```

# Private/Public Methods and Variables

- - Unlike C/C++ all methods and variables are public.
- - To make a class visible to the JVM and other files you must declare it as **public**.
- - All public classes need their respective files.
- - Non-public (private) class need not have their own files.

# Private/Public Methods and Variables

- **Final** keyword
  - - WORM type field (define and initialize in the beginning and use as many times as you like without reassining or changing it).
  - - Can be used for normal variable as well as class object variables.
- **Private** keyword
  - - Private keyword for class members – cannot be accessed through objects. Can only be accessed by class methods.

# Static Methods and Variables

- **Static keyword**
- - Philosophy: Have a single instance of a class object, variable, method etc.
- - Class variable with ‘static’ keyword – Can be used without the class object only with the class name only.
- - Only *one* variable shared across all objects of the class.

# Static Methods and Variables

- **Static** constants
- static final

```
public class Math
{
    .
    .
    public static final double PI = 3.14159265358979323846;
    .
    .
}
```

# Constructors and Constructor Overloading

- - Called when the class object is created.
- - Explicit constructor vs implicit.
- Implicit: Default when there are no explicit constructors.
- Explicit: Function with same name as class and no return values. Can have any number of arguments.
  - - Constructor overloading: Multiple constructors with the same name, differing only wrt the args.

# Initializations

- - Constructor(s).
- - Value initialization at declaration time.
- - Value *initialization block*.

```
class Employee
{
    private static int nextId;
    private int id;
    private String name;
    private double salary;

    // object initialization block
    {

        id = nextId;
        nextId++;
    }

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public Employee()
    {
        name = "";
        salary = 0;
    }

    ...
}
```

- - ~~No value before~~
- - initialization.

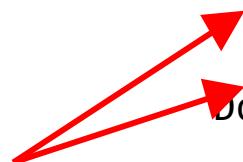
# Packages and Imports

- Collection of classes – *package*.
- - *Packaged* in a directory.
- - Packages are clustered in a directory structure.
- - Classes with same name can reside inside their respective *packages*.  
No conflict.
- - Classname can be a fully qualified classname:
  - java.time.LocalDate
  - OR
  - Implicit based on the imported packages:
    - import java.time.\*;
    - LocalDate today = LocalDate.now();

# What is Memory?

- Memory (system memory, not disk or other peripheral devices) is the hardware in which computers store information, both temporary and permanent
- Think of memory as a list of slots; each slot holds information (e.g., a local `int` variable, or a reference to an instance of a class)
- Here, two references are stored in memory: one to a `Dog` instance, and one to a `DogGroomer` instance

```
//Elsewhere in the program  
Petshop petSmart = new Petshop();  
  
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new  
        DogGroomer();  
        groomer.groom(django);  
    }  
}
```

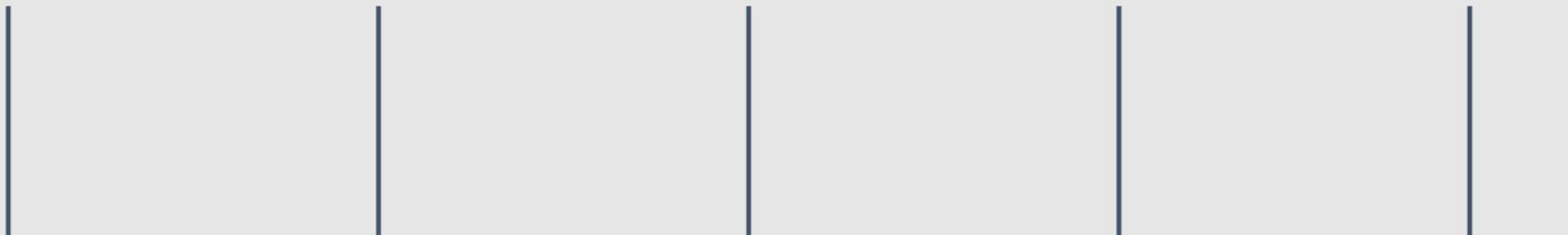


# Objects as Parameters: Under the Hood (1/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here  
    }  
}
```

Somewhere in memory...



# Objects as Parameters: Under the Hood (2/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
  
}  
  
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
  
}
```

Somewhere in memory...



When we instantiate a

Dog , he's stored somewhere in memory. Our

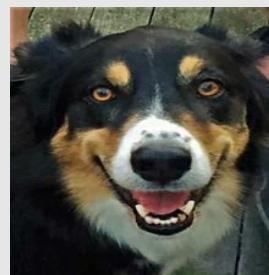
PetShop will use the

# Objects as Parameters: Under the Hood (3/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



The same goes for the

DogGroomer

—we store a particular

DogGroomer

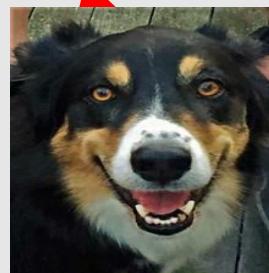
somewhere in

# Objects as Parameters: Under the Hood (4/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



We call the **groom** method on our **DogGroomer**, **groomer**. We need to tell her which **Dog** to

# Objects as Parameters: Under the Hood (5/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...

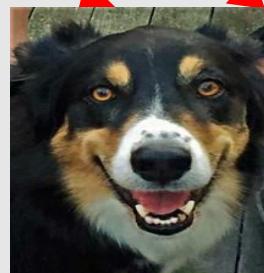


# Objects as Parameters: Under the Hood (6/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...

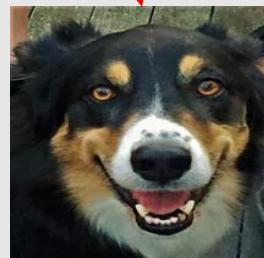


The `groom` method doesn't really care which

Dog it's told to groom—no matter what another

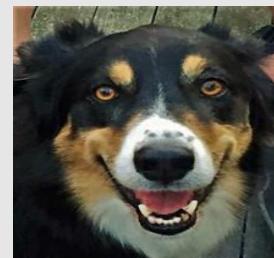
# Variable Reassignment: Under the Hood (1/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



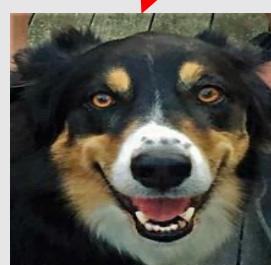
# Variable Reassignment: Under the Hood (2/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



# Variable Reassignment: Under the Hood (3/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

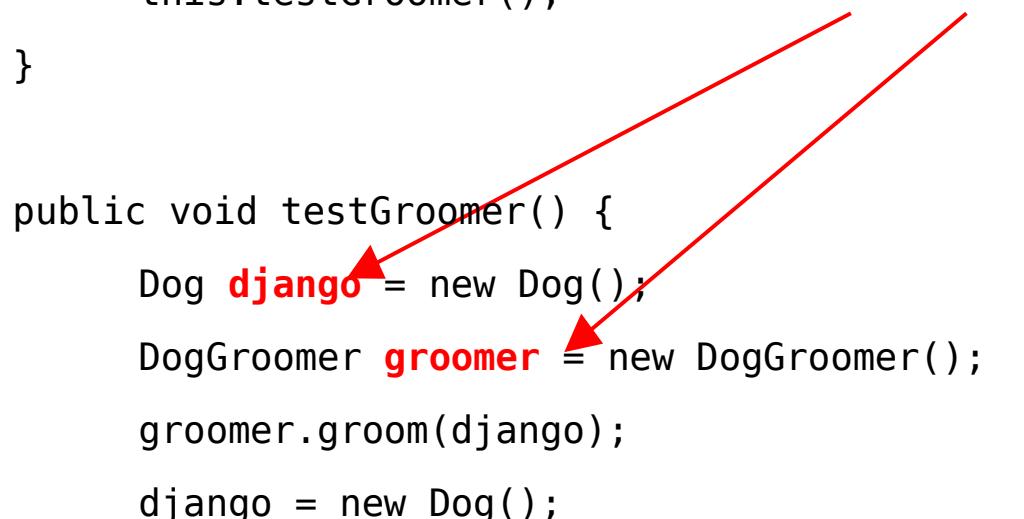


# Local Variables (1/2)

- All variables we've seen so far have been **local**  
**variables**: variables declared *within a method*
- Problem: the **scope** of a local variable (where it is known and can be accessed) is limited to its own method—it cannot be accessed from anywhere else
  - the same is true of method parameters

```
public class PetShop {  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

**local variables**

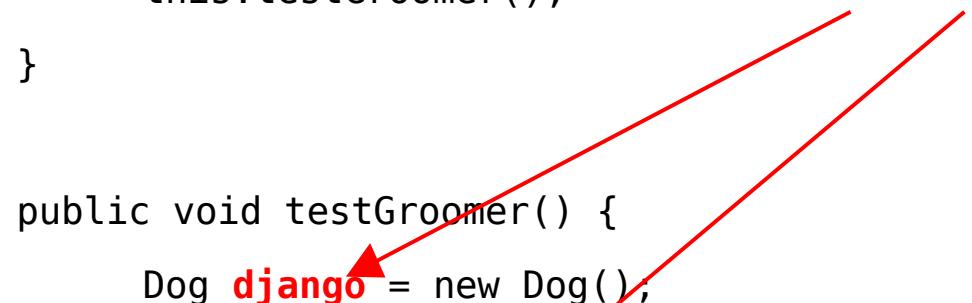


# Local Variables (2/2)

- We created **groomer** and **django** in our **PetShop**'s helper method, but as far as the rest of the class is concerned, they don't exist
- What happens to django after the method is executed?
  - “Garbage Collection”

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

**local variables**



# Variable Reassignment: Under the Hood (4/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog(); //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```



# Variable Reassignment: Under the Hood (5/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();           //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```



# *Advanced Programming*

## CSE 201

### Instructor: Sambuddho

(Semester: Monsoon 2024)

Week 3 – Inheritance

# Basics of Inheritance (in Java)

- Closely connected to polymorphism.
- Def\_1: Referencing many related objects as one generic type.
- Def\_2: Reference of 'parent' class utilizing attributes and methods of a 'child' class, depending on which one it is referencing.

# Slide Acknowledgement

CS15, Brown University

# Basics of Inheritance (in Java)

- Closely connected to polymorphism.
- Def\_1: Referencing many related objects as one generic type.
- Def\_2: Reference of 'parent' class utilizing attributes and methods of a 'child' class, depending on which one it is referencing.

# Spot the Similarities



- What are the similarities between a convertible and a sedan?
- What are the differences?

# Convertibles vs. Sedans

## Convertible

- Top Down Roof  
(Retractable Roof)

## Sedan

- Fixed Roof

- Drive
- Brake
- Play radio
- Lock/unlock doors
- Turn off/on turn engine

# Can we model this in code?

- In some cases, objects can be very closely related to each other
  - Convertibles and sedans drive the same way
  - Flip phones and smartphones call the same way
- Imagine we have an Convertible and a Sedan class
  - Can we enumerate their similarities in one place?
  - How do we portray their relationship through code?

## Convertible

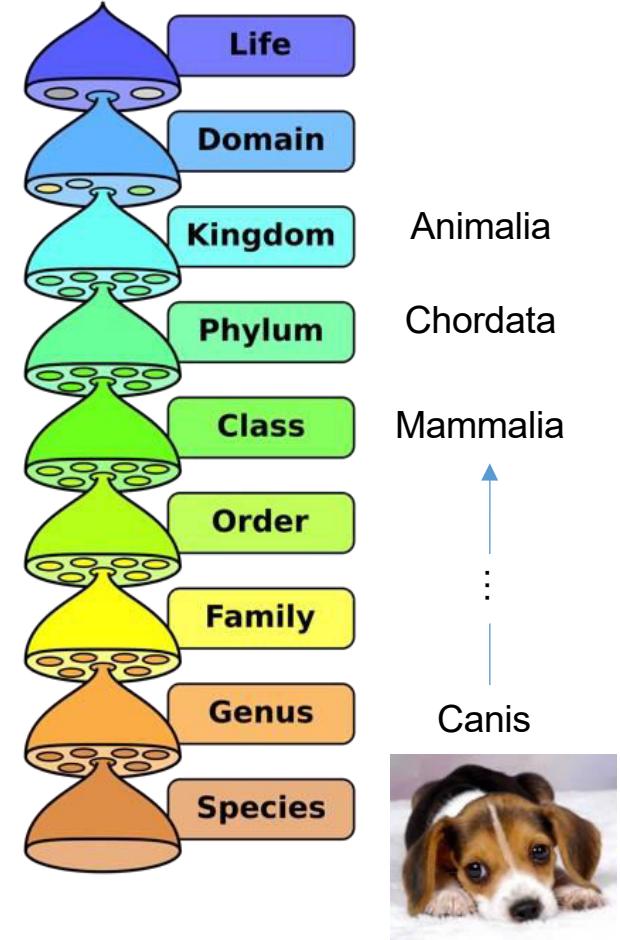
- putTopDown()
- turnOnEngine()
- turnOffEngine()
- drive()

## Sedan

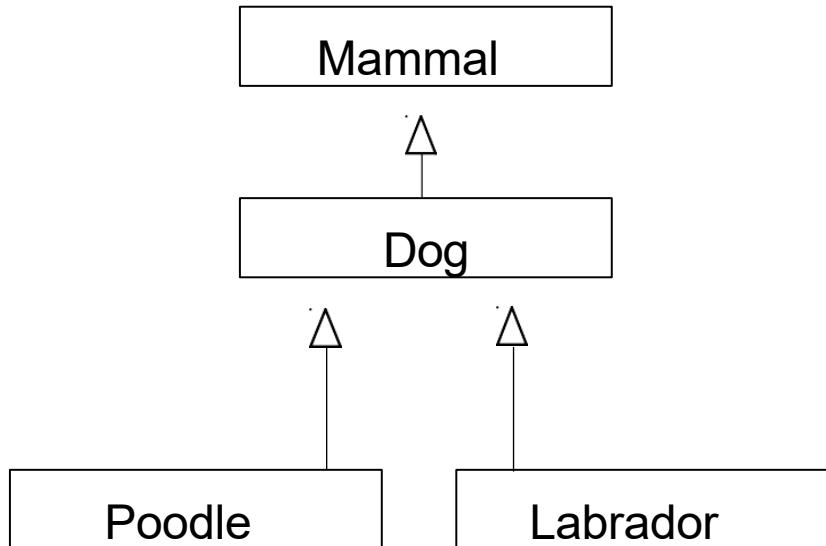
- parkInCompactSpace ()
- turnOnEngine()
- turnOffEngine()
- drive()

# Inheritance

- In OOP, inheritance is a way of modeling very similar classes
- **Inheritance** models an “**is-a**” relationship
  - A **sedan** “is a” **car**
  - A **dog** “is a” **mammal**
- Remember: **Interfaces** model an “**acts-as**” relationship
- You’ve probably seen inheritance before!
  - Taxonomy from biology class

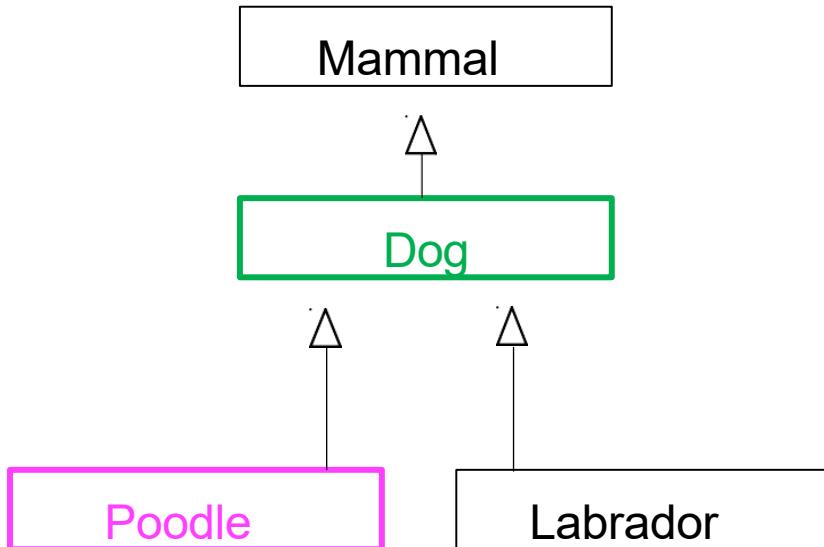


# Modeling Inheritance (1/2)



- This is an inheritance diagram
  - Each box represents a class
- A Poodle “is-a” Dog, a Dog “is-a” Mammal
  - Transitively, a Poodle is a Mammal
- “Inherits from” = “is-a”
  - Poodle inherits from Dog
  - Dog inherits from Mammal
- This relationship is not bidirectional
  - A Poodle is a Dog, but not every Dog is a Poodle (could be a Labrador, a German Shepard, etc)

# Modeling Inheritance (2/2)



- **Superclass/parent/base**: A class that is inherited from
- **Subclass/child/derived**: A class that inherits from another
  - “A **Poodle** is a **Dog**”
    - **Poodle** is the **subclass**
    - **Dog** is the **superclass**
  - A class can be both a **superclass** and a **subclass**
    - Ex. Dog
  - In Java you can only inherit from one superclass (no multiple inheritance)
    - Other languages, like C++, allow for multiple

inheritance, but too easy to mess up

# Motivations for Inheritance

- A subclass inherits all of its parent's **public and protected** capabilities
  - If Car defines drive(), Convertible inherits drive() from Car and drives the same way. This holds true for all of Convertible's subclasses as well
- Inheritance and Interfaces both legislate class's behavior, although in very different ways
  - Interfaces allow the compiler to enforce method implementation
    - An implementing class will have all capabilities outlined in an interface
  - Inheritance assures the compiler that all **subclasses of a superclass** will have the superclass's public capabilities without having to respecify code – methods are inherited
    - A Convertible knows how to drive and drives the same way as Car because of inherited code
- Benefit of inheritance
  - Code reuse
    - If drive() is defined in Car, Convertible doesn't need to redefine it! Code is inherited. Only need to implement what is different, i.e. what makes Convertible special

# Superclasses vs Subclasses

- A superclass factors out commonalities among its subclasses
  - describes everything that all subclasses have in common.
  - **NOTE: Java classes can have only one parent (super) class, unlike CPP.**
- A subclass differentiates/specializes its superclass by:
  - **adding new methods:**
    -
  - **Overriding inherited methods:**
    -

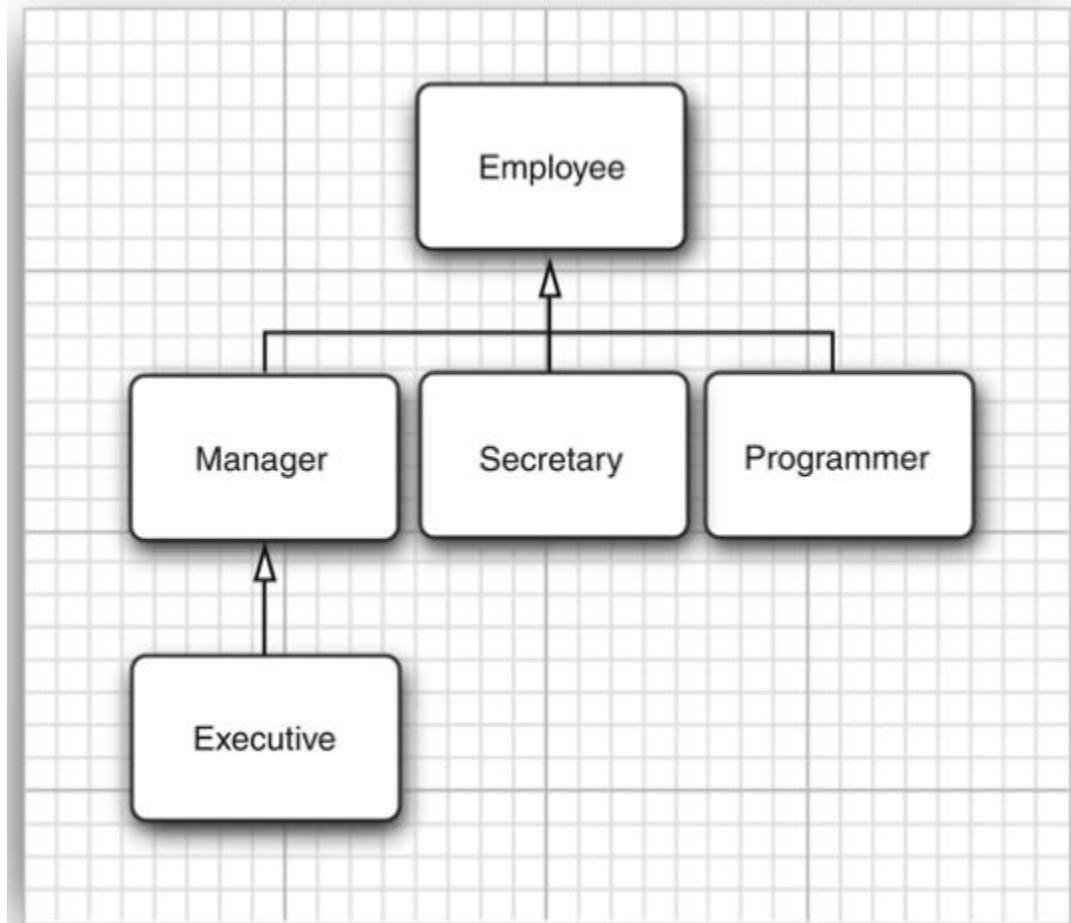
# Method Overriding

- Override parent class ‘public’ methods (not private).
  - Child can access ‘public’ methods and attributes of parent class.
  - Using parent class attributes – super.<Attib\_name>
  - Using parent class methods – super.f()
  - Calling parent class constructor

```
<subclass constructor>{  
    super(<arguments>)  
}
```

```
public class Manager extends Employee  
{  
    private double bonus;  
    ...  
    public void setBonus(double bonus)  
    {  
        this.bonus = bonus;  
    }  
    public double getSalary()  
    {  
        double baseSalary = super.getSalary();  
        return baseSalary + bonus;  
    }  
}
```

# Inheritance and Polymorphism



- Polymorphism is an incredibly powerful tool.
- Allows for generic programming.
- Treat multiple classes as their generic type while still allowing specific method implementations to be executed.
- Polymorphism+Inheritance is strong generic coding

# Inheritance and Polymorphism

```
1 package inheritance;
2
3 import java.time.*;
4
5 public class Employee
6 {
7     private String name;
8     private double salary;
9     private LocalDate hireDay;
10
11     public Employee(String name, double salary, int year, int month, int day)
12     {
13         this.name = name;
14         this.salary = salary;
15         hireDay = LocalDate.of(year, month, day);
16     }
17
18     public String getName()
19     {
20         return name;
21     }
22
23     public double getSalary()
24     {
25         return salary;
26     }
27
28     public LocalDate getHireDay()
29     {
30         return hireDay;
31     }
32
33     public void raiseSalary(double byPercent)
34     {
35         double raise = salary * byPercent / 100;
36         salary += raise;
37     }
38 }
```

# Inheritance and Polymorphism

```
1 package inheritance;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6
7     /**
8      * @param name the employee's name
9      * @param salary the salary
10     * @param year the hire year
11     * @param month the hire month
12     * @param day the hire day
13     */
14     public Manager(String name, double salary, int year, int month, int day)
15     {
16         super(name, salary, year, month, day);
17         bonus = 0;
18     }
19
20     public double getSalary()
21     {
22         double baseSalary = super.getSalary();
23         return baseSalary + bonus;
24     }
25
26     public void setBonus(double b)
27     {
28         bonus = b;
29     }
30 }
```

---

# Inheritance and Polymorphism

```
1 package inheritance;
2
3 import java.time.*;
4
5 public class Employee
6 {
7     private String name;
8     private double salary;
9     private LocalDate hireDay;
10
11     public Employee(String name, double salary, int year, int month, int day)
12     {
13         this.name = name;
14         this.salary = salary;
15         hireDay = LocalDate.of(year, month, day);
16     }
17
18     public String getName()
19     {
20         return name;
21     }
22
23     public double getSalary()
24     {
25         return salary;
26     }
27
28     public LocalDate getHireDay()
29     {
30         return hireDay;
31     }
32
33     public void raiseSalary(double byPercent)
34     {
35         double raise = salary * byPercent / 100;
36         salary += raise;
37     }
38 }
```

# Inheritance and Polymorphism

```
Employee e;  
e = new Employee(. . .); // Employee object expected  
e = new Manager(. . .); // OK, Manager can be used as well
```

```
Manager boss = new Manager(. . .);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

```
boss.setBonus(5000); // OK
```

Why ?

```
staff[0].setBonus(5000); // ERROR
```

# Rules for Method Calls

- 1. Compiler looks at types of objects and method names, determines the appropriate method based on the return type. The compiler also resolves.
- 2. Argument types.
- 3. Method types – ‘private’ , ‘static’ , ‘final’.
- 4. Polymorphic associations – dynamic binding at runtime.

# Preventing Inheritance

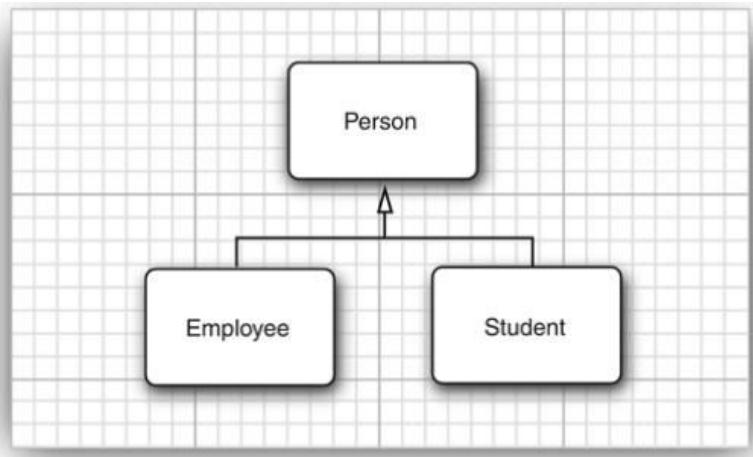
- 1. What all a child class inherits:
  - Public objects and methods.
  - Private of parent is never inherited.
  - Parent constructors (by default public, like everything else in Java).
- Preventing inheritance:

```
public final class Executive extends Manager
{
    ...
}
```

- You could also prevent specific methods from being inherited *without* making them private.

```
public class Employee
{
    ...
    public final String getName()
    {
        return name;
    }
    ...
}
```

# Abstract Classes



```
public abstract class Person
{
    ...
    public abstract String getDescription();
}
```

```
1 package abstractClasses;
2
3 public abstract class Person
4 {
5     public abstract String getDescription();
6     private String name;
7
8     public Person(String name)
9     {
10         this.name = name;
11     }
12
13     public String getName()
14     {
15         return name;
16     }
17 }
```

---

```
1 package abstractClasses;
2
3 import java.time.*;
4
5 public class Employee extends Person
6 {
7     private double salary;
8     private LocalDate hireDay;
9
10    public Employee(String name, double salary, int year, int month, int day)
11    {
12        super(name);
13        this.salary = salary;
14        hireDay = LocalDate.of(year, month, day);
15    }
16
17    public double getSalary()
18    {
19        return salary;
20    }
21
22    public LocalDate getHireDay()
23    {
24        return hireDay;
25    }
26
27    public String getDescription()
28    {
29        return String.format("an employee with a salary of %.2f", salary);
30    }
31
32    public void raiseSalary(double byPercent)
33    {
34        double raise = salary * byPercent / 100;
35        salary += raise;
36    }
37 }
```

# Protected Access

- ‘Protected’ methods and objects cannot be accessed by objects of the class much like ‘private’.
- ‘Protected’ methods and objects/variable can be accessed by child classes.

# Object – the Cosmic Super Class

- Object – parent of all classes (implicit); not of primitive types – int, char, byte etc.

```
Object obj = new Employee("Harry Hacker", 35000);
```

- Can be a: Employee e = (Employee) obj;
- Arrays – regardless they are of primitive types or of classes, are of type Object.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

# Object – equals() method

- Test if two objects ‘equal’ or same. Could mean many things – reference to same object (default), same value etc.
- Object class defines equal(). Other classes, can extend it with their own definition.

```
public class Employee
{
    ...
    public boolean equals(Object otherObject)
    {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;

        // must return false if the explicit parameter is null
        if (otherObject == null) return false;

        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass())
            return false;

        // now we know otherObject is a non-null Employee
        Employee other = (Employee) otherObject;
        ...
        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```

# Object – `toString()` method

- Used to for printing a string equivalent information of the class. E.g. `System.out.println(s);` the return type is String and the name is `toString()`
- ```
class MyClass{  
    ...  
    public String toString(){  
        return "XYZ";  
    }  
}
```

# Object Wrappers and Autoboxing

- Object corresponding to a primitive type [Integer, Long, Float, Double, Short, Byte and Boolean].
- Their objects are immutable – once a wrapper object has been created their values cannot be changed.
- They are ‘final’ and cannot be inherited.
- `Integer[] list;`
- `list = new Integer[500];`
- `list[0] = 1;`
- `list[1] = 1;`
- `list[i]++;`
- But `list[i] == list[j]` fails.

# Object Wrappers and Autoboxing

- `list[i].intValue() == list[j].intValue();`

## `java.lang.Integer 1.0`

- `int intValue()`  
returns the value of this Integer object as an int (overrides the intValue method in the Number class).
- `static String toString(int i)`  
returns a new String object representing the number i in base 10.
- `static String toString(int i, int radix)`  
lets you return a representation of the number i in the base specified by the radix parameter.
- `static int parseInt(String s)`
- `static int parseInt(String s, int radix)`  
returns the integer whose digits are contained in the string s. The string must represent an integer in base 10 (for the first method) or in the base given by the radix parameter (for the second method).
- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`  
returns a new Integer object initialized to the integer whose digits are contained in the string s. The string must represent an integer in base 10 (for the first method) or in the base given by the radix parameter (for the second method).

## `java.text.NumberFormat 1.1`

- `Number parse(String s)`  
returns the numeric value, assuming the specified String represents a number.

# Enum classes

- Enumerated types – alternatives to constants.
- public enum Size { SMALL, MEDIUM, LARGE, EXTRA\_LARGE }

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
}
```

```
Size s = Enum.valueOf(Size.class, "SMALL");
```

# Generic ArrayLists

- ArrayList is a *generic class* with a *type parameter*.
- An example of polymorphism.
- Provides a dynamically growing (or shrinking) array of objects.
- `ArrayList<Integer> mylist = new ArrayList<Integer>();`
- or
- `ArrayList<Integer> mylist = new ArrayList<>();`

# Generic ArrayLists

`java.util.ArrayList<E>` 1.2

- `ArrayList<E>()`

constructs an empty array list.

- `ArrayList<E>(int initialCapacity)`

constructs an empty array list with the specified capacity.

- `boolean add(E obj)`

appends `obj` at the end of the array list. Always returns true.

- `int size()`

returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)

- `void ensureCapacity(int capacity)`

ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.

- `void trimToSize()`

reduces the storage capacity of the array list to its current size.

- Accessing ArrayList elements
- `mylist.get(index);`
- `mylist.set(index,val);`
- `ArrayList.toString()` already defined.  
Prints comma separated array element values.

# Interfaces

```
public class Employee implements Comparable<Employee>
{
    private String name;
    private double salary;

    public Employee(String name, double salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    /**
     * Compares employees by salary
     * @param other another Employee object
     * @return a negative value if this employee has a lower salary than
     * otherObject, 0 if the salaries are the same, a positive value otherwise
     */
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
}
```

# Interfaces

```
x = new Comparable(. . .); // ERROR
```

```
Comparable x; // OK
```

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

# Interfaces – Single Inheritance with Multiple Interface Implementations.

- public interface Comparable{
  - public int compareTo(Object otherobject);
  - }
- 
- class Manager extends Employee implements Comparable {
  - ....
  - }



Can be as many as the programmer feels like.

# Interfaces –Default Methods

- *Somewhat* like a default constructor, but you cannot instantiate objects of interfaces!

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
        // by default, all elements are the same
}

public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove() { throw new UnsupportedOperationException("remove"); }
    ...
}
```

# Popular Use Case – Callbacks.

- Callback function frameworks use interfaces.
- Event listeners – need to implement these interfaces and interface functions.

java.awt.event

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

# Popular Use Case – Callbacks.

```
import java.awt.*;
import java.awt.event.*;
import java.time.*;
import javax.swing.*;
```

```
public class TimerTest
{
    public static void main(String[] args)
    {
        var listener = new TimePrinter();

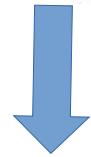
        // construct a timer that calls the listener
        // once every second
        var timer = new Timer(1000, listener);
        timer.start();

        // keep program running until the user selects "OK"
        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}
```

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
                           + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

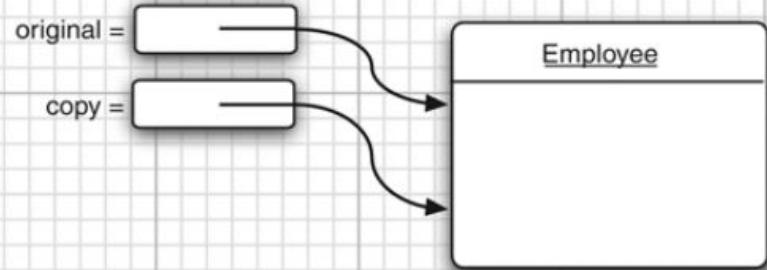
# Object Cloning

```
var original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

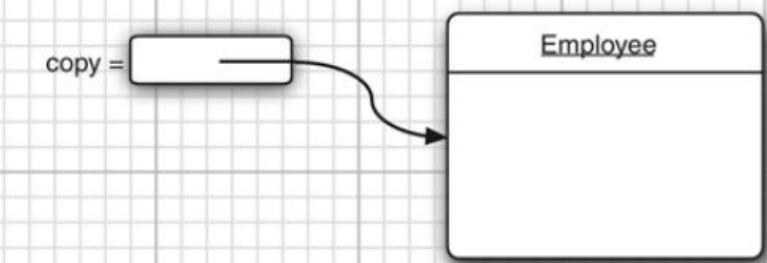
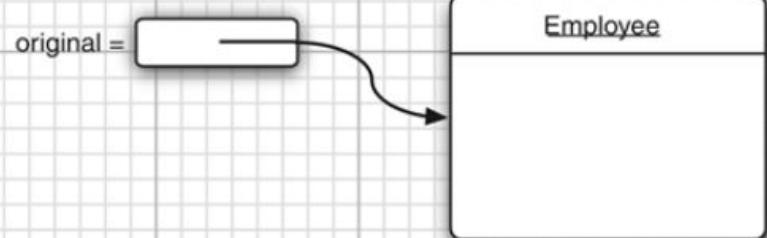


```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

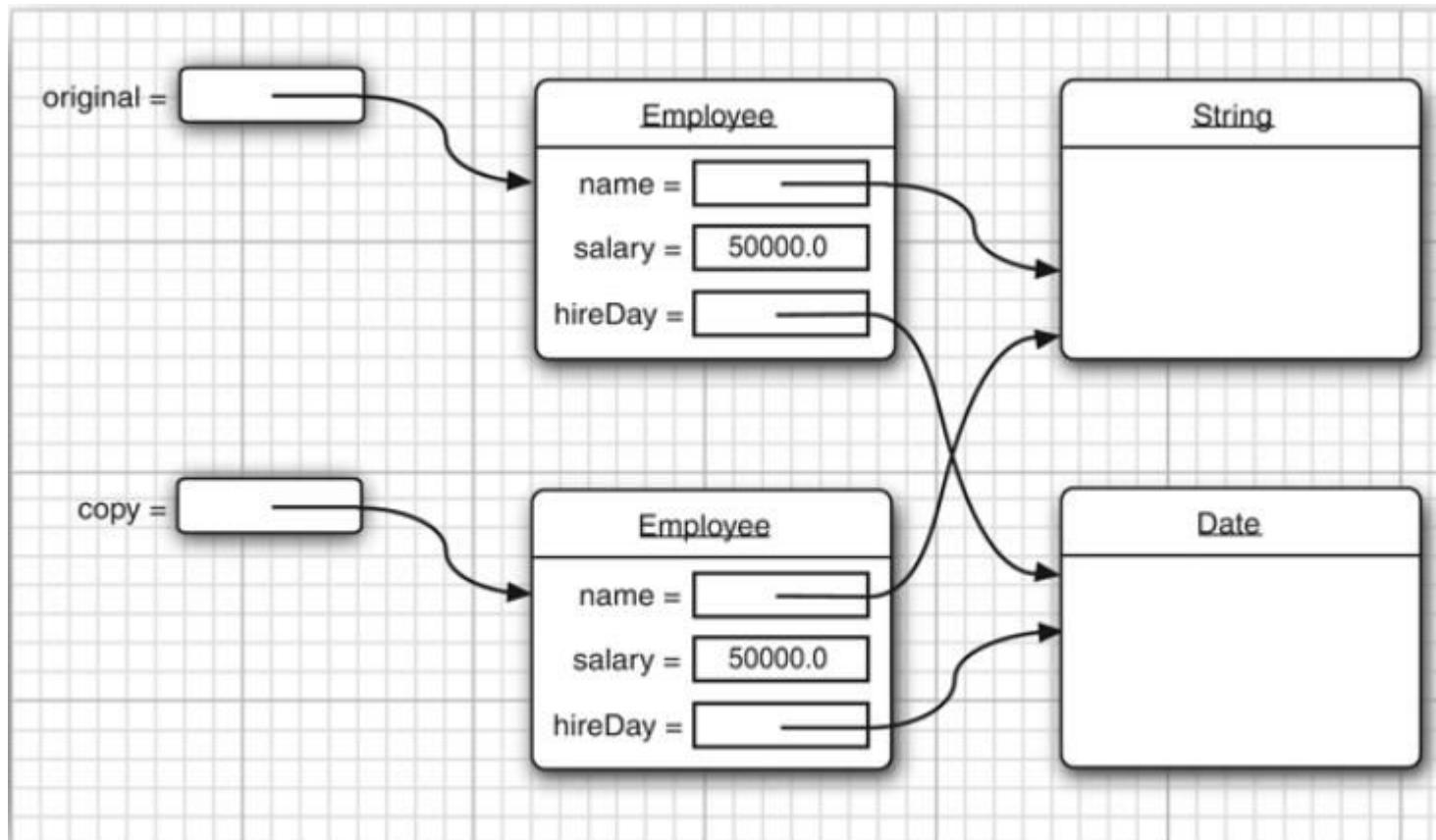
Copying



Cloning



# Object Cloning – Default:Shallow Copy



# Object Cloning – Deep Copy: Implement Clonable

```
class Employee implements Cloneable
{
    // public access, change return type
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    ...
}

class Employee implements Cloneable
{
    ...
    public Employee clone() throws CloneNotSupportedException
    {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

# *Advanced Programming*

## CSE 201

### Instructor: Sambuddho

(Semester: Monsoon 2024)

Week 4 – Inner Classes  
and Exceptions/Exception  
Handling

# Inner Classes

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

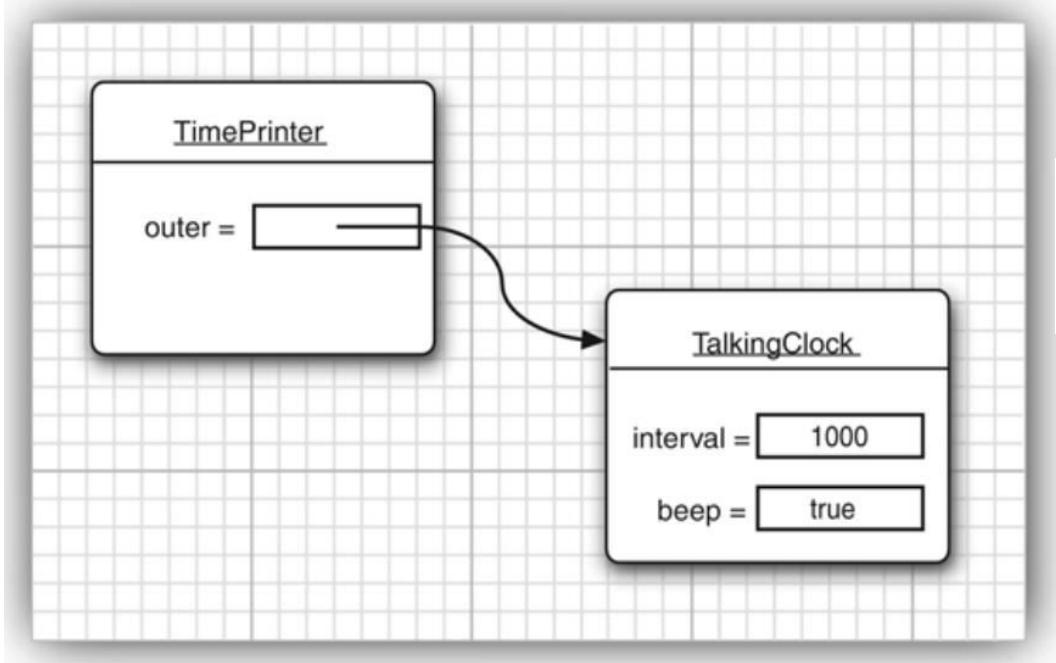
    public class TimePrinter implements ActionListener
        // an inner class
    {
        . . .
    }
}
```

# Inner Classes

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

Implicit reference to object of outer class. However the correct way to use is to refer with object of the outer class.

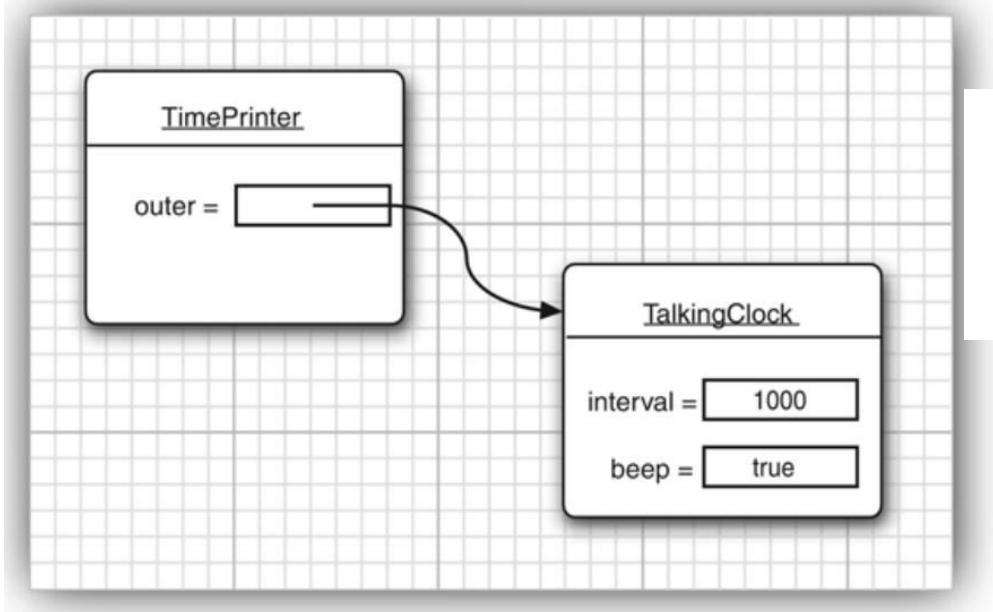
# Inner Classes



```
public TimePrinter(TalkingClock clock) // automatically generated code
{
    outer = clock;
}
```

Accessible with an object of the  
outer class in the inner.

# Inner Classes



```
public void actionPerformed(ActionEvent event)
{
    . . .
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

Accessible without requiring `in` object. Outer class encapsulates the inner class as well.

# Anonymous Class

```
new SuperType(construction parameters)
{
    inner class methods and data
}

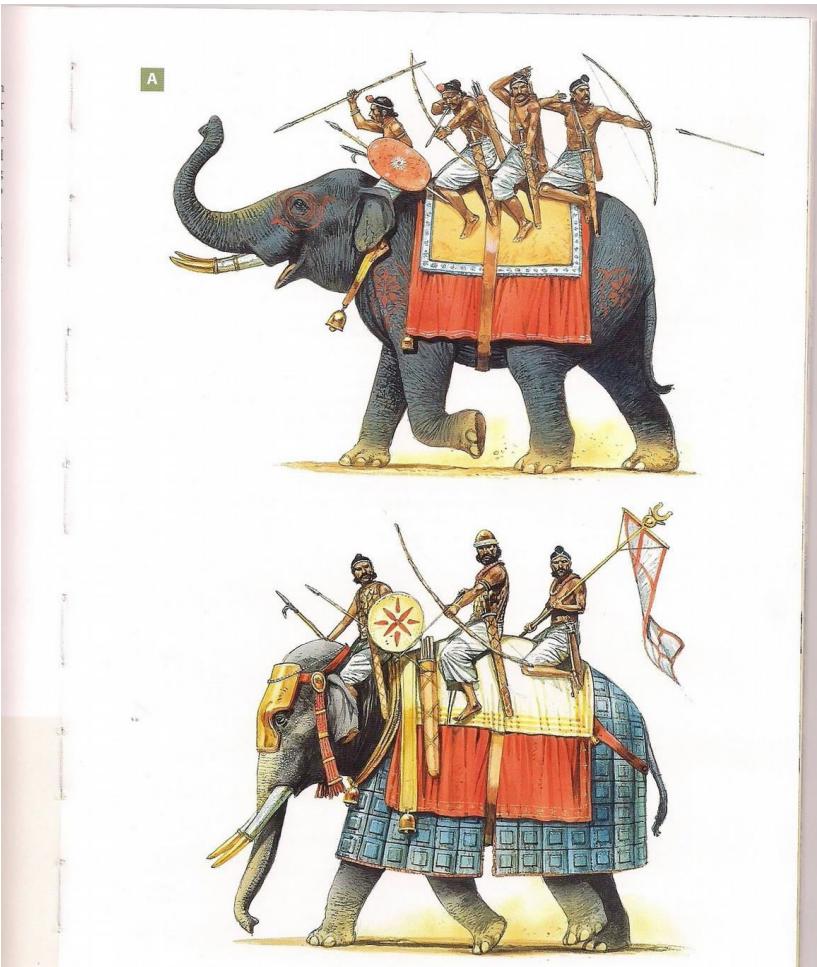
public void start(int interval, boolean beep)
{
    var listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    var timer = new Timer(interval, listener);
    timer.start();
}
```

# Exceptions and Exception Handling

OOP avatar of errors and error handling.

- Array out of bounds.
- IOException – e.g. File not found error.
- Mathematical exception – divide by zero error, NAN error, floating point exception *etc.*

# Being Defensive is Important

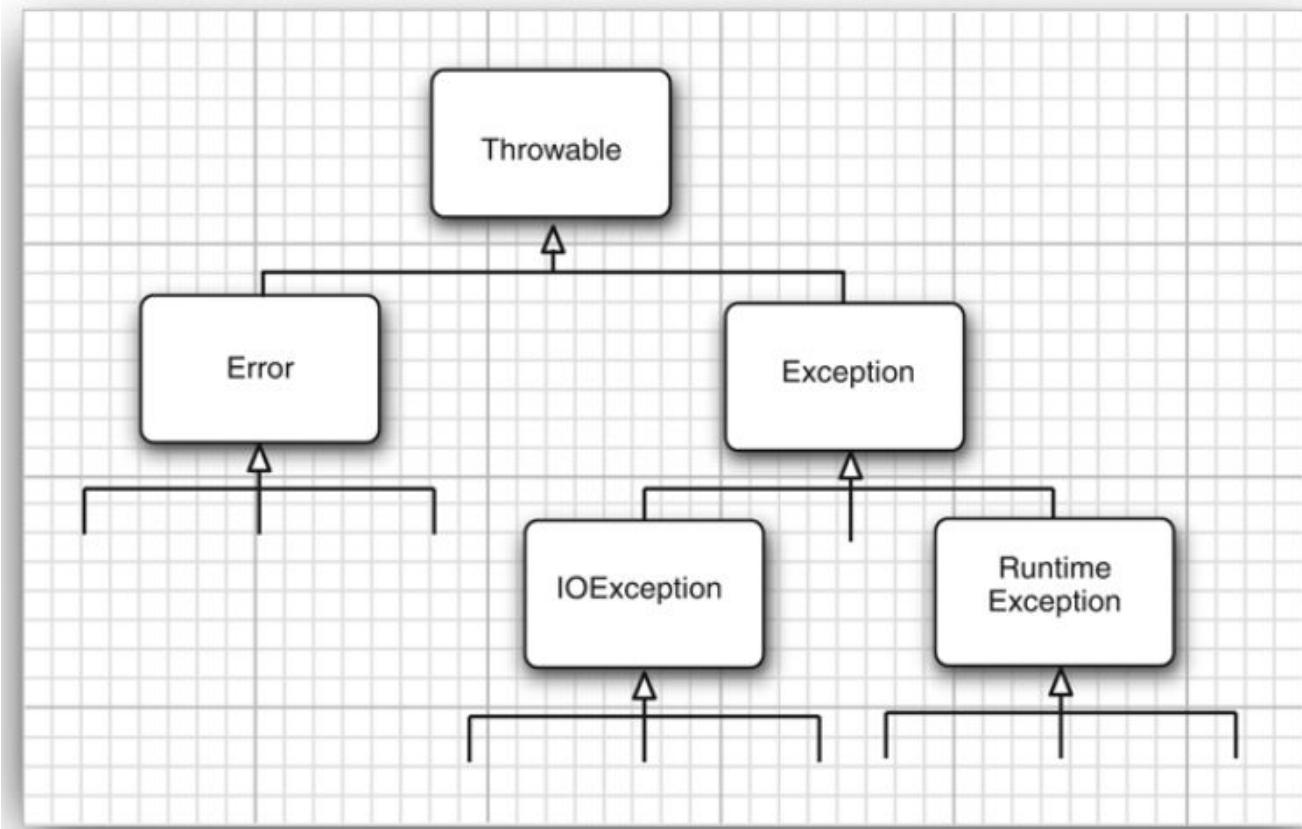


www.shutterstock.com · 109665452

# Exception Handling Syntax

- Process for handling exceptions
  - `try` some code, catch exception thrown by tried code, finally, “clean up” if necessary
  - `try`, `catch`, and `finally` are reserved words
- `try` denotes code that may throw an exception
  - place questionable code within a `try` block
  - a `try` block must be immediately followed by a `catch` block unlike an if w/o else thus,
  - `try-catch` blocks always occurs as pairs
- `catch` exception thrown in `try` block and write special code to handle it
  - catch blocks distinguished by type of exception
  - can have several **`catch blocks`**, each specifying a particular type of exception Once an
  - exception is handled, execution continues after the catch block
- `finally` (optional)
  - special block of code that is executed whether or not an exception is thrown
  - follows `catch block`

# Exceptions and Exception Handling



Exceptions are  
*also* classes

# Exceptions Handling by JVM

- Any method invocation is represented as a “**stack frame**” on the Java “**stack**”
- **Callee-Caller** relationship: If method A calls method B then A is **caller** and B is **callee**
- Each frame stores local variables, input parameters, return values and intermediate calculations
  - In addition, each frame also stores an “**exception table**”
  - This exception table stores information on each try/catch/finallyblock, i.e. the instruction offset where the catch/finally blocks are defined.

How JVM handles exceptions:

1. Look for exception handler in current stack frame (method)
2. If not found, then terminate the execution of current method and go to the callee method and repeat step 1 by looking into callee’s exception table
3. If no matching handler is found in any stack frame, then JVM finally terminates by throwing the stack trace (printStackTracemethod)

# Exception Handling Syntax

- Process for handling exceptions
  - `try` some code, catch exception thrown by tried code, finally, “clean up” if necessary
  - `try`, `catch`, and `finally` are reserved words
- `try` denotes code that may throw an exception
  - place questionable code within a `try` block
  - a `try` block must be immediately followed by a `catch` block unlike an if w/o else thus,
  - `try-catch` blocks always occurs as pairs
- `catch` exception thrown in `try` block and write special code to handle it
  - catch blocks distinguished by type of exception
  - can have several **`catch blocks`**, each specifying a particular type of exception Once an
  - exception is handled, execution continues after the catch block
- `finally` (optional)
  - special block of code that is executed whether or not an exception is thrown
  - follows `catch block`

# Exceptions and Exception Handling

Methods tells Java compiler that what kind of errors it can throw.

```
class MyAnimation
{
    ...
    public Image loadImage(String s) throws FileNotFoundException, EOFException
    {
        ...
    }
}
```

# Throwing an Exception

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (. . .)
    {
        if (!in.hasNext()) // EOF encountered
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

# Creating and Throwing Exception Class

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}

String readData(Scanner in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // EOF encountered
        {
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

# Catching What Was Thrown

```
try
{
    code
    more code
    more code
}
catch (ExceptionType e)
{
    handler for this type
}
```

In try {} code after the throw is skipped.

The program jumps to the catch() handler.

If no appropriate handler, then JRE handles it and show it on stdout

# Catching What Was Thrown

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException e)
{
    emergency action for missing files
}
catch (UnknownHostException e)
{
    emergency action for unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException | UnknownHostException e)
{
    emergency action for missing files and unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

Combining exceptions

# Rethrowing Exceptions

```
try
{
    access the database
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

# Finally Clause

- Executed at the end of all try{} catch{} blocks.
- Last set of instructions to be called before the program terminates. Usually used for resource release and cleanup operations.

# Finally Clause

```
var in = new FileInputStream(. . .);
try
{
    // 1
    code that might throw exceptions
```

```
    // 2
}
catch (IOException e)
{
    // 3
    show error message
    // 4
}
finally
{
    // 5
    in.close();
}
// 6
```

# Using Assertions

- Idiomatic tools for defensive programming.
- Faster execution than throwing exceptions.
- Not to be used for everyday programs.
- Irrecoverable.
- Usually program terminates.

```
assert <condition>;
```

```
assert <condition> : <expression> ;
```

[ Check condition. If false then create an object with argument <expression> of type AssertionException – JVM catches it and prints the details presented in the <expression> ]

# Using Assertions

```
if (x > 0){  
    ....  
}  
Else {  
    ...  
}  
  
    if (x > 0){  
        throw new  
exception("myexception");  
    }  
  
catch(Exception ep){  
    ...  
}
```

assert x>0 : new String("x>0");

# Logging

- `System.out.println()` cannot be always used – makes things slow.
- Adding and removing them at all places can be cumbersome.
- Usually not used in production code.
- Logging can be collectively enabled or suppressed.

```
Logger.getLogger().info("File->Open menu item selected");
```

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen
INFO: File->Open menu item selected
```

# Log Levels

SEVERE

WARNING

INFO

CONFIG

FINE

FINER

FINEST

Top -> Bottom levels of logging.

If you log a bottom level,  
then you log all levels above it.

SEVERE, WARNING and INFO  
are always enable for every  
Java program by default.

```
logger.warning(msg);  
logger.fine(msg);
```

```
logger.log(Level.<Levelname>,  
msg);
```

# Logging Unexpected Exceptions

Two methods commonly used:

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)

try
{
    if (. . .)
    {
        var e = new IOException(" . . .");
        logger.throwing("com.mycompany.mylib.Reader", "read", e);
        throw e;
    }
}

catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

# *Advanced Programming*

**CSE 201**

**Instructor: Sambuddho**

(Semester: Monsoon 2024)

Week 5 – Generic Classes

# Generic Container to Hold Different Types ?

- By using any of the concepts taught till now in this course, how can you store different types of objects in a same datastructure
  - E.g., String, Integer, Float, etc. ?

# Approach 1

```
public class MyGenericList { private ArrayList myList; public  
    MyGenericList() {  
        myList = new ArrayList();  
    }  
    public void add(Object o) { myList.add(o);  
    }  
    public Object get(int i) { return myList.get(i);  
    }  
  
    public static void main(String[] args) { MyGenericList generic = new  
MyGenericList();  
        generic.add("hello"); generic.add(10); generic.add(10.23f);  
        .....  
        String str = (String) generic.get(0); // OK String str = (String)  
        generic.get(1); //  
NOT OK  
    }
```

- Using inheritance we know Object class can hold any type of objects
  - We can create ArrayList of objects
- Problems we face:
  - Mandatory type casting while getting the object from list
  - No error checking while adding objects as we are allowed to add any type of objects
  - Wrong type casting can land you with runtime errors

# Generic Programming

- Code that can be reused. Need not be rewritten for individual types.
- Same class and methods can be used for multiple types (non primitive).
- Avoid generic casting errors.

```
public class ArrayList // before generic classes
{
    private Object[] elementData;
    ...
    public Object get(int i) { . . . }
    public void add(Object o) { . . . }
}
```

```
ArrayList files = new ArrayList();
files.add(new File("..."));
String filename = (String) files.get(0);
```

# Solution: Generic Programming



- Our generic cup can hold different types of liquid
- In the notation       $\text{Cup}\langle T \rangle$ :
  - $T = \text{Coffee}$
  - $T = \text{Tea}$
  - $T = \text{Milk}$
  - $T = \text{Soup}$
  - $\dots$

Cup == Generic Container

# Generic Programming

```
var files = new ArrayList<String>();
```

Or

```
ArrayList<String> files = new ArrayList();
```

```
public class Pair<T, U> { . . . }
```



```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

# Generic Methods

```
class ArrayAlg
{
    public static <T> T getMiddle(T... a)
    {
        return a[a.length / 2];
    }
}
```

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

# Solution to our Problem

```
public class MyGenericList <T> { private ArrayList <T>
    myList; public MyGenericList() {
        myList = new ArrayList
    <T>();
    }
    public void add(T o) { myList.add(o);
    }
    public T get(int i) { return myList.get(i);
    }
}
```

- Using generic programming we don't have to implement different classes for different object types.
  - Programmer friendly code!
- We just have to create different instances of MyGenericList for different objects.

```
public class Main {
    public static void main(String args[]) {
        MyGenericList<String> strList = new
        MyGenericList<String>();
        MyGenericList<Integer> intList = new
        MyGenericList<Integer>();

        strList.add("hello"); intList.add(1);
        ...
    }
}
```

# Generic Class with Two Fields (1/3)

```
public class Pair <T1, T2> { private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) { key = _k; value = _v;  
    }  
    public T1 getKey() { return key; } public T2 getValue()  
    { return value; } }
```

- Why this code isn't correct?
  - MyGenericList class instantiated without specifying the type of its two fields

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair> db =  
            new MyGenericList<Pair>();  
        db.add(new Pair<String, Integer>("John", 2343)); db.add(new Pair<String, Integer>("Susane",  
            8908));  
        ...  
    } }
```

# Generic Class with Two Fields (2/3)

```
public class Pair <T1, T2> { private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) { key = _k; value = _v;  
    }  
    public T1 getKey() { return key; } public T2 getValue() {  
        return value;  
    }
```

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair<String, Integer>> db = new MyGenericList<Pair>();  
        db.add(new Pair<String, Integer>("John", 2343)); db.add(new Pair<String, Integer>("Susane",  
        8908));  
        ...  
    }  
}
```

- Why this code isn't correct
  - During instantiation we have to declare the type of fields in MyGenericList class on both RHS and LHS of statement

# Generic Class with Two Fields (3/3)

```
public class Pair <T1, T2> { private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) { key = _k; value = _v;  
    }  
    public T1 getKey() { return key; } public T2 getValue() {  
        return value; }  
}
```

- This is the correct implementation and usage of a generic class with multiple fields

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair<String, Integer>> db =  
            new MyGenericList<Pair<String, Integer>>(); db.add(new Pair<String,  
            Integer>("John", 2343)); db.add(new Pair<String, Integer>("Susane", 8908));  
            ...  
    }  
}
```

# Why Generic Array Creation not Allowed ?

```
// Legal statement (arrays are covariant) Object array[] = new Integer[10];
// Compilation error below (generics are invariant)
```

```
List<Object> myList = new ArrayList<Integer>();
```

// Below line incorrect but let's assume its correct

```
List<Integer> intList[] = new ArrayList<Integer>[5]; List<String> stringList = new
ArrayList<String>();
```

```
stringList.add("John");
```

```
Object objArray[] = intList; objArray[0] = stringList;
```

```
// This will generate ClassCastException
int my_int_number = objArray[0].get()
```

- Arrays are covariant
  - Subclass array type can be assigned to superclass array reference
- Generics are invariant
  - Subclass type generic type cannot be assigned to superclass generic reference.
- If generic array creation was allowed then compile time strict type checking cannot be enforced.
  - Runtime ClassCastException will be generated in the example here.

# Bounds for Type Variables

```
class ArrayAlg
{
    public static <T> T min(T[] a) // almost correct
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}

public static <T extends Comparable> T min(T[] a) . . .
```

T extends Comparable & Serializable

## Issues?

Does an object of arbitrary type have a method `compareTo()`?

Adding multiple bounds

# Type Erasures – Basis of Generic Programming

Rule: *Erase* and replace generic type with a *raw type* (for bounded types) and *Object* for unbounded.

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```



```
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }

    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
}
```

# Type Erasures – Basis of Generic Programming

Rule: *Erase* and replace generic type with a *raw type* (for bounded types) and *Object* for unbounded.

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    private T lower;
    private T upper;
    ...
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
}
```



```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    ...
    public Interval(Comparable first, Comparable second) { . . . }
}
```

# Type Erasures – Implicit Casting

Step 1: Call to raw method `Pair.getFirst()`;

Step 2: Cast returned object to type `Object`.

```
Pair<Employee> epairs = new Pair<>;  
Employee epair = epairs.getFirst();
```

# Type Erasures – Translating Generic Methods

```
public static <T extends Comparable> T min(T[] a) → public static Comparable min(Comparable[] a)
```

```
class DateInterval extends Pair<LocalDate>
{
    public void setSecond(LocalDate second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . .
}
```



```
class DateInterval extends Pair // after erasure
{
    public void setSecond(LocalDate second) { . . . }
    . .
}
```

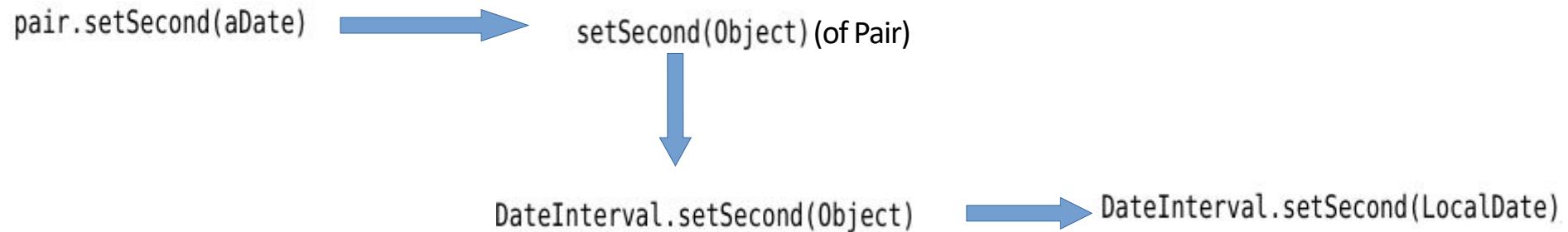
But Pair also has `setSecond(Object second);!!`

**Erasure interferes with polymorphism!!**

## Type Erasures – Translating Generic Methods – Bridge Methods

The compiler generates a *bridge method* in the DateInterval class.

```
public void setSecond(Object second) { setSecond((LocalDate) second); }
```



# *Advanced Programming*

## CSE 201

**Instructor: Sambuddho**

(Semester: Monsoon 2024)

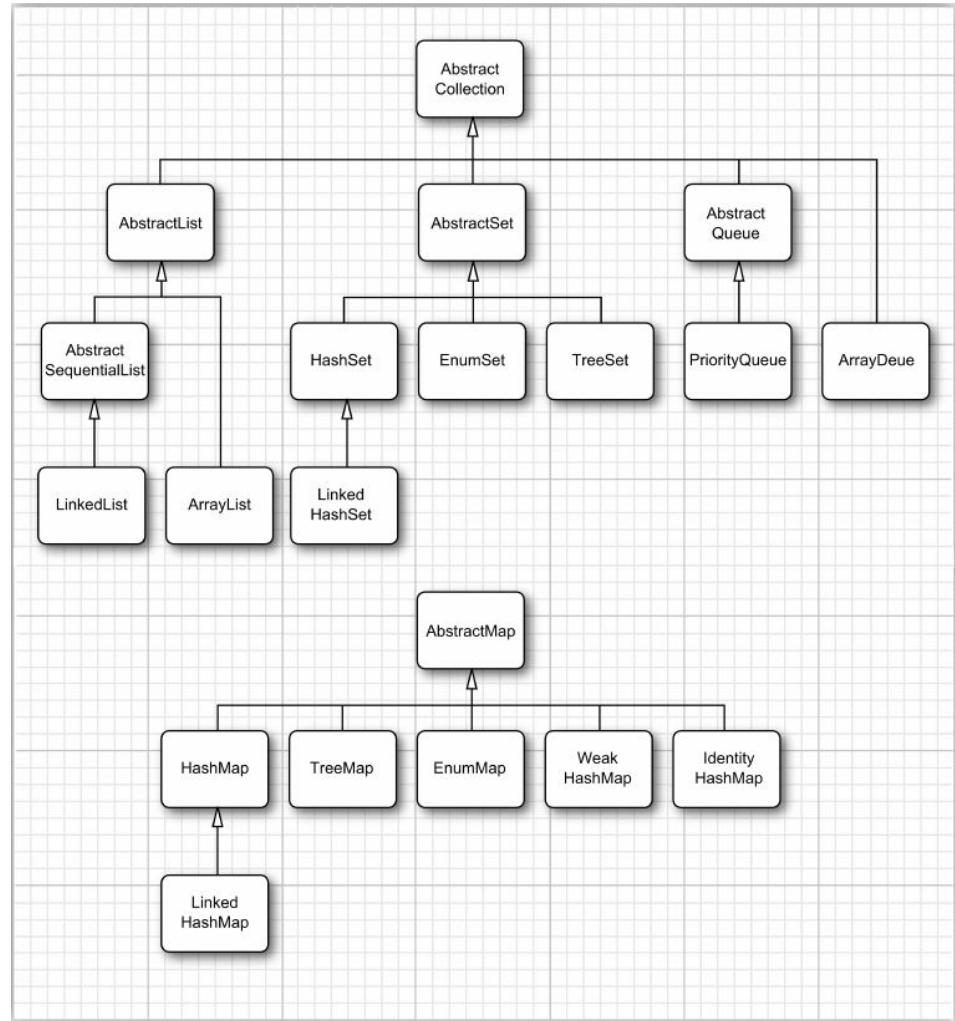
Week 6 – Collections

# Concrete Collections

- Concrete generic classes, useful for various purposes.

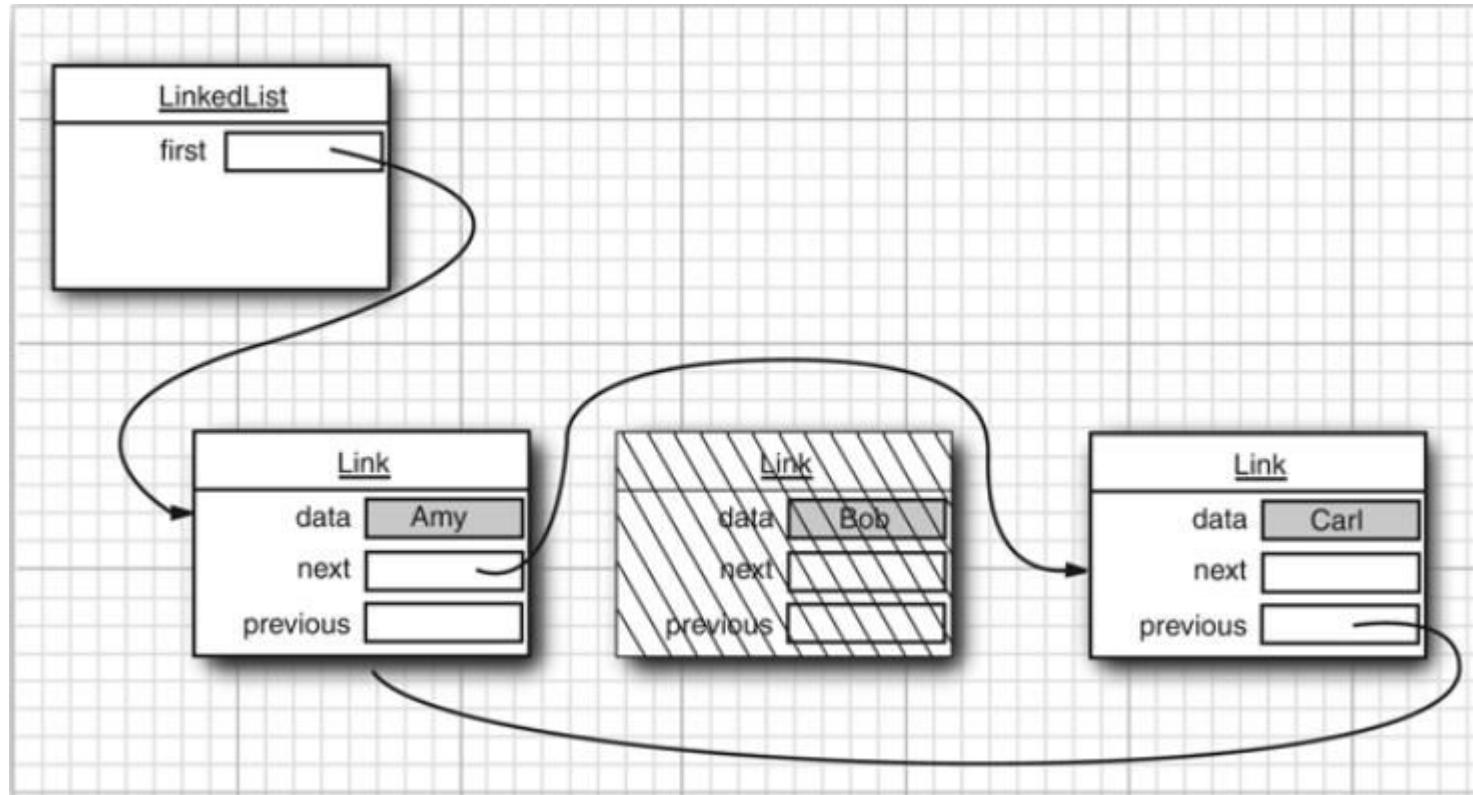
| Collection Type | Description                                                                                     |
|-----------------|-------------------------------------------------------------------------------------------------|
| ArrayList       | An indexed sequence that grows and shrinks dynamically                                          |
| LinkedList      | An ordered sequence that allows efficient insertion and removal at any location                 |
| ArrayDeque      | A double-ended queue that is implemented as a circular array                                    |
| HashSet         | An unordered collection that rejects duplicates                                                 |
| TreeSet         | A sorted set                                                                                    |
| EnumSet         | A set of enumerated type values                                                                 |
| LinkedHashSet   | A set that remembers the order in which elements were inserted                                  |
| PriorityQueue   | A collection that allows efficient removal of the smallest element                              |
| HashMap         | A data structure that stores key/value associations                                             |
| TreeMap         | A map in which the keys are sorted                                                              |
| EnumMap         | A map in which the keys belong to an enumerated type                                            |
| LinkedHashMap   | A map that remembers the order in which entries were added                                      |
| WeakHashMap     | A map with values that can be reclaimed by the garbage collector if they are not used elsewhere |
| IdentityHashMap | A map with keys that are compared by ==, not equals                                             |

# Concrete Collections



# Concrete Collections

- `LinkedList <T>`: In Java everything is a doubly linked list



# Concrete Collections

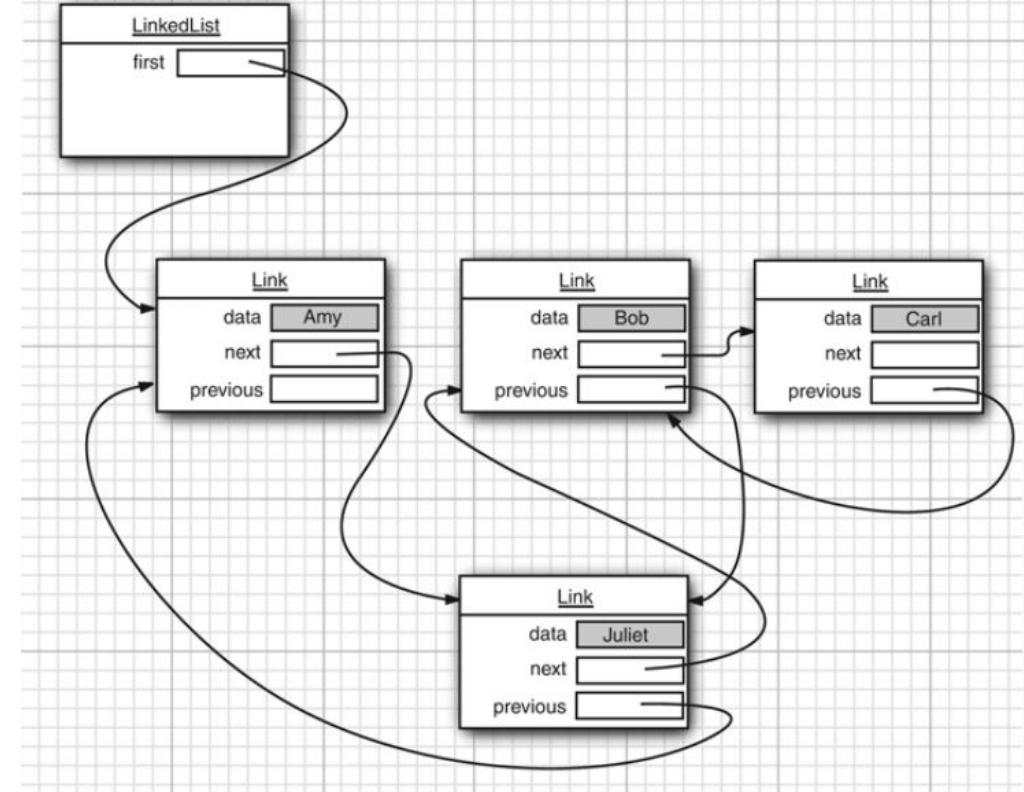
```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator<String> iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    ...
}
```

```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```

- `LinkedList.add()` always adds at the end. What if you want to add in the middle ?

# Concrete Collections



- `LinkedList.add()` always adds at the end. The `ListIterator<E>` has the responsibility to add() anywhere.

# Concrete Collections: ArrayLists

Ordered list of indexable items; much like a generic array of variable length.

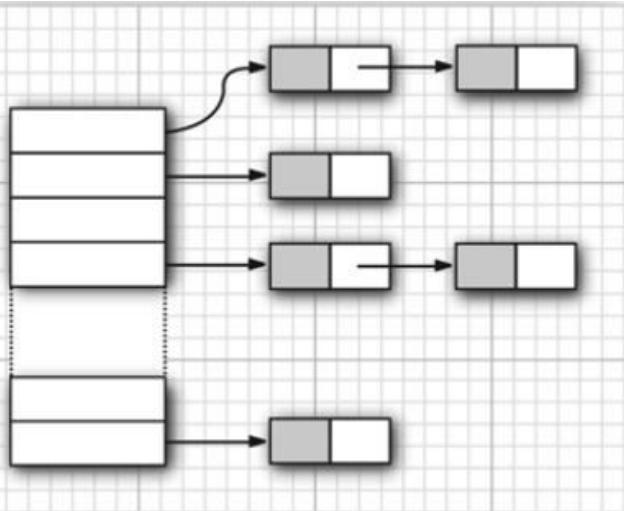
# ArrayList Methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - `Object get(int index)`
  - `Object set(int index, Object element)`
  - `May throw IndexOutOfBoundsException`
- Indexed add and remove are provided, but can be costly if used frequently
  - `void add(int index, Object element)`
  - `Object remove(int index)`
  - `May throw IndexOutOfBoundsException`
- May want to resize in one shot if adding many elements
  - `void ensureCapacity(int minCapacity)`
- ArrayList allows adding duplicate elements

# Sets

- Sets keep unique elements only
  - Like lists but no duplicates
- HashSet<E>
  - Keeps a set of elements in a hash tables
  - The elements are randomly ordered by their hash code
- TreeSet<E>
  - Keeps a set of elements in a red-black ordered search tree
  - The elements are ordered incrementally

# Concrete Collections: HashSet



| Modifier and Type | Method and Description                                                                                     |
|-------------------|------------------------------------------------------------------------------------------------------------|
| boolean           | <b>add(E e)</b><br>Adds the specified element to this set if it is not already present.                    |
| void              | <b>clear()</b><br>Removes all of the elements from this set.                                               |
| Object            | <b>clone()</b><br>Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| boolean           | <b>contains(Object o)</b><br>Returns true if this set contains the specified element.                      |
| boolean           | <b>isEmpty()</b><br>Returns true if this set contains no elements.                                         |
| Iterator<E>       | <b>iterator()</b><br>Returns an iterator over the elements in this set.                                    |
| boolean           | <b>remove(Object o)</b><br>Removes the specified element from this set if it is present.                   |
| int               | <b>size()</b><br>Returns the number of elements in this set (its cardinality).                             |

# Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface

# HashSet

- Find and add elements very quickly
  - uses hashing
- Hashing uses an array of linked lists
  - The **hashCode()** is used to index into the array
  - Then **equals()** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- Behaves like a set of unique elements

# HashSet

```
import java.util.HashSet;

HashSet<String> students = new HashSet<String>();

students.add("Alice");
students.add("Bob");
students.add("Charlie");
System.out.println(students);

students.contains("Bob"); //boolean outcome.

students.remove("Alice"); //boolean "could it be
                        //removed or not"

students.clear();    //clear the set

students.size();     //the size in terms of elements.
```

# Concrete Collections: TreeSet

- Similar to HashSet but all tree values are a sorted collection of generic object types. The object types must implement Comparable.

```
var sorter = new TreeSet<String>();  
sorter.add("Bob");  
sorter.add("Amy");  
sorter.add("Carl");  
for (String s : sorter) System.out.println(s);
```

String sorter tree

# Concrete Collections: TreeSet

```
package treeSet;

import java.util.*;

/**
 * This program sorts a set of Item objects by comparing their descriptions.
 * @version 1.13 2018-04-10
 * @author Cay Horstmann
 */
public class TreeSetTest
{
    public static void main(String[] args)
    {
        var parts = new TreeSet<Item>();
        parts.add(new Item("Toaster", 1234));
        parts.add(new Item("Widget", 4562));
        parts.add(new Item("Modem", 9912));
        System.out.println(parts);

        var sortByDescription = new TreeSet<Item>(Comparator.comparing(Item::getDescription));

        sortByDescription.addAll(parts);
        System.out.println(sortByDescription);
    }
}
```

```
public class Item implements Comparable<Item>
{
    private String description;
    private int partNumber;

    /**
     * Constructs an item.
     * @param aDescription the item's description
     * @param aPartNumber the item's part number
     */
    public Item(String aDescription, int aPartNumber)
    {
        description = aDescription;
        partNumber = aPartNumber;
    }

    /**
     * Gets the description of this item.
     * @return the description
     */
    public String getDescription()
    {
        return description;
    }

    public String toString()
    {
        return "[description=" + description + ", partNumber=" + partNumber + "]";
    }

    public boolean equals(Object otherObject)
    {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        var other = (Item) otherObject;
        return Objects.equals(description, other.description) && partNumber == other.partNumber;
    }

    public int hashCode()
    {
        return Objects.hash(description, partNumber);
    }

    public int compareTo(Item other)
    {
        int diff = Integer.compare(partNumber, other.partNumber);
        return diff != 0 ? diff : description.compareTo(other.description);
    }
}
```

# Concrete Collections: PQs

Similar to TreeSet. Insert in arbitrary order, remove the smallest first – much like ‘heap’ data structure.

```
package priorityQueue;

import java.util.*;
import java.time.*;

/**
 * This program demonstrates the use of a priority queue.
 * @version 1.02 2015-06-20
 * @author Cay Horstmann
 */
```

```
public class PriorityQueueTest
{
    public static void main(String[] args)
    {
        var pq = new PriorityQueue<LocalDate>();
        pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
        pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
        pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
        pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse

        System.out.println("Iterating over elements . . .");
        for (LocalDate date : pq)
            System.out.println(date);
        System.out.println("Removing elements . . .");
        while (!pq.isEmpty())
            System.out.println(pq.remove());
    }
}
```

---

# Maps

- Stores unique key/value pairs.
- Maps from the key to the value.
- Keys are unique.
  - a single key only appears once in the Map.
  - a key can map to only one value.
  - Values do not have to be unique.
- `HashMap<K, V>`: Keeps a map of elements in a hash table; elements randomly ordered
- `TreeMap<K, V>`: Keep a set of elements in a red-black ordered search tree.

# HashMap examples

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {

        HashMap<String, Integer> namesRollNos = new HashMap<String, Integer>();

        namesRollNos.put("Alice", new Integer(100));
        namesRollNos.put("Bob", new Integer (101));
        namesRollNos.put("Charlie", new Integer(102));
        System.out.println(namesRollNos);
        System.out.println(namesRollNos.get("Charlie"));
    }
}
```

# HashMap examples

|                                              |                                                           |                                                                                                                           |
|----------------------------------------------|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| void                                         | <code>clear()</code>                                      | Removes all of the mappings from this map.                                                                                |
| <code>Object</code>                          | <code>clone()</code>                                      | Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.                           |
| boolean                                      | <code>containsKey(Object key)</code>                      | Returns true if this map contains a mapping for the specified key.                                                        |
| boolean                                      | <code>containsValue(Object value)</code>                  | Returns true if this map maps one or more keys to the specified value.                                                    |
| <code>Set&lt;Map.Entry&lt;K,V&gt;&gt;</code> | <code>entrySet()</code>                                   | Returns a <code>Set</code> view of the mappings contained in this map.                                                    |
| <code>V</code>                               | <code>get(Object key)</code>                              | Returns the value to which the specified key is mapped, or <code>null</code> if this map contains no mapping for the key. |
| boolean                                      | <code>isEmpty()</code>                                    | Returns true if this map contains no key-value mappings.                                                                  |
| <code>Set&lt;K&gt;</code>                    | <code>keySet()</code>                                     | Returns a <code>Set</code> view of the keys contained in this map.                                                        |
| <code>V</code>                               | <code>put(K key, V value)</code>                          | Associates the specified value with the specified key in this map.                                                        |
| void                                         | <code>putAll(Map&lt;? extends K,? extends V&gt; m)</code> | Copies all of the mappings from the specified map to this map.                                                            |
| <code>V</code>                               | <code>remove(Object key)</code>                           | Removes the mapping for the specified key from this map if present.                                                       |
| int                                          | <code>size()</code>                                       | Returns the number of key-value mappings in this map.                                                                     |
| <code>Collection&lt;V&gt;</code>             | <code>values()</code>                                     | Returns a <code>Collection</code> view of the values contained in this map.                                               |

# TreeMap

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements  
NavigableMap<K,V>, Cloneable, Serializable
```

RB Tree sorted according to the natural ordering of the keys or by the Comparator provided at the map creation time.

$\log(n)$  time insertion and searching

# TreeMap Example

```
import java.util.TreeMap;

public class Main {
    public static void main(String[] args) {

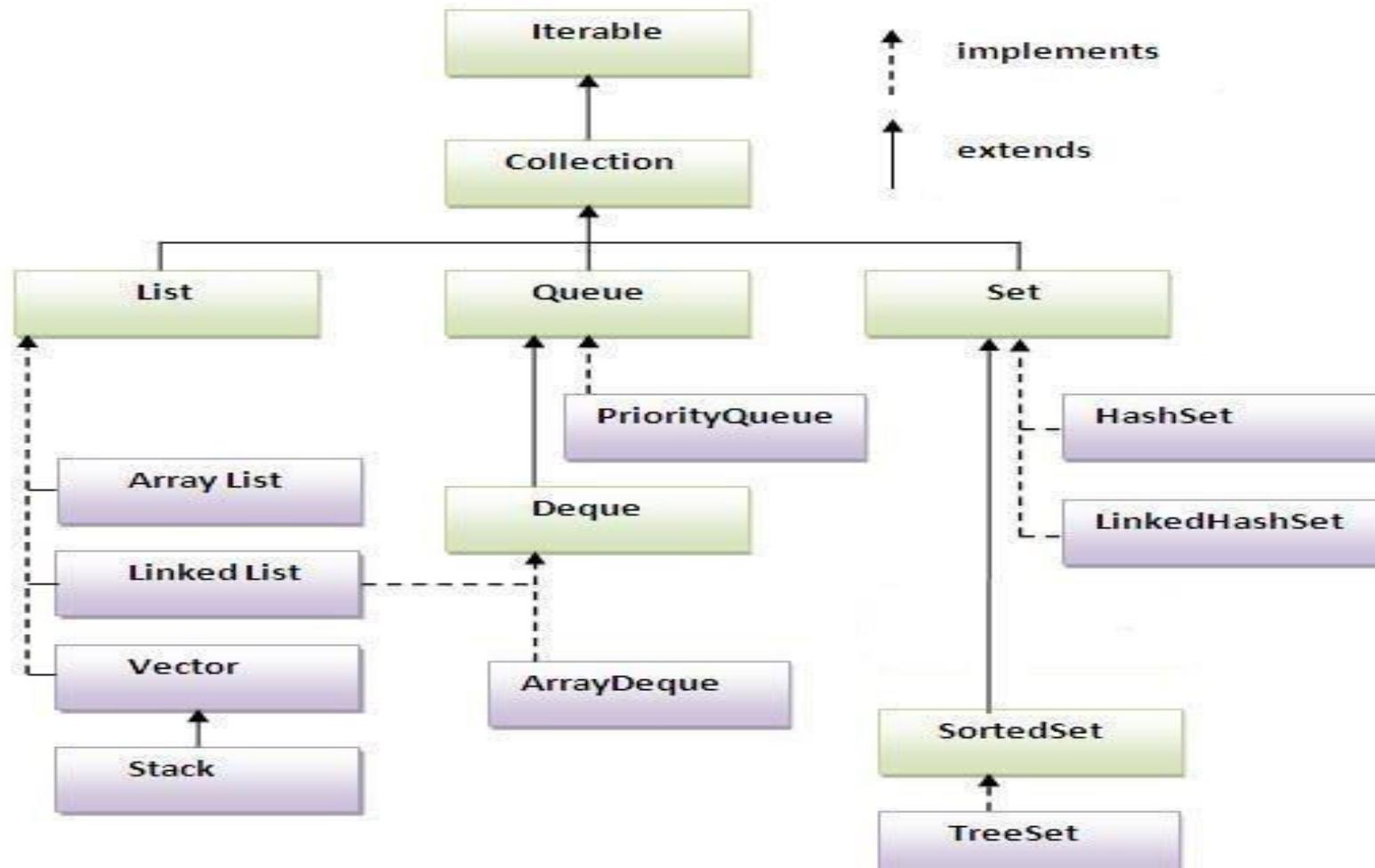
        TreeMap<String, Integer> namesRollNos = new TreeMap<>();
        namesRollNos.put("Charlie", new Integer(102));
        namesRollNos.put("Alice", new Integer(100));
        namesRollNos.put("Bob", new Integer(101));

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

Output:

{Alice=100, Bob=101, Charlie=102}

# Collection Hierarchy



## Collection Interfaces

- Interfaces for building complex data structures.
- Minimal number of own method declarations.
- Operations like add(), remove(), iterator() [To traverse complex data structures].
- Can be used for any form of complex data structures.

# Collection and Interfaces

```
public interface Queue<E> // a simplified form of the interface in the standard library
{
    void add(E element);
    E remove();
    int size();
}
```



```
public class CircularArrayQueue<E> implements Queue<E>
{
    private int head;
    private int tail;

    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private E[] elements;
}
```

```
public class LinkedListQueue<E> implements Queue<E>
{
    private Link head;
    private Link tail;

    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
}
```

```
Queue<Customer> expressLane = new CircularArrayQueue<>(100);
expressLane.add(new Customer("Harry"));
```

```
Queue<Customer> expressLane = new LinkedListQueue<>();
expressLane.add(new Customer("Harry"));
```

# Collection

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    ...
}
```

# Iterators

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    do something with element
}
```

# Iterators – forEach() / forEachRemaining – Lambda functions

```
iterator.forEachRemaining(element -> do something with element);
```

- General Lambda functions/expressions:
- Single-line code snippets that can be used as without a function name – only behaviour.

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second)
    {
        return first.length() - second.length();
    }
}
```



```
(String first, String second) ->
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

## More on Lambda functions/expressions.

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

- Parameterless lambda function

```
ActionListener listener = event ->  
    System.out.println("The time is "  
        + Instant.ofEpochMilli(event.getWhen()));  
    // instead of (event) -> . . . or (ActionEvent event) -> . . .
```

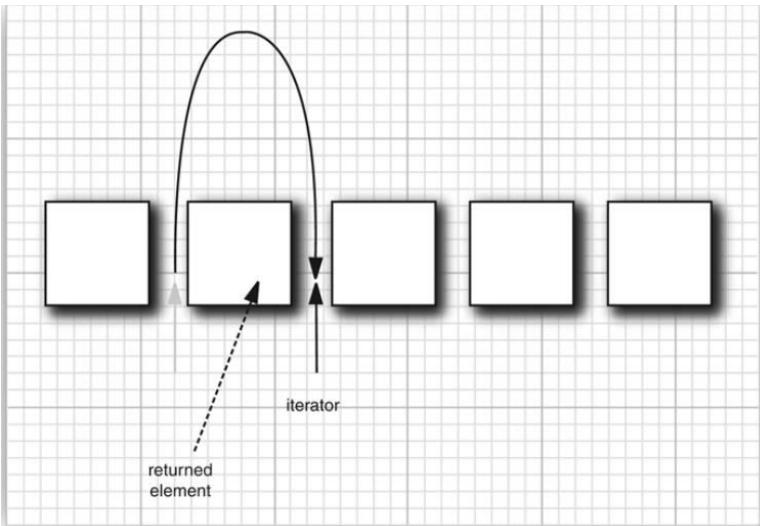
- Single parameter lambda function

# More on Lambda functions/expressions.

```
1 package lambda;
2
3 import java.util.*;
4
5 import javax.swing.*;
6 import javax.swing.Timer;
7
8 /**
9  * This program demonstrates the use of lambda expressions.
10 * @version 1.0 2015-05-12
11 * @author Cay Horstmann
12 */
13 public class LambdaTest
14 {
15     public static void main(String[] args)
16     {
17         var planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
18             "Jupiter", "Saturn", "Uranus", "Neptune" };
19         System.out.println(Arrays.toString(planets));
20         System.out.println("Sorted in dictionary order:");
21         Arrays.sort(planets);
22         System.out.println(Arrays.toString(planets));
23         System.out.println("Sorted by length:");
24         Arrays.sort(planets, (first, second) -> first.length() - second.length());
25         System.out.println(Arrays.toString(planets));
26
27         var timer = new Timer(1000, event ->
28             System.out.println("The time is " + new Date()));
29         timer.start();
30
31         // keep program running until user selects "OK"
32         JOptionPane.showMessageDialog(null, "Quit program?");
33         System.exit(0);
34     }
35 }
```

# Back to Iterators

- Overall semantics – Iterators are “in-between” elements.
- Iterator.jump() -- Jump past the next element and return the value that you jumped passed.
- 
- Iterator.remove() -- Remove the element that you just jumped passed.



```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

```
it.remove();
it.remove(); // ERROR
```

```
it.remove();
it.next();
it.remove(); // OK
```

Remove  
consecutive  
elements.

# *Advanced Programming*

**CSE 201**

**Instructor: Sambuddho**

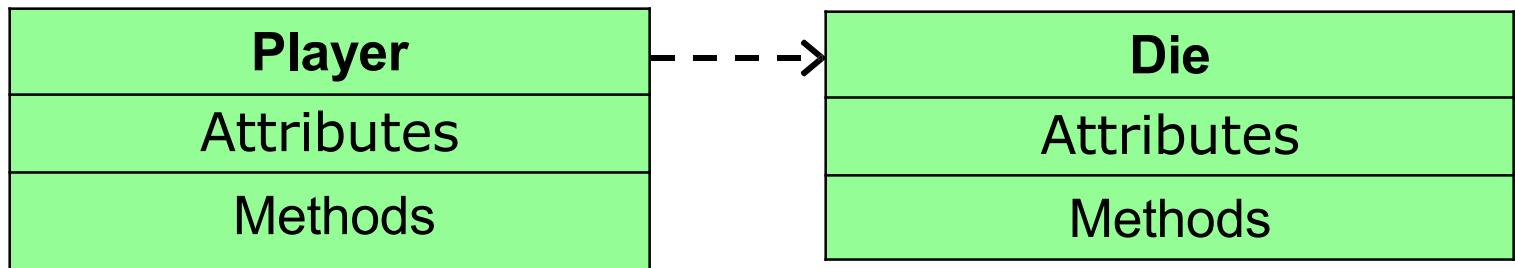
(Semester: Monsoon 2024)

Week 4 – Relationships

# UML: Quick Introduction

- UML stands for the *Unified Modeling Language*
  - We will cover this in depth in later lectures
- Much detailed than sequence diagrams
- *UML diagrams* show relationships among classes and objects
  - Lines connecting the classes
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

# A Sample UML Class Diagram



# Class Relationships

- The whole point of OOP is that your code replicates real world objects, thus making your code readable and maintainable.
- When we say real world, the real world has relationships.
- When writing a program, need to keep in mind “big picture”—how are different classes related to each other?

# Most Common Class Relationships

- **Composition**
  - A “contains” B – “part-of” relationship
- **Association/Aggregation**
  - A “knows-about” B – “uses-a” or “has-a” relationship
- **Dependency**
  - A “depends on” B – Sort of “depends-on” relationship
- **Inheritance**
  - HarleyDavidson “is-a” Bike

# Composition Relationship

- Class A **contains** object of class B
  - A **instantiates** B
- Thus A knows about B and can call methods on it
- But this is **not symmetrical!**
  - B can't automatically call methods on A
- Lifetime?
  - **The death relationship**
  - Garbage collection of A means B also gets garbage collected

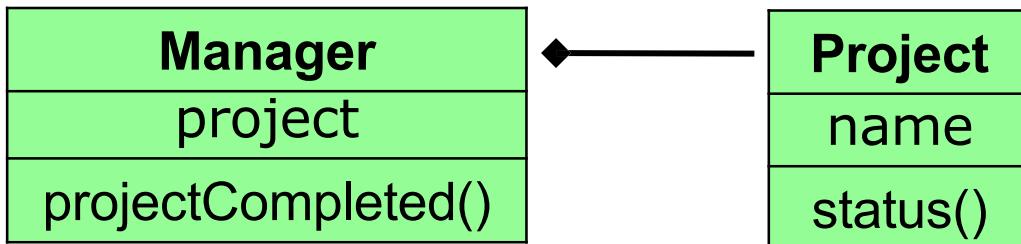
# Composition in UML

- Represented by a solid arrow with diamond head
- In below UML diagram, A is composed of B



# Composition Example (1/2)

- Manager is fixed for a project and is responsible for the timely completion of the project. If manager leaves, project is ruined



```
class Project { private String name;
    public boolean status() { ... }

    ....
}

// A manager is fixed for a project
class Manager {
    private Project project;
    public Manager() {
        this.project = new Project("ABC");
    }

    public boolean projectCompleted() {
        return project.status();
    }
}
```

# Composition Example (2/2)

- PetShop **contains** a DogGroomer
- Composition relationship because PetShop itself instantiates a DogGroomer with “new DogGroomer();”
- Since PetShop created a DogGroomer and stored it in an instance variable, all PetShop’s methods “know” about the \_groomer and can access it

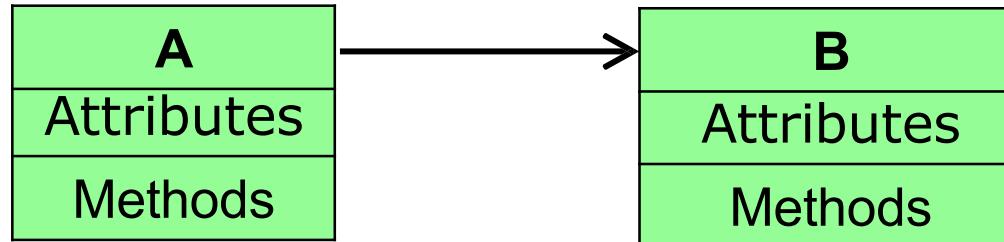
```
public class PetShop {  
  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer();  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog(); //local var  
        _groomer.groom(django);  
    }  
}
```

# Association Relationship

- Association is a relationship between two objects
- Class A and class B are **associated** if A “knows about” B, but B is not a component of A
- But this is **not symmetrical!**
- **Class A holds a class level reference to class B**
- Lifetime?
  - Objects of class A and B have their own lifetime, i.e., they can exist without each other

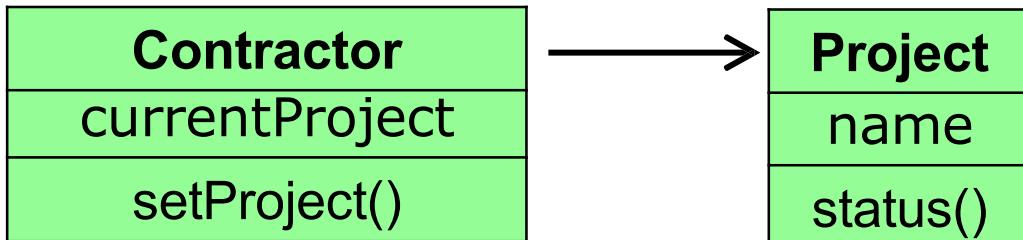
# Association in UML

- Represented by a solid arrow
- In below UML diagram, A holds a reference of B



# Association Example (1/4)

- A contractor's project keeps changing as per company's policy and contractor's performance



```
class Project { private String name;
    public boolean status() { ... }
    ....
}
// Contractor's project keep changing
class Contractor {
    private Project currentProject;
    public Contractor(Project proj) {
        this.currentProject = proj;
    }
    public void setProject(Project proj){
        this.currentProject = proj;
    }
}
```

# Associations Example (2/4)

- **Association** means that one object knows about another object that is not one of its components.

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- As noted, **PetShop** contains a **DogGroomer**, so it can send messages to the **DogGroomer**
- But what if the **DogGroomer** needs to send messages to the **PetShop** she works in?
  - The **DogGroomer** probably needs to know several things about her **PetShop**: for example, operating hours, grooming supplies in stock, customers currently in the shop...

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- The **PetShop** keeps track of such information in its properties.
- Can set up an **association** so that **DogGroomer** can send her **PetShop** messages to retrieve information she needs.

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- This is what the full association looks like
- Let's break it down line by line
- **But note we're not yet making use of the association in this fragment**

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- We declare an instance variable named `_petShop`
- We want this variable to record the instance of `PetShop` that the `DogGroomer` belongs to

```
public class DogGroomer {  
  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- Modified `DogGroomer`'s constructor to take in a parameter of type `PetShop`
- Constructor will refer to it by the name `myPetShop`
- Whenever we instantiate a `DogGroomer`, we'll need to pass it an instance of `PetShop` as an argument. Which? The `PetShop` instance that created the `DogGroomer`, hence use `this`

```
public class DogGroomer { private PetShop  
    _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
    //groom method elided  
}  
  
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    //testGroomer() elided  
}
```

# Associations Example (2/4)

- Now store `myPetShop` in instance variable `_petShop`
- `_petShop` now points to same `PetShop` instance passed to its constructor
- After constructor has been executed and can no longer reference `myPetShop`, any `DogGroomer` method can still access same `PetShop` instance by the name `_petShop`

```
public class DogGroomer {  
  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

# Associations Example (2/4)

- Let's say we've written an accessor method and a mutator method in the `PetShop` class:  
`getClosingTime()` and  
`setNumCustomers(int customers)`

- If the `DogGroomer` ever needs to know the closing time, or needs to update the number of customers, she can do so by calling
  - `getClosingTime()`
  - `setNumCustomers(int customers)`

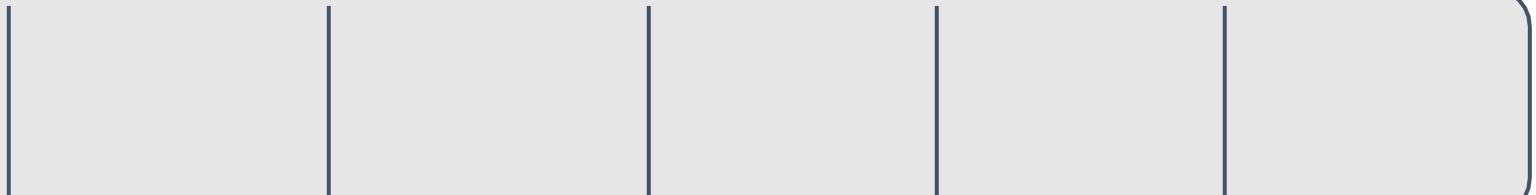
```
public class DogGroomer {  
  
    private PetShop _petShop;  
    private Time _closingTime;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store assoc.  
        _closingTime = myPetShop.getClosingTime();  
        _petShop.setNumCustomers(10);  
    }  
}
```

# Association: Under the Hood (1/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
    example */  
}
```

Somewhere in memory...



# Association: Under the Hood (2/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
    example */  
}
```

Somewhere in memory...



Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop's constructor initializes all its instance variables (just a DogGroomer here)

# Association: Under the Hood (3/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
    example */  
}
```

Somewhere in memory...



The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor  
(remember the this keyword?)

# Association: Under the Hood (4/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;
```

```
public DogGroomer(PetShop myPetShop) {  
    _petShop = myPetShop;  
}  
  
/* groom and other methods elided for this  
example */  
}
```

Somewhere in memory...



When the DogGroomer's constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument.

# Association: Under the Hood (5/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
       example */  
}
```

Somewhere in memory...



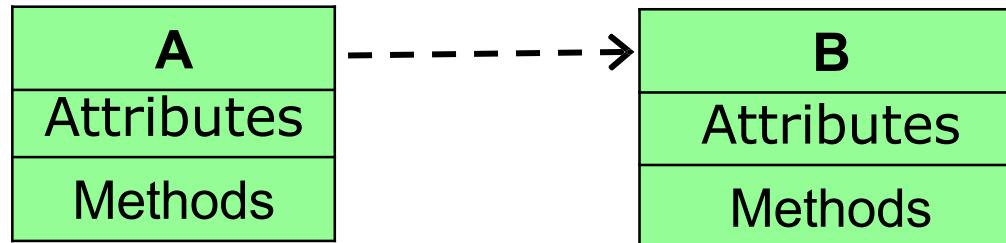
The DogGroomer sets its `_petShop` instance variable to point to the same `PetShop` it received as an argument. Now it “knows about” the `petShop` that instantiated it! And therefore so do all its methods...

# Dependency

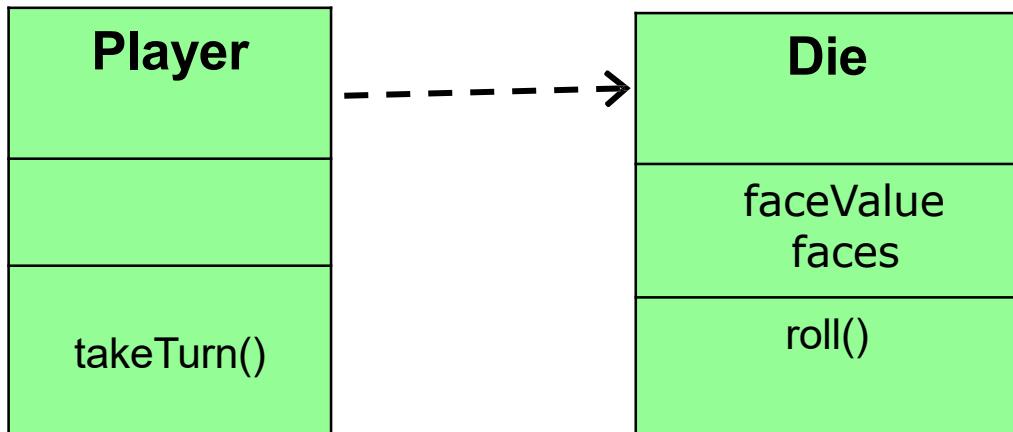
- Class A **depends** on class B if A cannot carry out its work without B, but B is neither a component of A nor it has association with A
- **A is requesting service from an object of class B**
  - A or B “**doesn’t know**” about each other (no association)
  - A or B “**doesn’t contain**” each other (no composition)
- But this is **not symmetrical!** B doesn’t depends on A

# Dependency in UML

- Represented by a dashed arrow starting from the dependent class to its dependency
  - A is dependent on B
  - A is requesting service from B

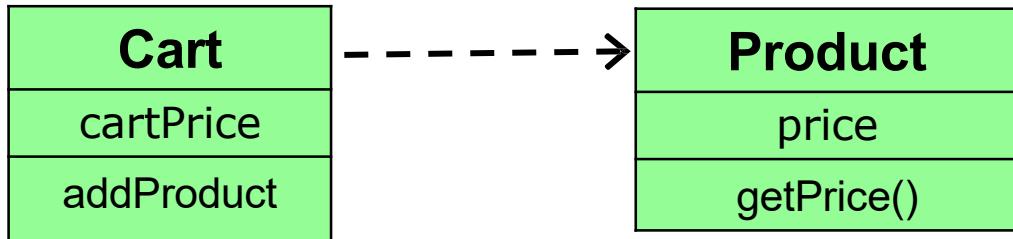


# Dependency Example (1/3)



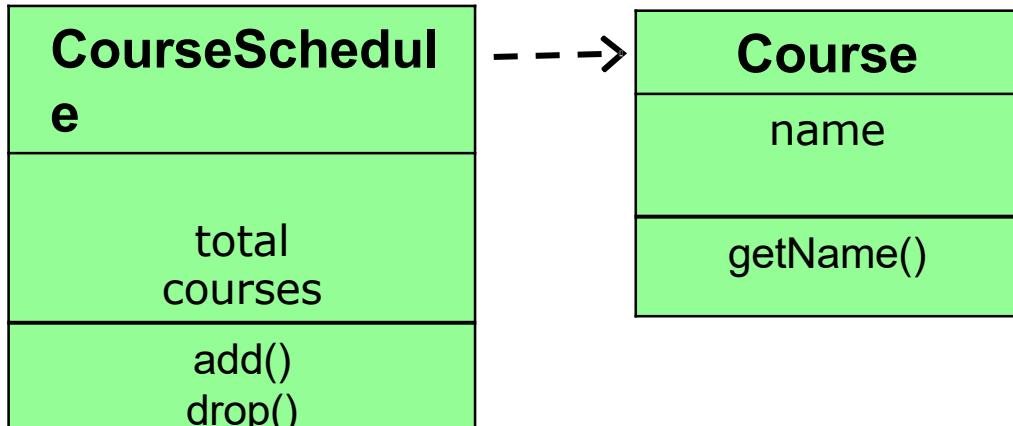
```
class Die {  
    private int faceValue, faces;  
    ....  
    public void roll() { ..... }  
}  
  
class Player {  
    public void takeTurn(Die die) {  
        die.roll();  
    }  
}
```

# Dependency Example (2/3)



```
class Product {  
    private double price;  
    ....  
    public double getPrice() { .... }  
}  
  
class Cart {  
    private double cartPrice;  
    public void addProduct(Product p) {  
        cartPrice += p.getPrice();  
    }  
}
```

# Dependency Example (3/3)



```
class Course {  
    private String name;  
    ....  
    public String getName() { ..... }  
}  
  
class CourseSchedule { private int total; private String courses[];  
public void addCourse(Course c) {  
    courses[total++] = c.getName();  
}  
....  
}
```

# *Advanced Programming*

CSE 201

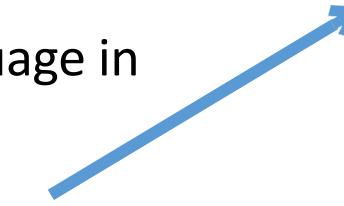
**Instructor: Sambuddho**

(Semester: Monsoon 2024)

Week 7 -- UML Diagrams

# What is UML?

- UML stands for Unified Modeling Language
- It's a widely used modeling language in the field of software engineering
- It's used to analyze, design, and implement software-based systems
- Pretty pictures (diagrams)



## LECTURE 02

- Analysis
  - **What to do and not how to do it**
  - Decide corner cases and exact functionalities
- Design
  - Define classes, their attributes and methods, objects, and class relationships
- Implementation
  - Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Testing
  - A program should be free of errors

4

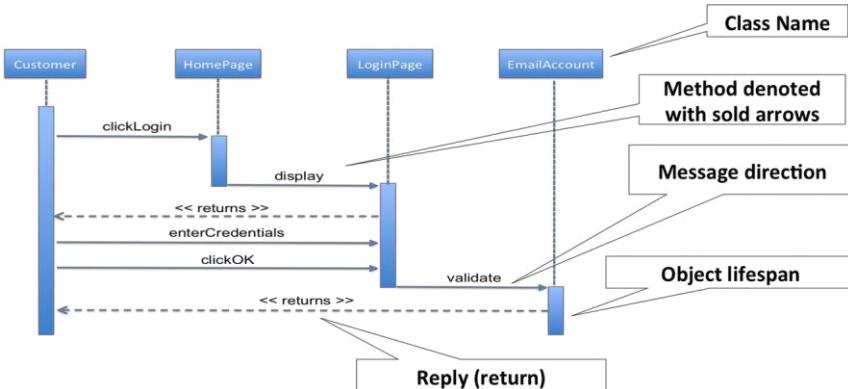
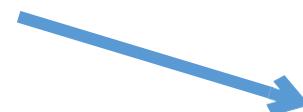
# Motivations for UML

- We need a modeling language to:
  - help develop efficient, effective and correct designs, particularly Object Oriented designs
  - communicate clearly with project stakeholders (concerned parties: developers, customer, etc)
  - give us the “big picture” view of the project

# UML Diagrams

Three types of UML diagrams that we will cover:

1. **Class diagrams:** Represents static structure
2. **Use case diagrams:** Sequence of actions a system performs to yield an observable result to an actor
3. **Sequence diagrams:** Shows how groups of objects interact in some behavior
  - Already covered in Lecture 02



# UML Diagrams: Class Diagrams

- Better name: “Static structure diagram”
  - Doesn’t describe temporal aspects
  - Doesn’t describe individual objects: Only the overall structure of the system
- There are “object diagrams” where the boxes represent instances
  - Rarely used and not covered in this course

# UML Class Notation

- A class is a rectangle divided into three parts

- Class name
- Class attributes (i.e. data members, variables)
- Class operations (i.e. methods)

- Modifiers

- Private: - Public:
- + Protected: #
- Static: Underlined
- 
- 

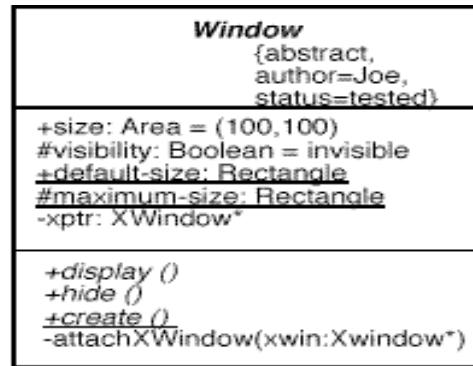
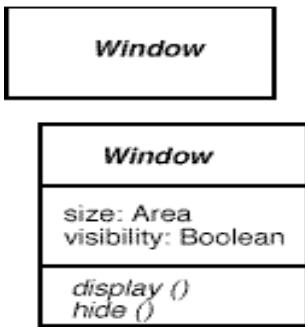
- Abstract class/methods

**Employee**

-Name: String  
+ID: long #Salary: double  
+getName: String  
+setName()  
-calcInternalStuff(in x : byte, in y : decimal)

- Name in italics

# Different Levels of Specifying Classes



Use this for your project

# Class Relationships

- UML diagrams for these class relationships are already covered before
  - Association
  - Composition
  - Dependency
  - Inheritance
- We will only cover binary association relationship here

# Class Relationship: Binary Association

Both entities “Knows About” each other (two-way association)



# UML Multiplicities

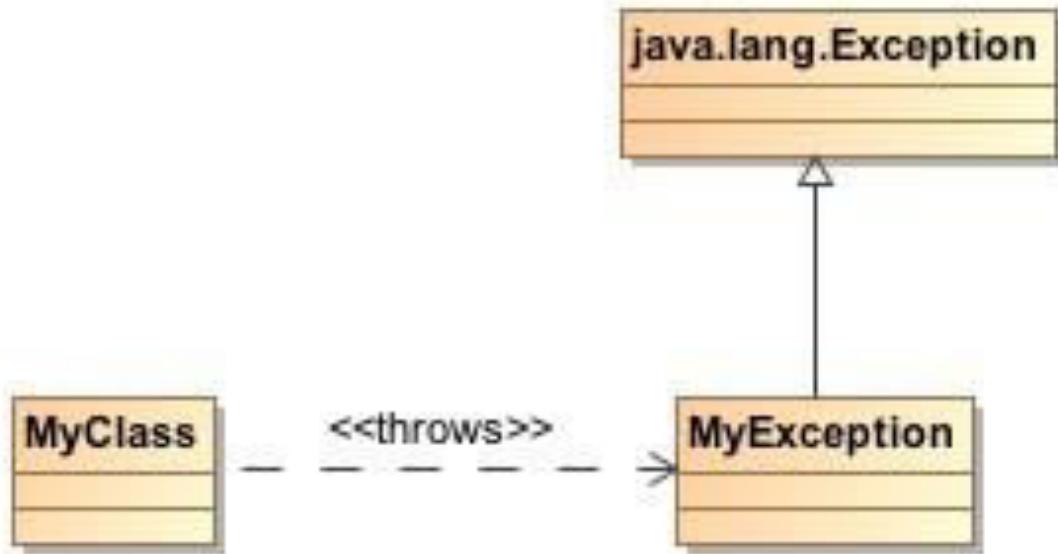
Links on associations to specify more details about the relationship

| Multiplicities | Meaning                                                                               |
|----------------|---------------------------------------------------------------------------------------|
| 0..1           | zero or one instance.<br>The notation “ $n \dots M$ ” indicates $n$ to $m$ instances. |
| 0..* or *      | no limit on the number of instances (including none).                                 |
| 1              | exactly one instance                                                                  |
| 1..*           | at least one instance                                                                 |

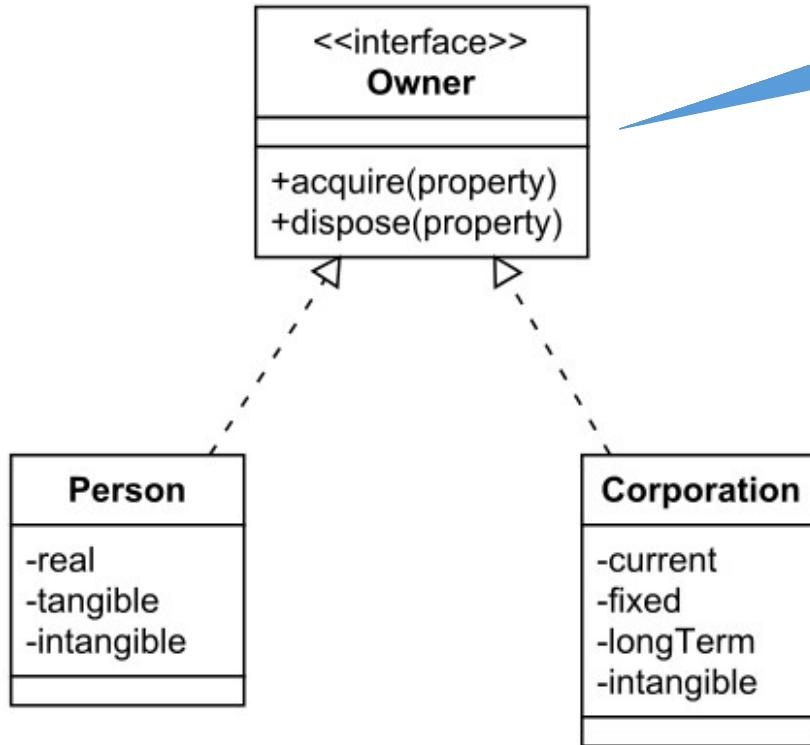


How you will implement?

# Exceptions



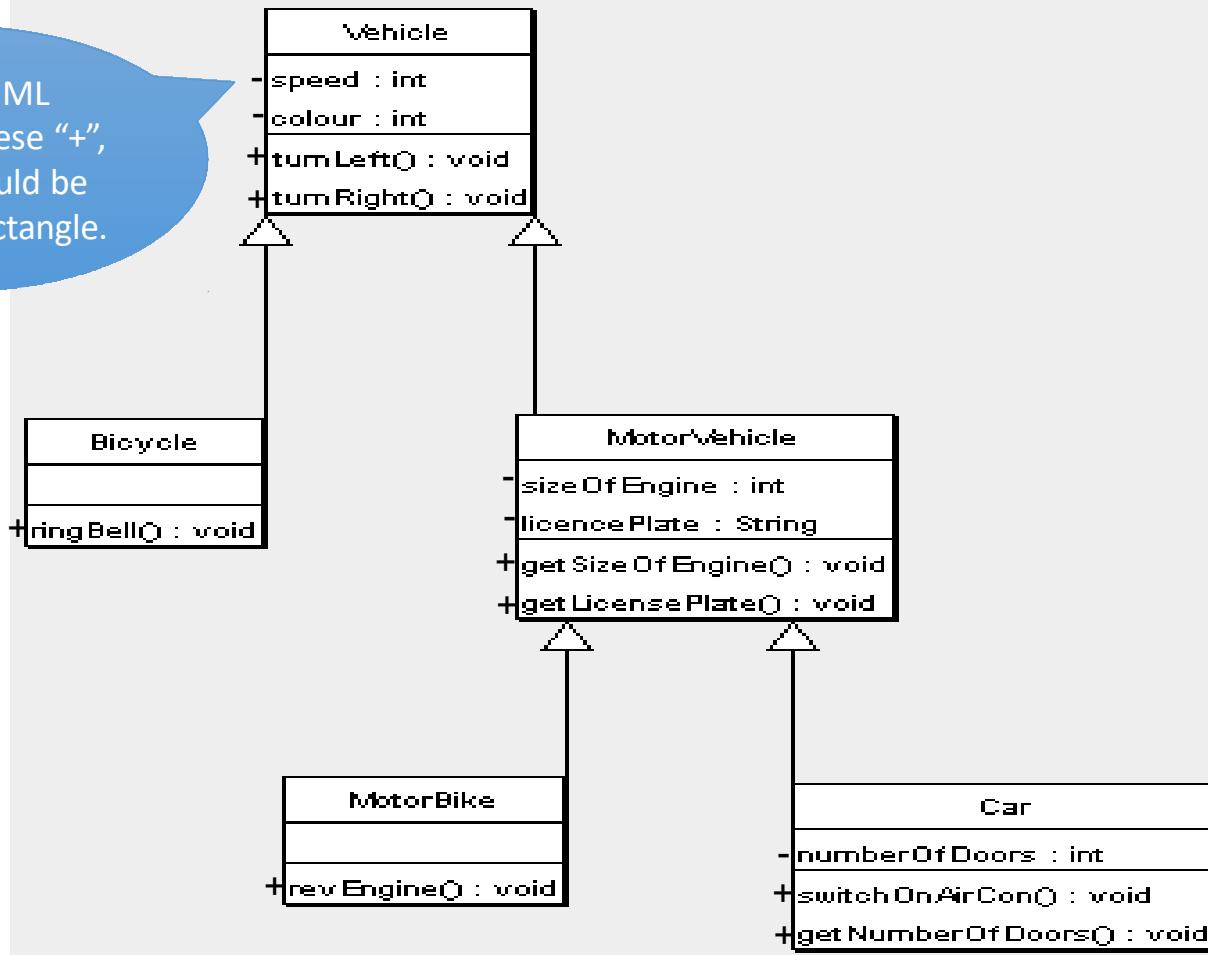
# Interfaces



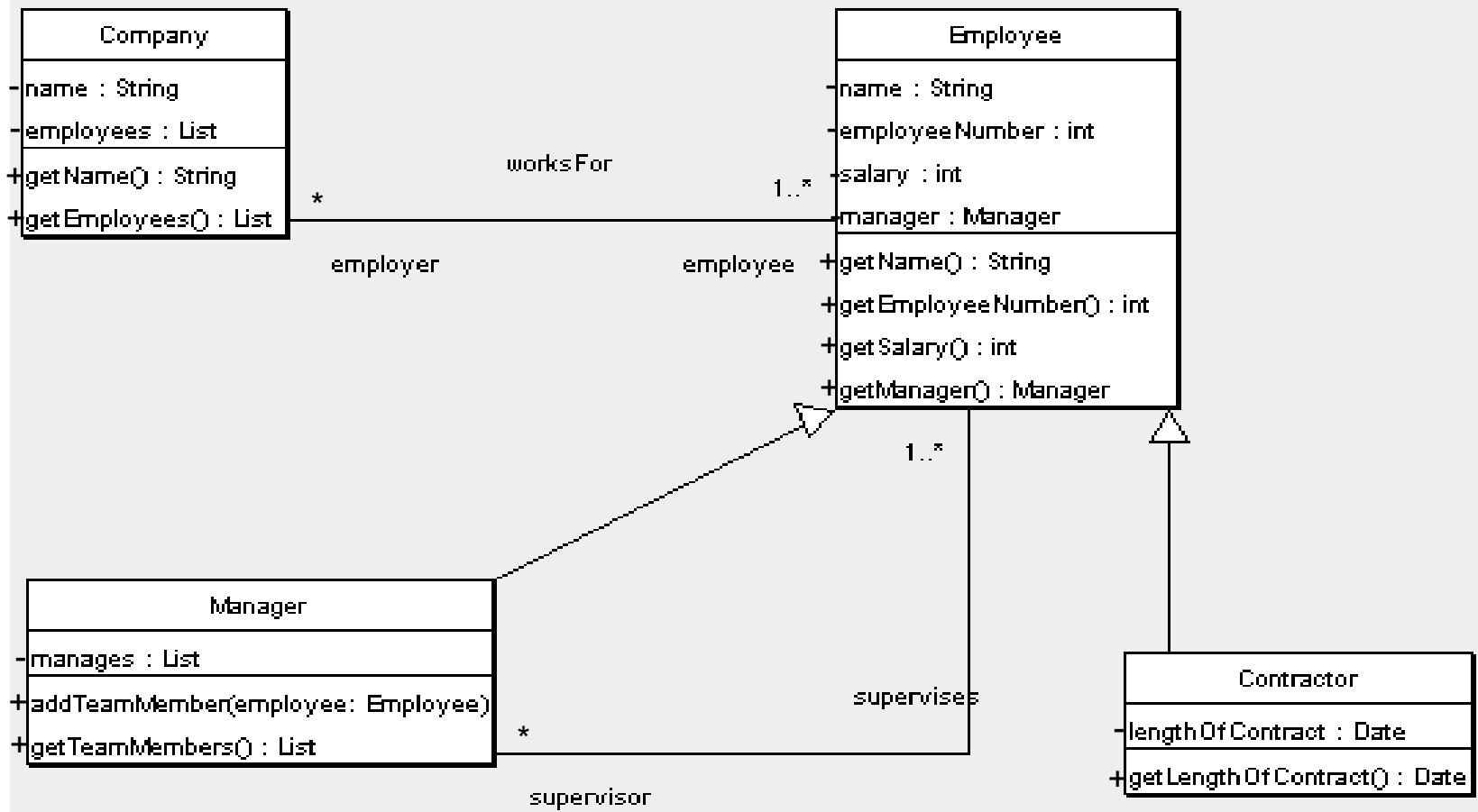
How is this diagram different from that of a class ?

# Sample Class Diagram (1/2)

In your UML diagrams, these "+", "-", etc, should be inside the rectangle.



# Sample Class Diagram (2/2)



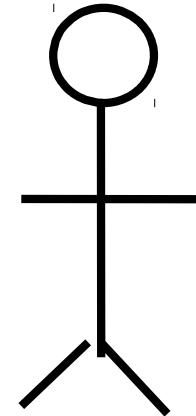
# UML Diagrams: Use Cases

- Means of capturing requirements
  - Used at a very early phase of software development for requirement gathering (analysis phase)
  - Provides a high level overview of the system
  - Class diagrams are created after generating use case diagrams
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system
- A scenario based technique in UML
- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a

system does rather than *how*

# Actors in Use Case

- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
  - It can be a:
    - Human
    - Peripheral device (hardware)
    - External system or subsystem
    - Time or time-based event
  - Labelled using a descriptive noun or phrase
  - Represented by stick figure

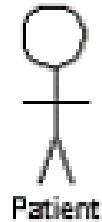


# Use Case Analysis (1/4)

- Sample scenario
  - *A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot*
- We want to write a use case for this scenario

# Use Case Analysis (2/4)

- Sample scenario
  - “A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot”
- Who is the actor?
  - The actor is a “Patient” here



# Use Case Analysis (3/4)

- Sample scenario
  - “A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot”
- A **use case** is a summary of scenarios for a single task or goal
  - So, what is the use case here?  
The use case is “Make Appointment”

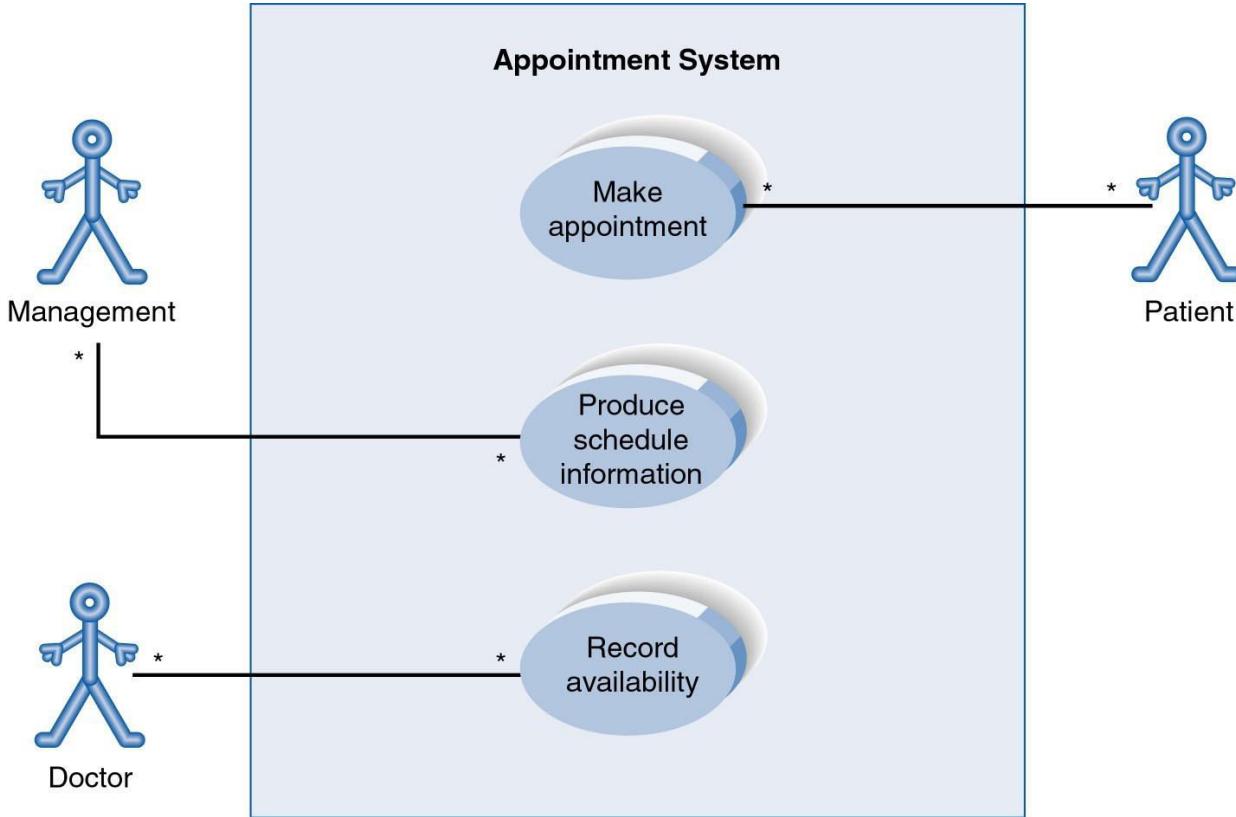
# Use Case Analysis (4/4)

- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication**
- Actors are stick figures
- Use cases are ovals
- Labelled using a descriptive verb-noun phrase
- Communications are lines that link actors to use cases
- Boundary rectangle is placed around the perimeter of the system to show how the actors communicate with the system



# Use Case Diagram

- A use case diagram is a collection of actors, use cases, and their communications

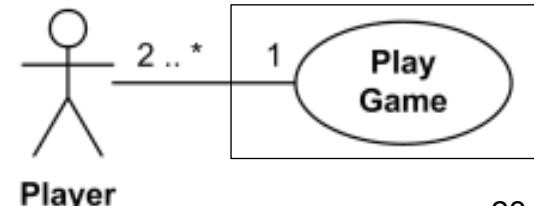
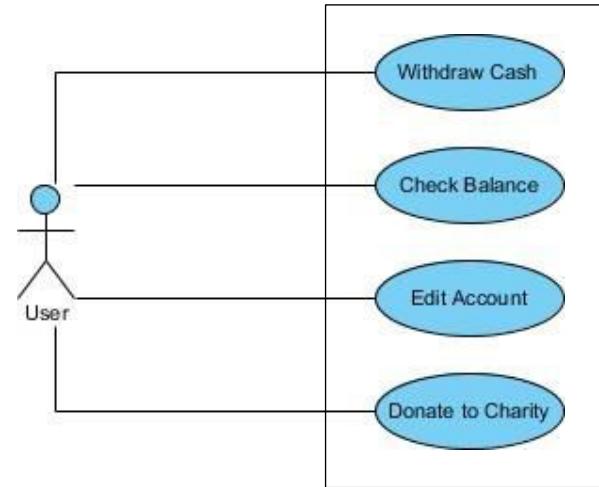


# Relationships for Use Cases

- Association
- Generalization
- Extend
- Include

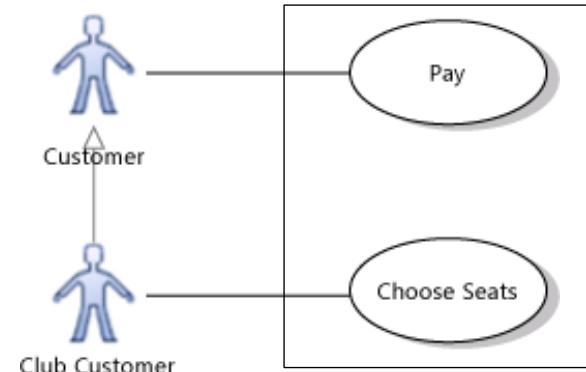
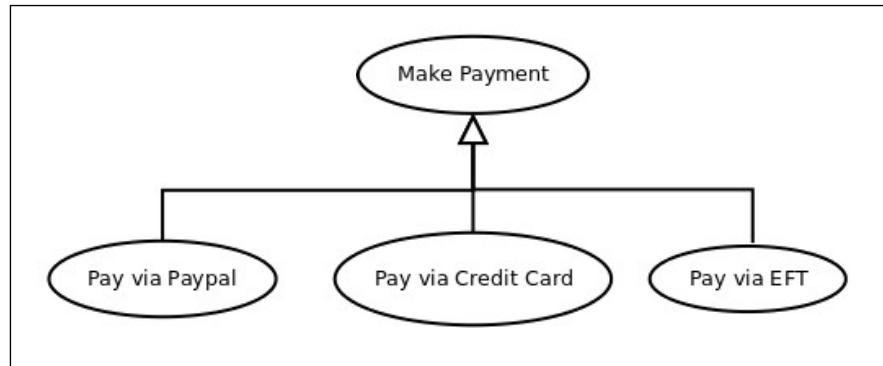
# Association Relationship

- Exists only between an actor and a use case
  - Indicates that an actor can use certain functionality of the system
- Represented by a solid line without arrowhead
  - Most commonly used representation
  - Uncommon to show one-way association
- The association between an actor and a use case can also show multiplicity at each end



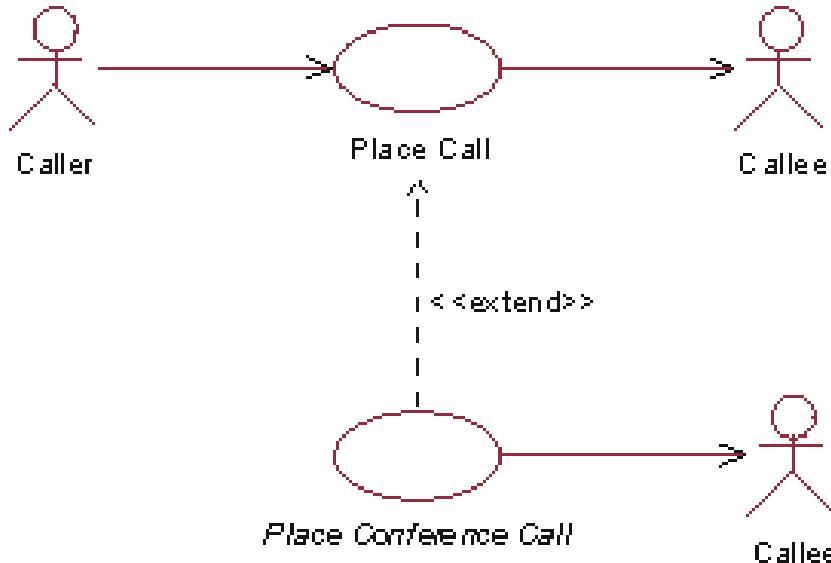
# Generalization Relationship

- Could exist between two actors or between two use cases
  - Indicates parent/child relationship
- Represented by a solid line with a triangular and hollow arrowhead
  - From child to parent



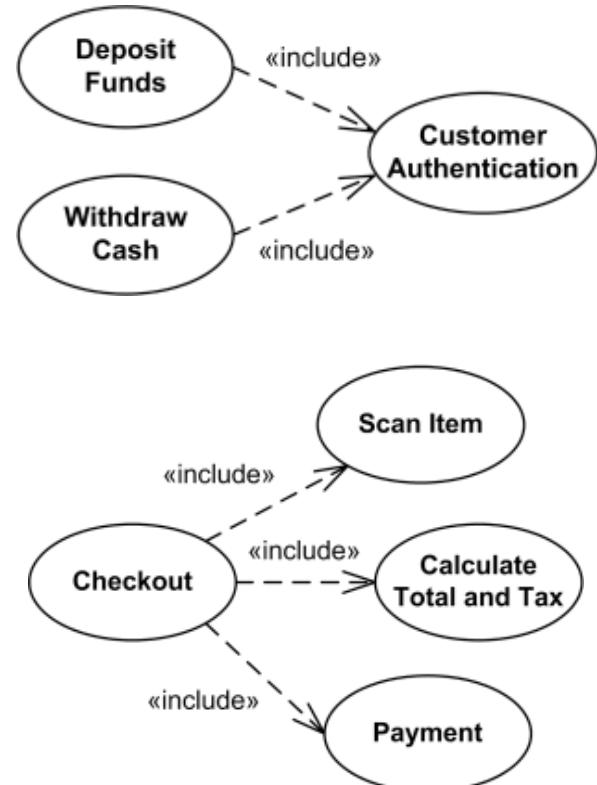
# Extend Relationship “<<extend>>”

- Exists only between use cases
  - This relationships represent optional or seldom invoked cases
  - Indicates that although one use case is a variation of another but it is invoked rarely
    - Lot of shared code between these use cases (**not to be confused with inheritance**)
- Represented using a dashed arrow with an arrowhead. The notation “<< extend >>” is also mentioned above the arrow
  - The direction of the arrow is toward the extended use cases

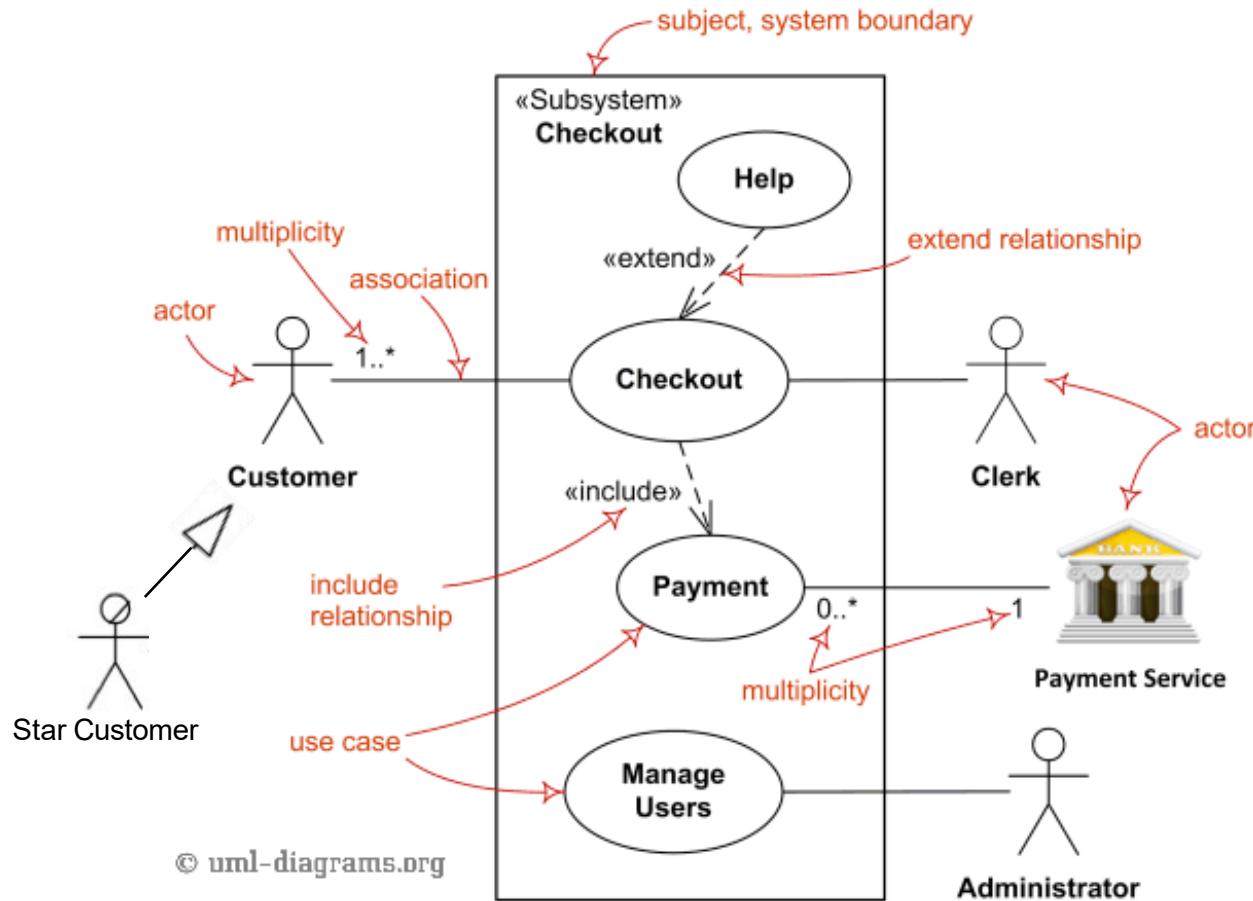


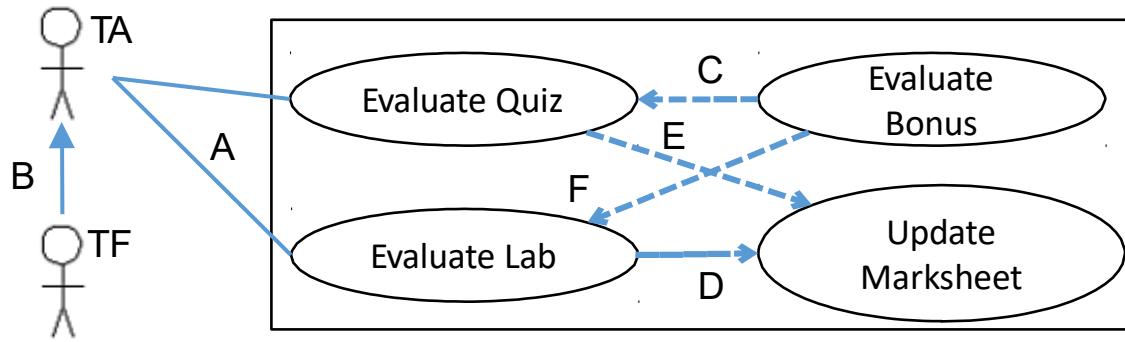
# Include Relationship “<<include>>”

- Exists only between use cases
  - Represents behavior that is factored out of the use case
  - Doesn't mean that the factored out use case is an optional or seldom invoked cases
- Represented using a dashed arrow with an arrowhead. The notation “<<include>>” is also mentioned above the arrow
  - The direction of the arrow is toward the included use case



# Sample Use Case



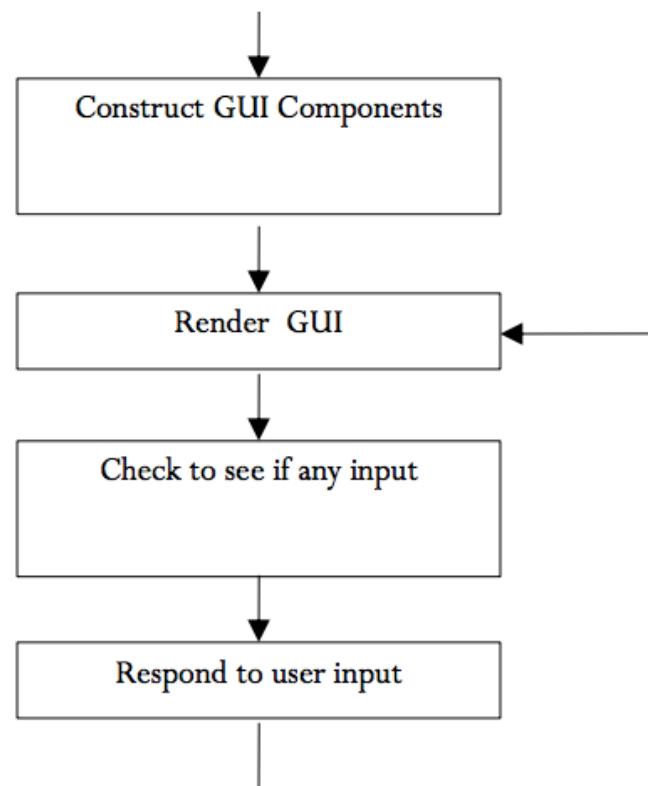


*Advanced Programming*  
**CSE 201**  
**Instructor: Sambuddho**

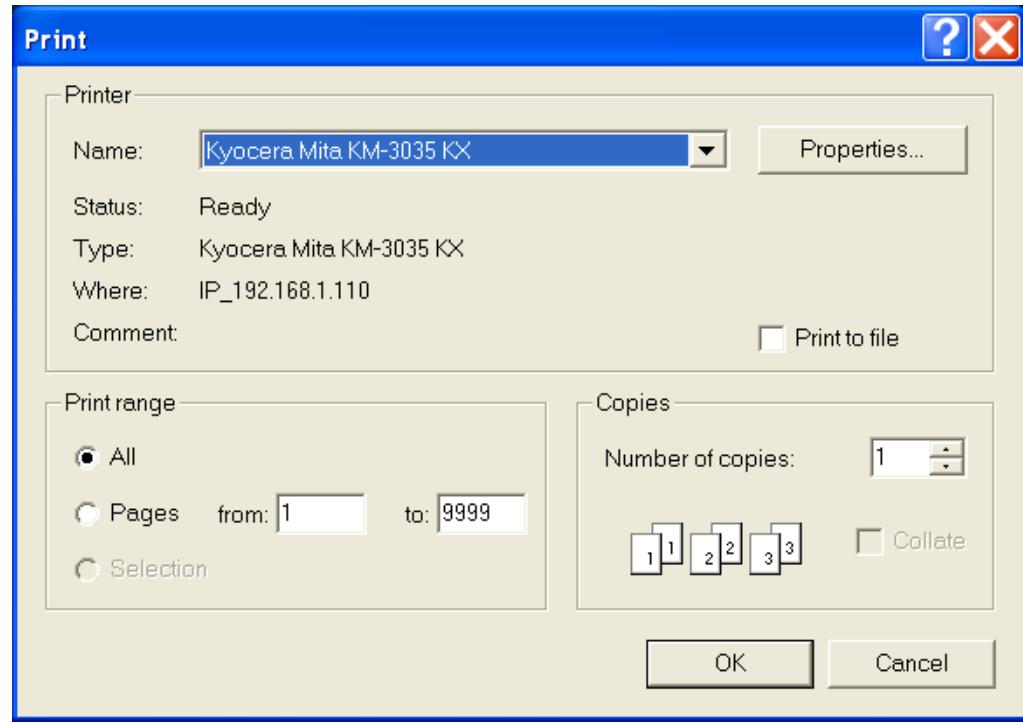
(Semester: Monsoon 2024)  
Week 8 – GUI/AWT

# How do GUI Works?

- They loop and respond to events

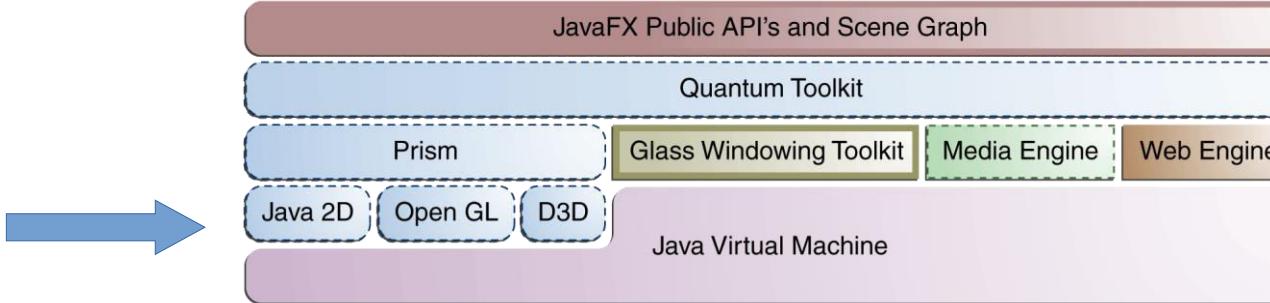


# GUI Examples



# Java Runtime High Level Architecture for GUIs

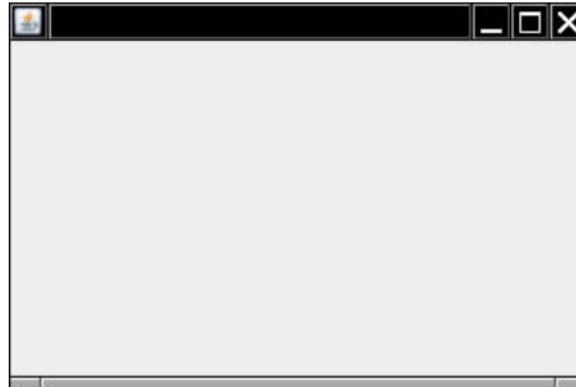
- You are here



# Creating a Frame

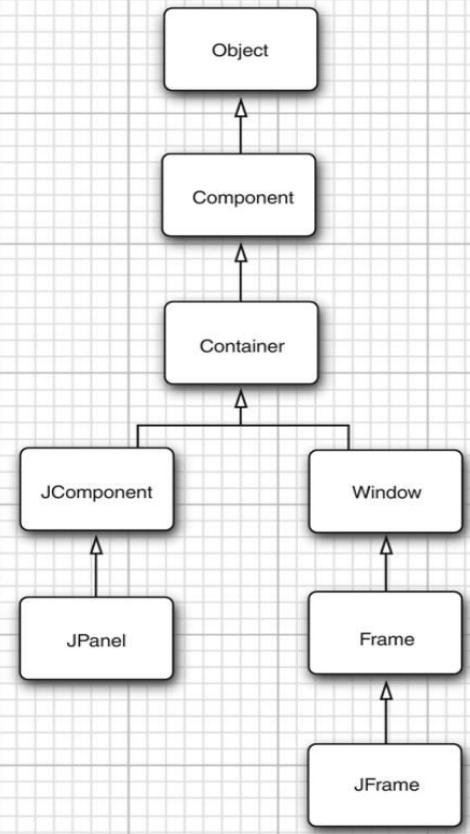
## Swing JFrame

```
1 package simpleFrame;  
2  
3 import java.awt.*;  
4 import javax.swing.*;  
5  
6 /**  
7  * @version 1.34 2018-04-10  
8  * @author Cay Horstmann  
9  */  
10 public class SimpleFrameTest  
11 {  
12     public static void main(String[] args)  
13     {  
14         EventQueue.invokeLater(() ->  
15         {  
16             var frame = new SimpleFrame();  
17             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
18             frame.setVisible(true);  
19         });  
20     }  
21 }  
22 }
```



```
23 class SimpleFrame extends JFrame  
24 {  
25     private static final int DEFAULT_WIDTH = 300;  
26     private static final int DEFAULT_HEIGHT = 200;  
27  
28     public SimpleFrame()  
29     {  
30         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
31     }  
32 }
```

# Frame Properties



## java.awt.Component 1.0

- `boolean isVisible()`
- `void setVisible(boolean b)`

gets or sets the visible property. Components are initially visible, with the exception of top-level components such as `JFrame`.

- `void setSize(int width, int height) 1.1`  
resizes the component to the specified width and height.
- `void setLocation(int x, int y) 1.1`

moves the component to a new location. The `x` and `y` coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a `JFrame`).

- `void setBounds(int x, int y, int width, int height) 1.1`  
moves and resizes this component.
- `Dimension getSize() 1.1`
- `void setSize(Dimension d) 1.1`  
gets or sets the size property of this component.

## java.awt.Window 1.0

- `void setLocationByPlatform(boolean b) 5`

gets or sets the `locationByPlatform` property. When the property is set before this window is displayed, the platform picks a suitable location.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`

gets or sets the `resizable` property. When the property is set, the user can resize the frame.

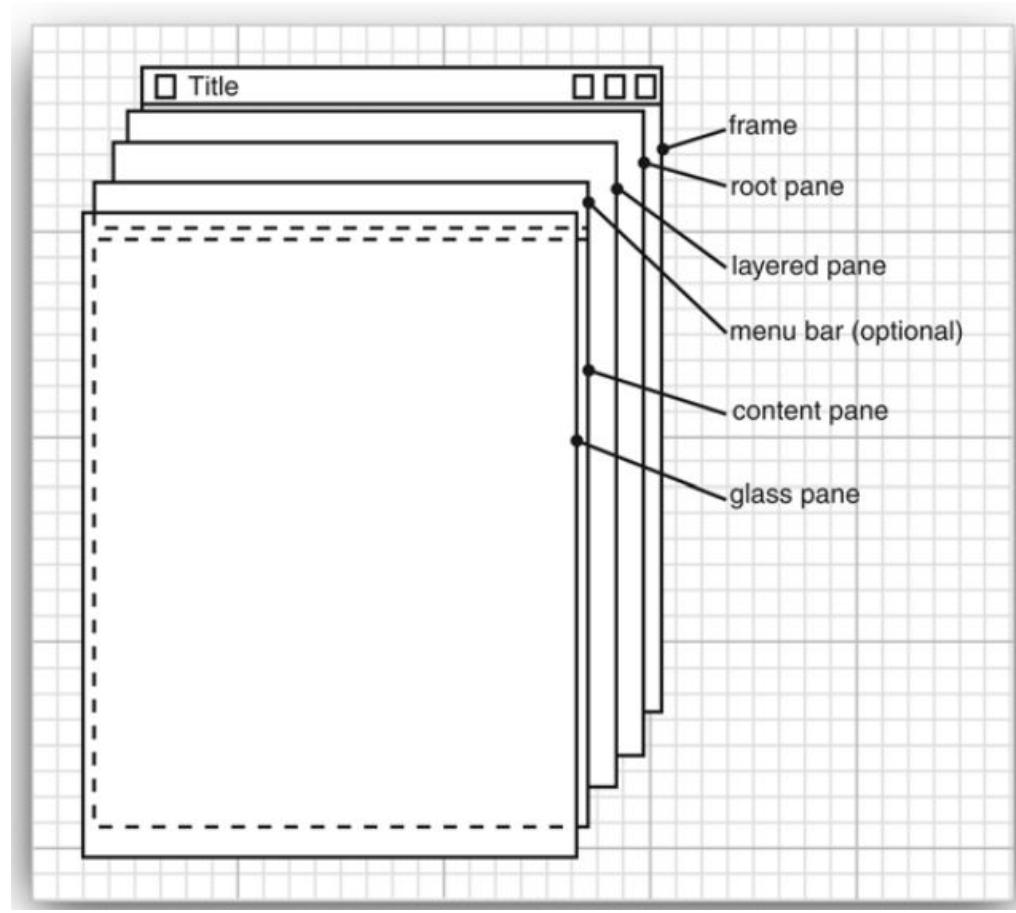
- `String getTitle()`
- `void setTitle(String s)`

gets or sets the `title` property that determines the text in the title bar for the frame.

# Components

*Content pane is the way to go -->*

```
Component c = . . .;  
frame.add(c); // added to the content pane
```



# Components

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        code for drawing
    }
}
```

```
public class NotHelloWorldComponent extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    public void paintComponent(Graphics g)
    {
        g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
    }
    ...
}
```

```
public class NotHelloWorldComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    ...
    public Dimension getPreferredSize()
    {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```



```
class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
        add(new NotHelloWorldComponent());
        pack();
    }
}
```

# A Basic GUI Program Using AWT

```
1 package notHelloWorld;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 /**
7  * @version 1.34 2018-04-10
8  * @author Cay Horstmann
9  */
10 public class NotHelloWorld
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(() ->
15         {
16             var frame = new NotHelloWorldFrame();
17             frame.setTitle("NotHelloWorld");
18             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20         });
21     }
22 }
23
24 /**
25  * A frame that contains a message panel.
26  */
27 class NotHelloWorldFrame extends JFrame
28 {
29     public NotHelloWorldFrame()
30     {
31         add(new NotHelloWorldComponent());
32         pack();
33     }
34 }
35
```

```
36 /**
37  * A component that displays a message.
38  */
39 class NotHelloWorldComponent extends JComponent
40 {
41     public static final int MESSAGE_X = 75;
42     public static final int MESSAGE_Y = 100;
43
44     private static final int DEFAULT_WIDTH = 300;
45     private static final int DEFAULT_HEIGHT = 200;
46
47     public void paintComponent(Graphics g)
48     {
49         g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
50     }
51
52     public Dimension getPreferredSize()
53     {
54         return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
55     }
56 }
```

# 2D Objects

Line2D, Rectangle2D, Ellipse2D etc. ← all implementations of Shape interface.

To draw a shape:

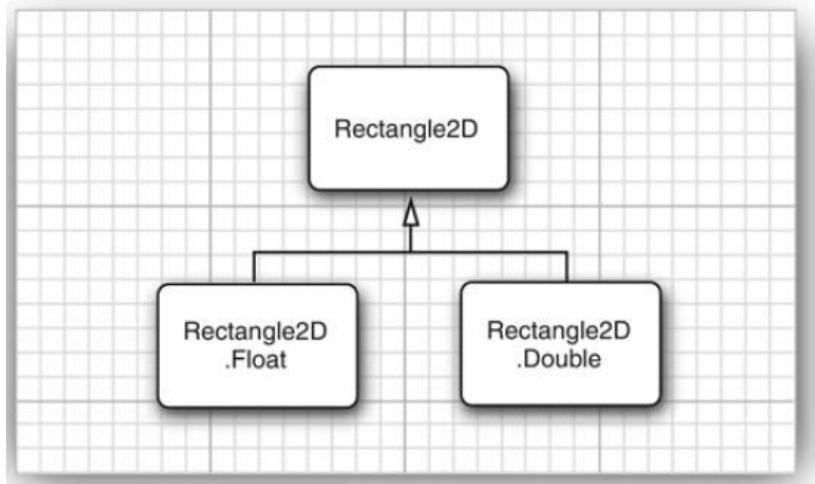
1. Create an object of the class that implements the Shape interface.
2. Then pass on the object to Graphics2D.draw().

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    Rectangle2D rect = . . . ;
    g2.draw(rect);

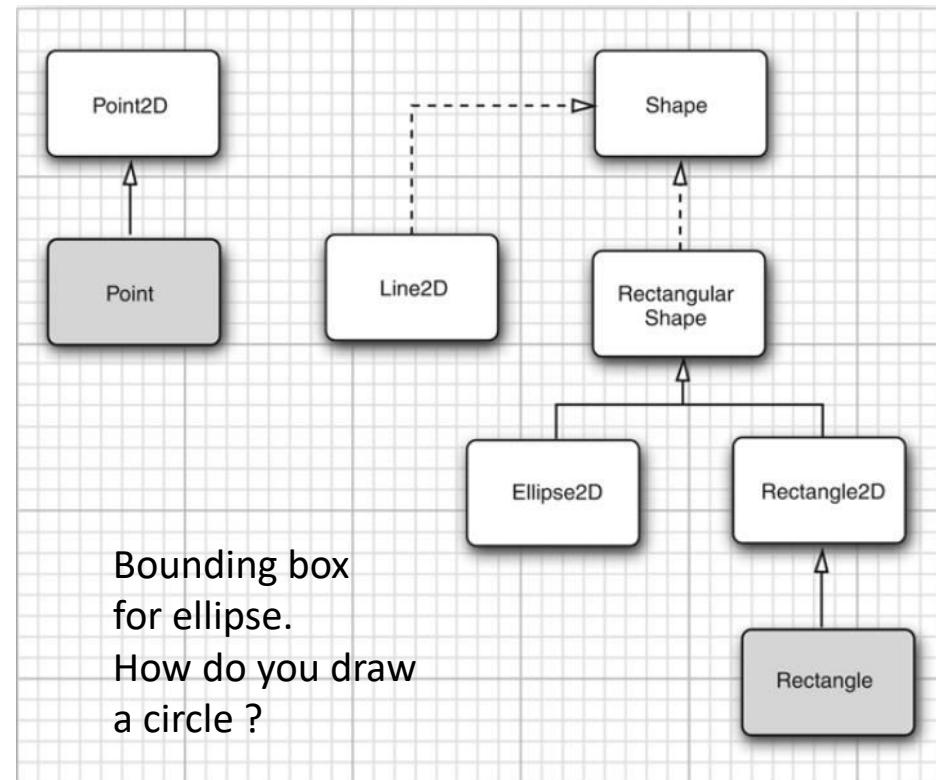
}
```

# 2D Objects

Shapes are of two types – based on floating point or double pixels.



```
var floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
var doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```



# 2D Objects

```
1 package draw;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import javax.swing.*;
6
7 /**
8 * @version 1.34 2018-04-10
9 * @author Cay Horstmann
10 */
11 public class DrawTest
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(() ->
16         {
17             var frame = new DrawFrame();
18             frame.setTitle("DrawTest");
19             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20             frame.setVisible(true);
21         });
22     }
23 }
24
25 /**
26 * A frame that contains a panel with drawings.
27 */
28 class DrawFrame extends JFrame
29 {
30     public DrawFrame()
31     {
32         add(new DrawComponent());
33         pack();
34     }
35 }
36
37 /**
38 * A component that displays rectangles and ellipses.
39 */
40 class DrawComponent extends JComponent
41 {
42     private static final int DEFAULT_WIDTH = 400;
43     private static final int DEFAULT_HEIGHT = 400;
44
45     public void paintComponent(Graphics g)
46     {
47         var g2 = (Graphics2D) g;
48
49         // draw a rectangle
50
51         double leftX = 100;
52         double topY = 100;
53         double width = 200;
54         double height = 150;
55
56         var rect = new Rectangle2D.Double(leftX, topY, width, height);
57         g2.draw(rect);
58
59         // draw the enclosed ellipse
60
61         var ellipse = new Ellipse2D.Double();
62         ellipse setFrame(rect);
63         g2.draw(ellipse);
64
65         // draw a diagonal line
66
67         g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
68
69         // draw a circle with the same center
70
71         double centerX = rect.getCenterX();
72         double centerY = rect.getCenterY();
73         double radius = 150;
74
75         var circle = new Ellipse2D.Double();
76         circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
77         g2.draw(circle);
78     }
79 }
```

# 2D Objects - Colors

```
Rectangle2D rect = . . .;  
g2.setPaint(Color.RED);  
g2.fill(rect); // fills rect with red
```

java.awt.Color – BLACK, BLUE, CYAN,  
DARK\_GRAY, GREEN, GRAY,  
LIGHT\_GRAY, MAGENTA, ORANGE,  
PINK, RED, WHITE, YELLOW

```
var component = new MyComponent();  
component.setBackground(Color.PINK);
```

## java.awt.Color 1.0

- Color(int r, int g, int b)

creates a color object with the given red, green, and blue components between 0 and 255.

## java.awt.Graphics2D 1.2

- Paint getPaint()
- void setPaint(Paint p)

gets or sets the paint property of this graphics context. The Color class implements the Paint interface. Therefore, you can use this method to set the paint attribute to a solid color.

- void fill(Shape s)

fills the shape with the current paint.

## java.awt.Component 1.0

- Color getForeground()
- Color getBackground()
- void setForeground(Color c)
- void setBackground(Color c)

gets or sets the foreground or background color.

# 2D Objects - Fonts

Font face names available – SansSerif, Serif, Monospaced, Dialog, DialogInput

```
import java.awt.*;  
  
public class ListFonts  
{  
    public static void main(String[] args)  
    {  
        String[] fontNames = GraphicsEnvironment  
            .getLocalGraphicsEnvironment()  
            .getAvailableFontFamilyNames();  
        for (String fontName : fontNames)  
            System.out.println(fontName);  
    }  
}
```

Set font for a string



```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);  
g2.setFont(sansbold14);  
var message = "Hello, World!";  
g2.drawString(message, 75, 100);
```

# 2D Objects - Images

```
Image image = new ImageIcon(filename).getImage();  
  
public void paintComponent(Graphics g)  
{  
    . . .  
    g.drawImage(image, x, y, null);  
}
```

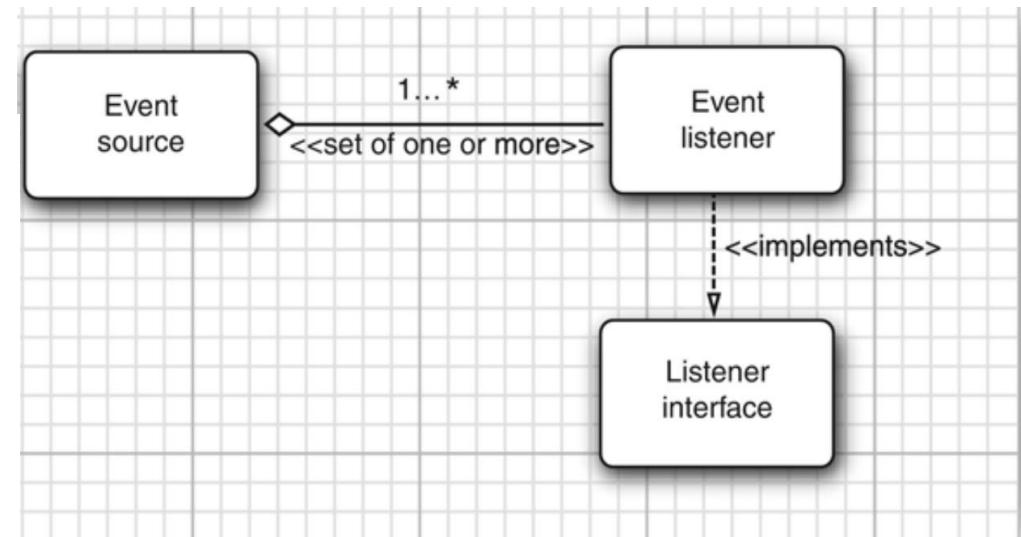
## java.awt.Graphics 1.0

- boolean drawImage(Image img, int x, int y, ImageObserver observer)
- boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)  
draws an unscaled or scaled image. Note: This call may return before the image is drawn. The `imageObserver` object is notified of the rendering progress. This was a useful feature in the distant past. Nowadays, just pass a `null` observer.
- void copyArea(int x, int y, int width, int height, int dx, int dy)  
copies an area of the screen. The `dx` and `dy` parameters are the distance from the source area to the target area.

# Event Handling

- All events are converted to AWT *event objects* belonging to `java.util.EventObject` (subclasses e.g. `ActionEvent`, `WindowEvent` etc.).

```
class MyListener implements ActionListener
{
    ...
    public void actionPerformed(ActionEvent event)
    {
        // reaction to button click goes here
        ...
    }
}
```



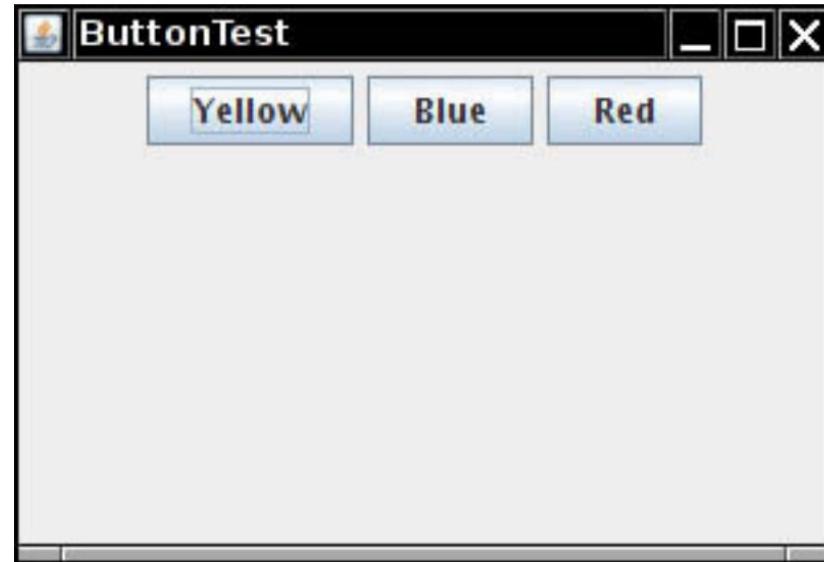
- JButton object creates an ActionEvent and calls `listener.actionPerformed(event)`

# Handling a Button Click Event

```
var yellowButton = new JButton("Yellow");
var blueButton = new JButton("Blue");

var redButton = new JButton("Red");

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```



# Handling a Button Click Event

```
class ColorAction implements ActionListener  
{  
    private Color backgroundColor;  
  
    public ColorAction(Color c)  
    {  
        backgroundColor = c;  
    }  
  
    public void actionPerformed(ActionEvent event)  
    {  
        // set panel background color  
        . . .  
    }  
}
```

```
var yellowAction = new ColorAction(Color.YELLOW);  
var blueAction = new ColorAction(Color.BLUE);  
var redAction = new ColorAction(Color.RED);  
  
yellowButton.addActionListener(yellowAction);  
blueButton.addActionListener(blueAction);  
redButton.addActionListener(redAction);
```

# Handling a Button Click Event

```
1 package button;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8 * A frame with a button panel.
9 */
10 public class ButtonFrame extends JFrame
11 {
12     private JPanel buttonPanel;
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     public ButtonFrame()
17     {
18         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20         // create buttons
21         var yellowButton = new JButton("Yellow");
22         var blueButton = new JButton("Blue");
23         var redButton = new JButton("Red");
24
25         buttonPanel = new JPanel();
26
27         // add buttons to panel
28         buttonPanel.add(yellowButton);
29         buttonPanel.add(blueButton);
30         buttonPanel.add(redButton);
31
32         // add panel to frame
33         add(buttonPanel);
34
35         // add panel to frame
36         add(buttonPanel);
37
38         // create button actions
39         var yellowAction = new ColorAction(Color.YELLOW);
40         var blueAction = new ColorAction(Color.BLUE);
41         var redAction = new ColorAction(Color.RED);
42
43         // associate actions with buttons
44         yellowButton.addActionListener(yellowAction);
45         blueButton.addActionListener(blueAction);
46         redButton.addActionListener(redAction);
47
48     }
49
50     /**
51      * An action listener that sets the panel's background color.
52      */
53     private class ColorAction implements ActionListener
54     {
55         private Color backgroundColor;
56
57         public ColorAction(Color c)
58         {
59             backgroundColor = c;
60         }
61
62         public void actionPerformed(ActionEvent event)
63         {
64             buttonPanel.setBackground(backgroundColor);
65         }
66     }
67 }
```

# Handling a Button Click Event – Single Event Handler for All Types

```
public void makeButton(String name, Color backgroundColor)
{
    var button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(event ->
        buttonPanel.setBackground(backgroundColor));
}
```

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

# Adapter Classes

- Used to monitor window open/close events.
- An appropriate listener must be required to catch such events.

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

Requires implementing all the methods.

# Adapter Classes

- Alternative – extend and use *adapter* classes
- Adapter classes have dummy do-nothing methods for all the rest of the operations.

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (user agrees)
            System.exit(0);
    }
}
```

```
var listener = new Terminator();
frame.addWindowListener(listener);
```

# Mouse Events

- Three main listener methods – mousePressed, mouseClicked and mouseReleased.
- MouseEvent object sent to the event handler class (adapter or listener implementation).

**java.awt.event.MouseEvent 1.1**

- int getX()
- int getY()
- Point getPoint()

returns the *x* (horizontal) and *y* (vertical) coordinates of the point where the event happened, measured from the top left corner of the component that is the event source.

- int getClickCount()

returns the number of consecutive mouse clicks associated with this event. (The time interval for what constitutes “consecutive” is system-dependent.)

```
private class MouseHandler extends MouseAdapter
{
    public void mousePressed(MouseEvent event)
    {
        // add a new square if the cursor isn't inside a square
        current = find(event.getPoint());
        if (current == null) add(event.getPoint());
    }

    public void mouseClicked(MouseEvent event)
    {
        // remove the current square if double clicked
        current = find(event.getPoint());
        if (current != null && event.getClickCount() >= 2) remove(current);
    }
}
```

# Mouse Event

```
package mouse;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * A component with mouse operations for adding and removing squares.
 */
public class MouseComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private static final int SIDELENGTH = 10;
    private ArrayList<Rectangle2D> squares;
    private Rectangle2D current; // the square containing the mouse cursor

    public MouseComponent()
    {
        squares = new ArrayList<>();
        current = null;
    }

    addMouseListener(new MouseHandler());
    addMouseMotionListener(new MouseMotionHandler());
}

public Dimension getPreferredSize()
{
    return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
}

public void paintComponent(Graphics g)
{
    var g2 = (Graphics2D) g;

    // draw all squares
    for (Rectangle2D r : squares)
        g2.draw(r);
}

/**
 * Finds the first square containing a point.
 * @param p a point
 * @return the first square that contains p
 */
public Rectangle2D find(Point2D p)
{
    for (Rectangle2D r : squares)
    {
        if (r.contains(p)) return r;
    }
    return null;
}

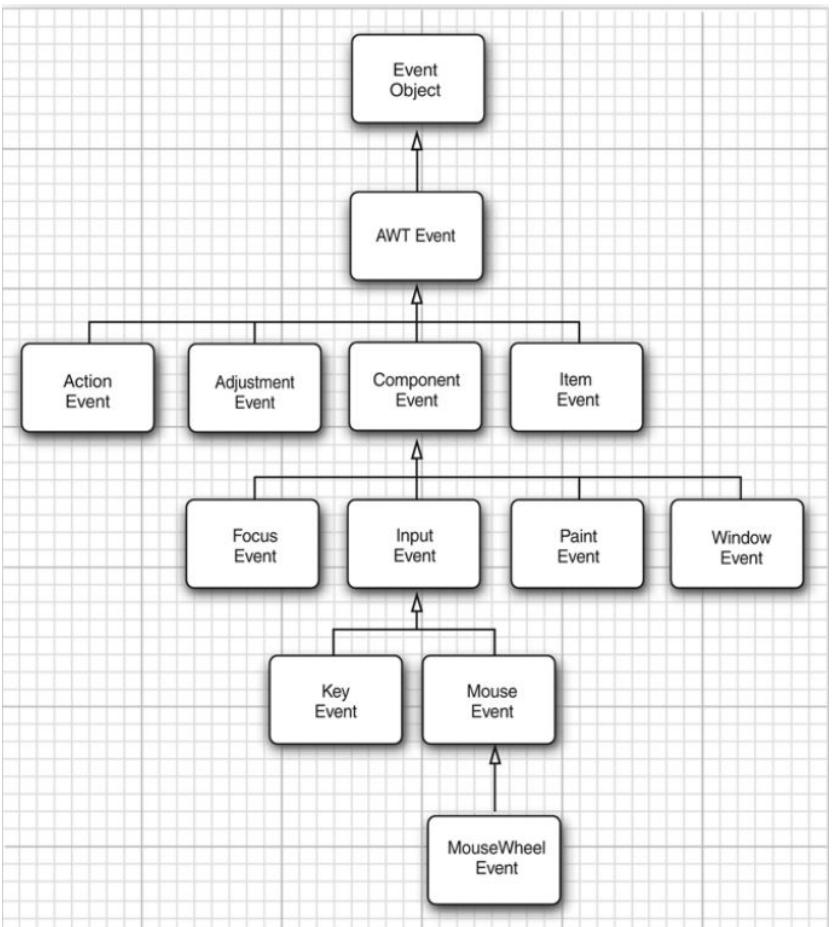
/**
 * Adds a square to the collection.
 * @param p the center of the square
 */
public void add(Point2D p)
{
    double x = p.getX();
    double y = p.getY();

    current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2,
                                    SIDELENGTH, SIDELENGTH);
    squares.add(current);
    repaint();
}
```

# Mouse Event

```
/**  
 * Removes a square from the collection.  
 * @param s the square to remove  
 */  
public void remove(Rectangle2D s)  
{  
    if (s == null) return;  
    if (s == current) current = null;  
    squares.remove(s);  
    repaint();  
}  
  
private class MouseHandler extends MouseAdapter  
{  
    public void mousePressed(MouseEvent event)  
    {  
        // add a new square if the cursor isn't inside a square  
        current = find(event.getPoint());  
        if (current == null) add(event.getPoint());  
    }  
  
    public void mouseClicked(MouseEvent event)  
    {  
        // remove the current square if double clicked  
        current = find(event.getPoint());  
        if (current != null && event.getClickCount() >= 2) remove(current);  
    }  
}  
  
private class MouseMotionHandler implements MouseMotionListener  
{  
    public void mouseMoved(MouseEvent event)  
    {  
        // set the mouse cursor to cross hairs if it is inside a rectangle  
  
        if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());  
        else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));  
    }  
  
    public void mouseDragged(MouseEvent event)  
    {  
        if (current != null)  
        {  
            int x = event.getX();  
            int y = event.getY();  
  
            // drag the current rectangle to center it at (x, y)  
            current setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);  
            repaint();  
        }  
    }  
}
```

# AWT Hierarchy



| Interface           | Methods                                                                      | Parameter/Accessors                                                                                | Events Generated By                                |
|---------------------|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|----------------------------------------------------|
| ActionListener      | actionPerformed                                                              | ActionEvent<br>• getActionCommand<br>• getModifiers                                                | AbstractButton<br>JComboBox<br>JTextField<br>Timer |
| AdjustmentListener  | adjustmentValueChanged                                                       | AdjustmentEvent<br>• getAdjustable<br>• getAdjustmentType                                          | JScrollbar                                         |
| Interface           | Methods                                                                      | Parameter/Accessors                                                                                | Events Generated By                                |
| ItemListener        | itemStateChanged                                                             | ItemEvent<br>• getItem<br>• getItemSelectable<br>• getStateChange                                  | AbstractButton<br>JComboBox                        |
| FocusListener       | focusGained<br>focusLost                                                     | FocusEvent<br>• isTemporary                                                                        | Component                                          |
| KeyListener         | keyPressed<br>keyReleased<br>keyTyped                                        | KeyEvent<br>• getKeyChar<br>• getKeyCode<br>• getKeyModifiersText<br>• getKeyText<br>• isActionKey | Component                                          |
| MouseListener       | mousePressed<br>mouseReleased<br>mouseEntered<br>mouseExited<br>mouseClicked | MouseEvent<br>• getClickCount<br>• getX<br>• getY<br>• getPoint<br>• translatePoint                | Component                                          |
| MouseMotionListener | mouseDragged<br>mouseMoved                                                   | MouseEvent                                                                                         | Component                                          |

# AWT Hierarchy

|                |                                                                                                                               |                                                                                              |           |
|----------------|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|-----------|
| MouseListener  | mouseWheelMoved                                                                                                               | MouseWheelEvent                                                                              | Component |
|                |                                                                                                                               | <ul style="list-style-type: none"><li>• getWheelRotation</li><li>• getScrollAmount</li></ul> |           |
| WindowListener | windowClosing<br>windowOpened<br>windowIconified<br>windowDeiconified<br>windowClosed<br>windowActivated<br>windowDeactivated | WindowEvent                                                                                  | Window    |
|                |                                                                                                                               | <ul style="list-style-type: none"><li>• getWindow</li></ul>                                  |           |

| Interface           | Methods                              | Parameter/Accessors                                                                 | Events Generated By |
|---------------------|--------------------------------------|-------------------------------------------------------------------------------------|---------------------|
| WindowFocusListener | windowGainedFocus<br>windowLostFocus | WindowEvent                                                                         | Window              |
|                     |                                      | <ul style="list-style-type: none"><li>• getOppositeWindow</li></ul>                 |                     |
| WindowStateListener | windowStateChanged                   | WindowEvent                                                                         | Window              |
|                     |                                      | <ul style="list-style-type: none"><li>• getOldState</li><li>• getNewState</li></ul> |                     |

# *Advanced Programming*

## CSE 201

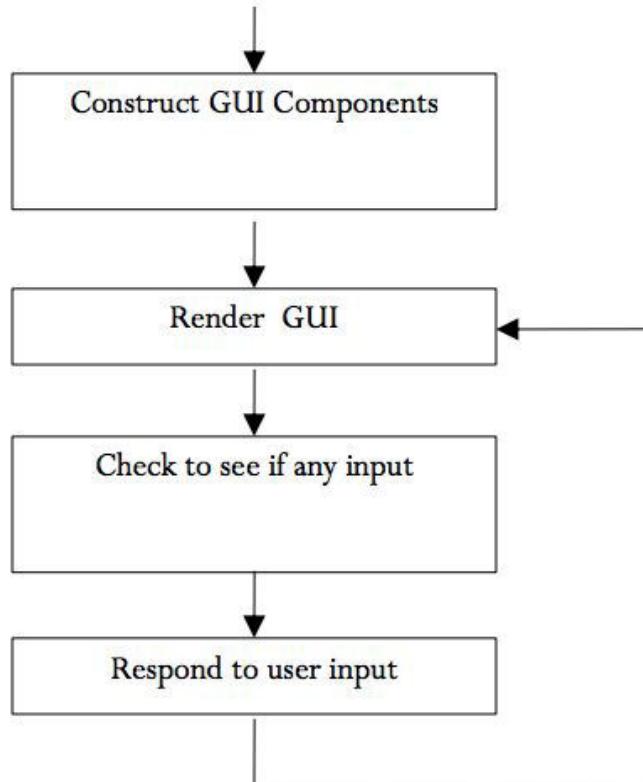
### Instructor: Sambuddho

(Semester: Monsoon 2024)

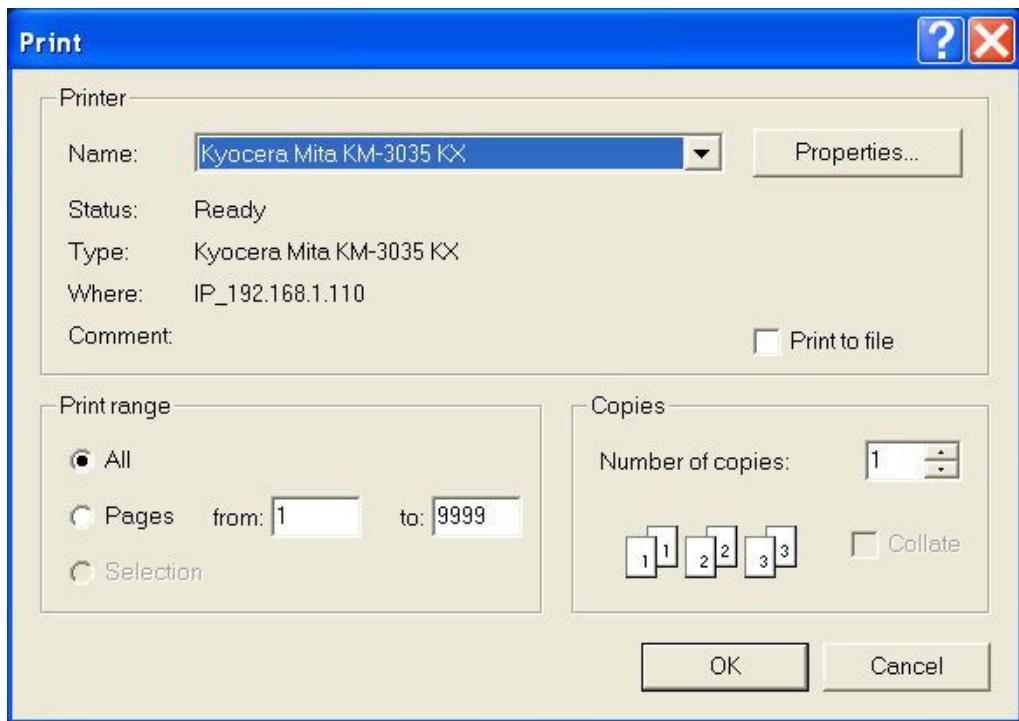
Week 8 – GUI/AWT

# How do GUI Works?

- They loop and respond to events

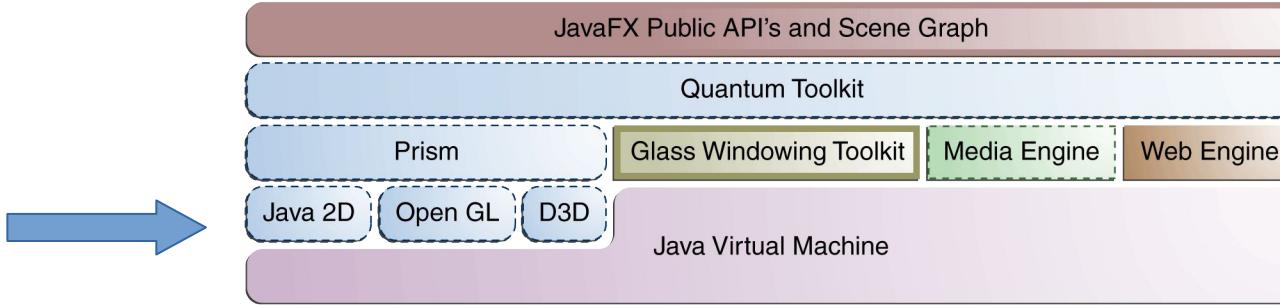


# GUI Examples



# Java Runtime High Level Architecture for GUIs

- You are here



# Creating a Frame

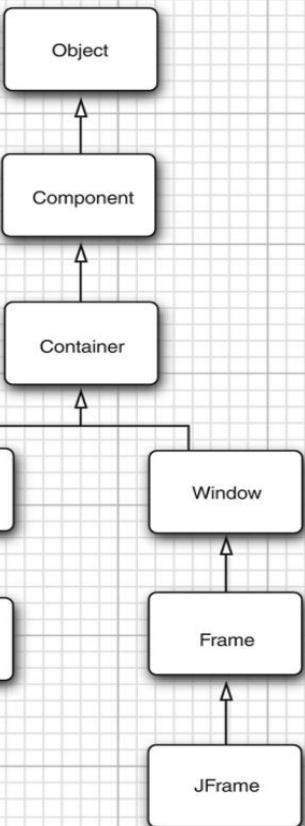
## Swing JFrame

```
1 package simpleFrame;  
2  
3 import java.awt.*;  
4 import javax.swing.*;  
5  
6 /**  
7  * @version 1.34 2018-04-10  
8  * @author Cay Horstmann  
9 */  
10 public class SimpleFrameTest  
11 {  
12     public static void main(String[] args)  
13     {  
14         EventQueue.invokeLater(() ->  
15         {  
16             var frame = new SimpleFrame();  
17             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
18             frame.setVisible(true);  
19         });  
20     }  
21 }  
22 }
```



```
23 class SimpleFrame extends JFrame  
24 {  
25     private static final int DEFAULT_WIDTH = 300;  
26     private static final int DEFAULT_HEIGHT = 200;  
27  
28     public SimpleFrame()  
29     {  
30         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
31     }  
32 }
```

# Frame Properties



## java.awt.Component 1.0

- `boolean isVisible()`
  - `void setVisible(boolean b)`
- resizes the component to the specified width and height.

gets or sets the visible property. Components are initially visible, with the exception of top-level components such as `JFrame`.

- `void setSize(int width, int height) 1.1`
- `void setLocation(int x, int y) 1.1`

moves the component to a new location. The `x` and `y` coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a `JFrame`).

- `void setBounds(int x, int y, int width, int height) 1.1`
- `Dimension getSize() 1.1`
- `void setSize(Dimension d) 1.1`

gets or sets the size property of this component.

## java.awt.Window 1.0

- `void setLocationByPlatform(boolean b) 5`
- gets or sets the `locationByPlatform` property. When the property is set before this window is displayed, the platform picks a suitable location.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`

gets or sets the `resizable` property. When the property is set, the user can resize the frame.

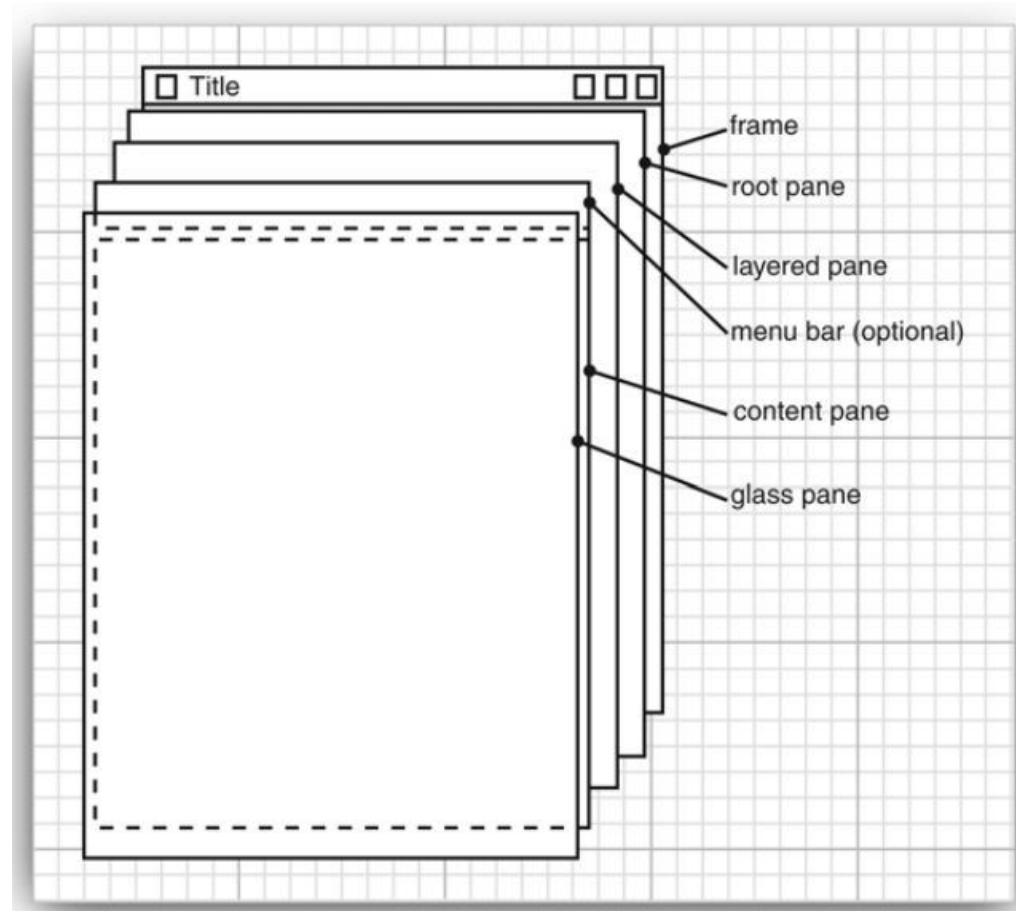
- `String getTitle()`
- `void setTitle(String s)`

gets or sets the `title` property that determines the text in the title bar for the frame.

# Components

*Content pane is the way to go ---->*

```
Component c = . . .;  
frame.add(c); // added to the content pane
```



# Components

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        code for drawing
    }
}
```

```
public class NotHelloWorldComponent extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    public void paintComponent(Graphics g)
    {
        g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
    }
    ...
}
```

```
public class NotHelloWorldComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    ...
    public Dimension getPreferredSize()
    {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```



```
class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
        add(new NotHelloWorldComponent());
        pack();
    }
}
```

# A Basic GUI Program Using AWT

```
1 package notHelloWorld;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 /**
7  * @version 1.34 2018-04-10
8  * @author Cay Horstmann
9  */
10 public class NotHelloWorld
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(() ->
15         {
16             var frame = new NotHelloWorldFrame();
17             frame.setTitle("NotHelloWorld");
18             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20         });
21     }
22 }
23
24 /**
25  * A frame that contains a message panel.
26  */
27 class NotHelloWorldFrame extends JFrame
28 {
29     public NotHelloWorldFrame()
30     {
31         add(new NotHelloWorldComponent());
32         pack();
33     }
34 }
35
36 /**
37  * A component that displays a message.
38  */
39 class NotHelloWorldComponent extends JComponent
40 {
41     public static final int MESSAGE_X = 75;
42     public static final int MESSAGE_Y = 100;
43
44     private static final int DEFAULT_WIDTH = 300;
45     private static final int DEFAULT_HEIGHT = 200;
46
47     public void paintComponent(Graphics g)
48     {
49         g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
50     }
51
52     public Dimension getPreferredSize()
53     {
54         return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
55     }
56 }
```

# 2D Objects

Line2D, Rectangle2D, Ellipse2D etc. ← all implementations of Shape interface.

To draw a shape:

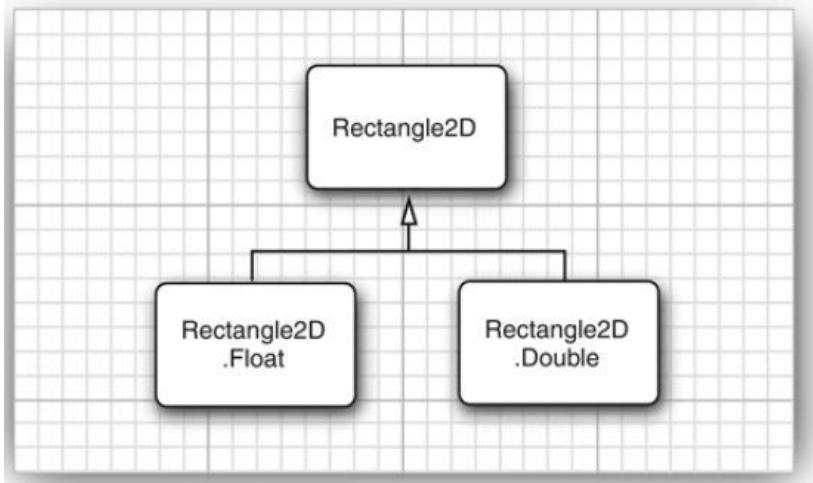
1. Create an object of the class that implements the Shape interface.
2. Then pass on the object to Graphics2D.draw().

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    Rectangle2D rect = . . .;
    g2.draw(rect);

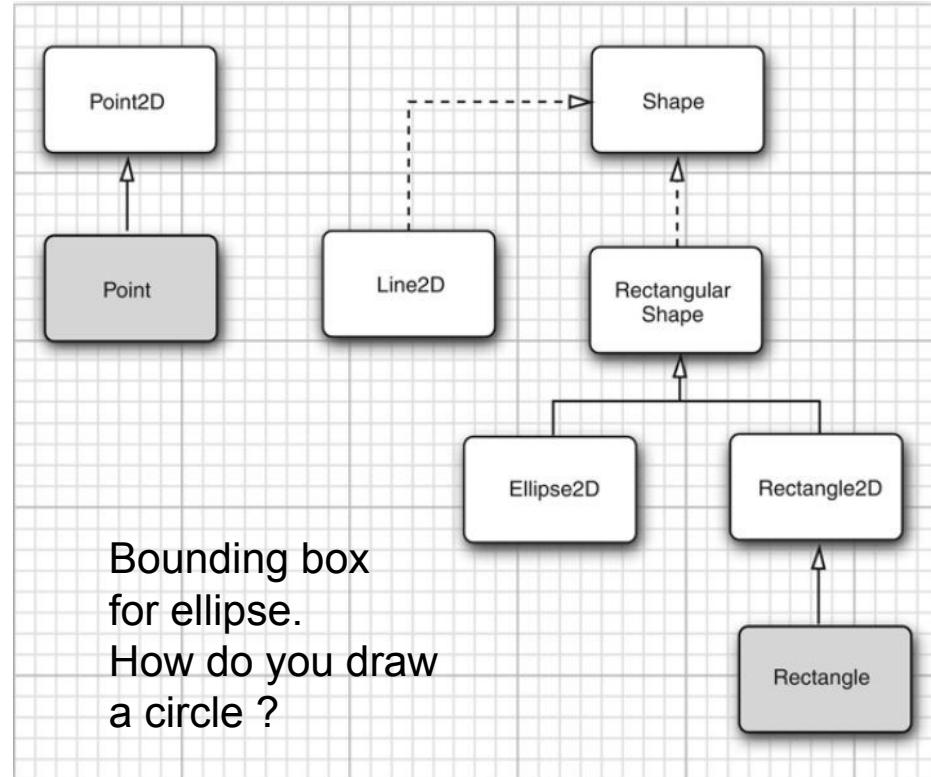
}
```

# 2D Objects

Shapes are of two types – based on floating point or double pixels.



```
var floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
var doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```



# 2D Objects

```
1 package draw;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import javax.swing.*;
6
7 /**
8  * @version 1.34 2018-04-10
9  * @author Cay Horstmann
10 */
11 public class DrawTest
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(() ->
16         {
17             var frame = new DrawFrame();
18             frame.setTitle("DrawTest");
19             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20             frame.setVisible(true);
21         });
22     }
23 }
24
25 /**
26 * A frame that contains a panel with drawings.
27 */
28 class DrawFrame extends JFrame
29 {
30     public DrawFrame()
31     {
32         add(new DrawComponent());
33         pack();
34     }
35 }
```

```
37 /**
38  * A component that displays rectangles and ellipses.
39 */
40 class DrawComponent extends JComponent
41 {
42     private static final int DEFAULT_WIDTH = 400;
43     private static final int DEFAULT_HEIGHT = 400;
44
45     public void paintComponent(Graphics g)
46     {
47         var g2 = (Graphics2D) g;
48
49         // draw a rectangle
50
51         double leftX = 100;
52         double topY = 100;
53         double width = 200;
54         double height = 150;
55
56         var rect = new Rectangle2D.Double(leftX, topY, width, height);
57         g2.draw(rect);
58
59         // draw the enclosed ellipse
60
61         var ellipse = new Ellipse2D.Double();
62         ellipse setFrame(rect);
63         g2.draw(ellipse);
64
65         // draw a diagonal line
66
67         g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
68
69         // draw a circle with the same center
70
71         double centerX = rect.getCenterX();
72         double centerY = rect.getCenterY();
73         double radius = 150;
74
75         var circle = new Ellipse2D.Double();
76         circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
77         g2.draw(circle);
78     }
79 }
```

# 2D Objects - Colors

```
Rectangle2D rect = . . .;  
g2.setPaint(Color.RED);  
g2.fill(rect); // fills rect with red
```

**java.awt.Color** – BLACK, BLUE, CYAN, DARK\_GRAY, GREEN, GRAY, LIGHT\_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW

```
var component = new MyComponent();  
component.setBackground(Color.PINK);
```

## java.awt.Color 1.0

- Color(int r, int g, int b)

creates a color object with the given red, green, and blue components between 0 and 255.

## java.awt.Graphics2D 1.2

- Paint getPaint()
- void setPaint(Paint p)

gets or sets the paint property of this graphics context. The Color class implements the Paint interface. Therefore, you can use this method to set the paint attribute to a solid color.

- void fill(Shape s)

fills the shape with the current paint.

## java.awt.Component 1.0

- Color getForeground()
- Color getBackground()
- void setForeground(Color c)
- void setBackground(Color c)

gets or sets the foreground or background color.

# 2D Objects - Fonts

Font face names available – SansSerif, Serif, Monospaced, Dialog, DialogInput

```
import java.awt.*;  
  
public class ListFonts  
{  
    public static void main(String[] args)  
    {  
        String[] fontNames = GraphicsEnvironment  
            .getLocalGraphicsEnvironment()  
            .getAvailableFontFamilyNames();  
        for (String fontName : fontNames)  
            System.out.println(fontName);  
    }  
}
```

Set font for a string



```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);  
g2.setFont(sansbold14);  
var message = "Hello, World!";  
g2.drawString(message, 75, 100);
```

# 2D Objects - Images

```
Image image = new ImageIcon(filename).getImage();  
  
public void paintComponent(Graphics g)  
{  
    . . .  
    g.drawImage(image, x, y, null);  
}
```

## java.awt.Graphics 1.0

- boolean drawImage(Image img, int x, int y, ImageObserver observer)
- boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

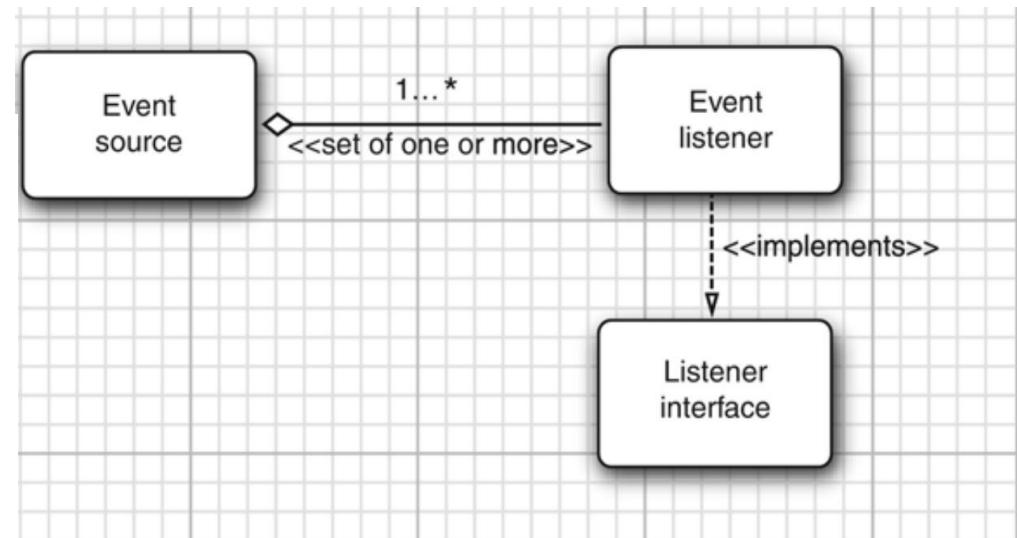
draws an unscaled or scaled image. Note: This call may return before the image is drawn. The `imageObserver` object is notified of the rendering progress. This was a useful feature in the distant past. Nowadays, just pass a `null` observer.

- void copyArea(int x, int y, int width, int height, int dx, int dy)  
copies an area of the screen. The `dx` and `dy` parameters are the distance from the source area to the target area.

# Event Handling

- All events are converted to AWT *event objects* belonging to `java.util.EventObject` (subclasses e.g. `ActionEvent`, `WindowEvent` etc.).

```
class MyListener implements ActionListener
{
    ...
    public void actionPerformed(ActionEvent event)
    {
        // reaction to button click goes here
        ...
    }
}
```



- JButton object creates an ActionEvent and calls `listener.actionPerformed(event)`

# Handling a Button Click Event

```
var yellowButton = new JButton("Yellow");
var blueButton = new JButton("Blue");

var redButton = new JButton("Red");

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```



# Handling a Button Click Event

```
class ColorAction implements ActionListener  
{  
    private Color backgroundColor;  
  
    public ColorAction(Color c)  
    {  
        backgroundColor = c;  
    }  
  
    public void actionPerformed(ActionEvent event)  
    {  
        // set panel background color  
        . . .  
    }  
}
```

```
var yellowAction = new ColorAction(Color.YELLOW);  
var blueAction = new ColorAction(Color.BLUE);  
var redAction = new ColorAction(Color.RED);  
  
yellowButton.addActionListener(yellowAction);  
blueButton.addActionListener(blueAction);  
redButton.addActionListener(redAction);
```

# Handling a Button Click Event

```
1 package button;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * A frame with a button panel.
9 */
10 public class ButtonFrame extends JFrame
11 {
12     private JPanel buttonPanel;
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     public ButtonFrame()
17     {
18         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20         // create buttons
21         var yellowButton = new JButton("Yellow");
22         var blueButton = new JButton("Blue");
23         var redButton = new JButton("Red");
24
25         buttonPanel = new JPanel();
26
27         // add buttons to panel
28         buttonPanel.add(yellowButton);
29         buttonPanel.add(blueButton);
30         buttonPanel.add(redButton);
31
32         // add panel to frame
33         add(buttonPanel);
34
35         // add panel to frame
36         add(buttonPanel);
37
38         // create button actions
39         var yellowAction = new ColorAction(Color.YELLOW);
40         var blueAction = new ColorAction(Color.BLUE);
41         var redAction = new ColorAction(Color.RED);
42
43         // associate actions with buttons
44         yellowButton.addActionListener(yellowAction);
45         blueButton.addActionListener(blueAction);
46         redButton.addActionListener(redAction);
47
48     }
49
50     /**
51      * An action listener that sets the panel's background color.
52      */
53     private class ColorAction implements ActionListener
54     {
55         private Color backgroundColor;
56
57         public ColorAction(Color c)
58         {
59             backgroundColor = c;
60         }
61
62         public void actionPerformed(ActionEvent event)
63         {
64             buttonPanel.setBackground(backgroundColor);
65         }
66     }
67 }
```

# Handling a Button Click Event – Single Event Handler for All Types

```
public void makeButton(String name, Color backgroundColor)
{
    var button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(event ->
        buttonPanel.setBackground(backgroundColor));
}
```

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

# Adapter Classes

- Used to monitor window open/close events.
- An appropriate listener must be required to catch such events.

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

Requires implementing all the methods.

# Adapter Classes

- Alternative – extend and use *adapter* classes
- Adapter classes have dummy do-nothing methods for all the rest of the operations.

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (user agrees)
            System.exit(0);
    }
}
```

```
var listener = new Terminator();
frame.addWindowListener(listener);
```

# Mouse Events

- Three main listener methods – mousePressed, mouseClicked and mouseReleased.
- MouseEvent object sent to the event handler class (adapter or listener implementation).

**java.awt.event.MouseEvent 1.1**

- int getX()
- int getY()
- Point getPoint()

returns the *x* (horizontal) and *y* (vertical) coordinates of the point where the event happened, measured from the top left corner of the component that is the event source.

- int getClickCount()

returns the number of consecutive mouse clicks associated with this event. (The time interval for what constitutes “consecutive” is system-dependent.)

```
private class MouseHandler extends MouseAdapter
{
    public void mousePressed(MouseEvent event)
    {
        // add a new square if the cursor isn't inside a square
        current = find(event.getPoint());
        if (current == null) add(event.getPoint());
    }

    public void mouseClicked(MouseEvent event)
    {
        // remove the current square if double clicked
        current = find(event.getPoint());
        if (current != null && event.getClickCount() >= 2) remove(current);
    }
}
```

# Mouse Event

```
package mouse;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * A component with mouse operations for adding and removing squares.
 */
public class MouseComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private static final int SIDELENGTH = 10;
    private ArrayList<Rectangle2D> squares;
    private Rectangle2D current; // the square containing the mouse cursor

    public MouseComponent()
    {
        squares = new ArrayList<>();
        current = null;
    }

    addMouseListener(new MouseHandler());
    addMouseMotionListener(new MouseMotionHandler());
}

public Dimension getPreferredSize()
{
    return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
}

public void paintComponent(Graphics g)
{
    var g2 = (Graphics2D) g;

    // draw all squares
    for (Rectangle2D r : squares)
        g2.draw(r);
}

/**
 * Finds the first square containing a point.
 * @param p a point
 * @return the first square that contains p
 */
public Rectangle2D find(Point2D p)
{
    for (Rectangle2D r : squares)
    {
        if (r.contains(p)) return r;
    }
    return null;
}

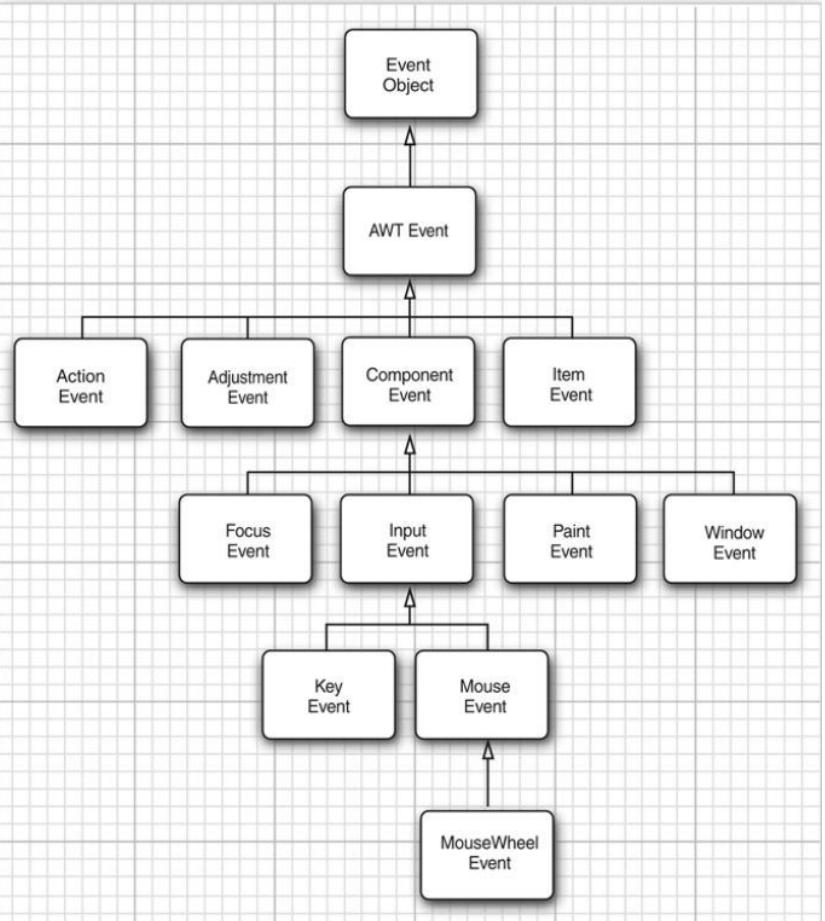
/**
 * Adds a square to the collection.
 * @param p the center of the square
 */
public void add(Point2D p)
{
    double x = p.getX();
    double y = p.getY();

    current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2,
                                    SIDELENGTH, SIDELENGTH);
    squares.add(current);
    repaint();
}
```

# Mouse Event

```
/**  
 * Removes a square from the collection.  
 * @param s the square to remove  
 */  
public void remove(Rectangle2D s)  
{  
    if (s == null) return;  
    if (s == current) current = null;  
    squares.remove(s);  
    repaint();  
}  
  
private class MouseHandler extends MouseAdapter  
{  
    public void mousePressed(MouseEvent event)  
    {  
        // add a new square if the cursor isn't inside a square  
        current = find(event.getPoint());  
        if (current == null) add(event.getPoint());  
    }  
  
    public void mouseClicked(MouseEvent event)  
    {  
        // remove the current square if double clicked  
        current = find(event.getPoint());  
        if (current != null && event.getClickCount() >= 2) remove(current);  
    }  
}  
  
private class MouseMotionHandler implements MouseMotionListener  
{  
    public void mouseMoved(MouseEvent event)  
    {  
        // set the mouse cursor to cross hairs if it is inside a rectangle  
  
        if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());  
        else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));  
    }  
  
    public void mouseDragged(MouseEvent event)  
    {  
        if (current != null)  
        {  
            int x = event.getX();  
            int y = event.getY();  
  
            // drag the current rectangle to center it at (x, y)  
            current setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);  
            repaint();  
        }  
    }  
}
```

# AWT Hierarchy



| Interface           | Methods                                                                      | Parameter/Accessors                                                                                                                                                 | Events Generated By                                |
|---------------------|------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|
| ActionListener      | actionPerformed                                                              | ActionEvent <ul style="list-style-type: none"><li>• getActionCommand</li><li>• getModifiers</li></ul>                                                               | AbstractButton<br>JComboBox<br>JTextField<br>Timer |
| AdjustmentListener  | adjustmentValueChanged                                                       | AdjustmentEvent <ul style="list-style-type: none"><li>• getAdjustable</li><li>• getAdjustmentType</li></ul>                                                         | JScrollbar                                         |
| Interface           | Methods                                                                      | Parameter/Accessors                                                                                                                                                 | Events Generated By                                |
| ItemListener        | itemStateChanged                                                             | ItemEvent <ul style="list-style-type: none"><li>• getItem</li><li>• getItemSelectable</li><li>• getStateChange</li></ul>                                            | AbstractButton<br>JComboBox                        |
| FocusListener       | focusGained<br>focusLost                                                     | FocusEvent <ul style="list-style-type: none"><li>• isTemporary</li></ul>                                                                                            | Component                                          |
| KeyListener         | keyPressed<br>keyReleased<br>keyTyped                                        | KeyEvent <ul style="list-style-type: none"><li>• getKeyChar</li><li>• getKeyCode</li><li>• getKeyModifiersText</li><li>• getKeyText</li><li>• isActionKey</li></ul> | Component                                          |
| MouseListener       | mousePressed<br>mouseReleased<br>mouseEntered<br>mouseExited<br>mouseClicked | MouseEvent <ul style="list-style-type: none"><li>• getClickCount</li><li>• getX</li><li>• getY</li><li>• getPoint</li><li>• translatePoint</li></ul>                | Component                                          |
| MouseMotionListener | mouseDragged<br>mouseMoved                                                   | MouseEvent                                                                                                                                                          | Component                                          |

# AWT Hierarchy

|                    |                                                                                                                               |                                                                                              |           |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|-----------|
| MouseWheelListener | mouseWheelMoved                                                                                                               | MouseWheelEvent                                                                              | Component |
|                    |                                                                                                                               | <ul style="list-style-type: none"><li>• getWheelRotation</li><li>• getScrollAmount</li></ul> |           |
| WindowListener     | windowClosing<br>windowOpened<br>windowIconified<br>windowDeiconified<br>windowClosed<br>windowActivated<br>windowDeactivated | WindowEvent                                                                                  | Window    |

| Interface           | Methods                              | Parameter/Accessors                                                                 | Events Generated By |
|---------------------|--------------------------------------|-------------------------------------------------------------------------------------|---------------------|
| WindowFocusListener | windowGainedFocus<br>windowLostFocus | WindowEvent                                                                         | Window              |
|                     |                                      | <ul style="list-style-type: none"><li>• getOppositeWindow</li></ul>                 |                     |
| WindowStateListener | windowStateChanged                   | WindowEvent                                                                         | Window              |
|                     |                                      | <ul style="list-style-type: none"><li>• getOldState</li><li>• getNewState</li></ul> |                     |

# *Advanced Programming*

## CSE 201

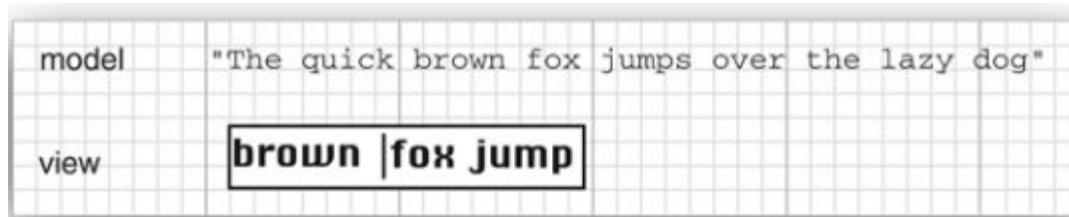
**Instructor: Sambuddho**

(Semester: Monsoon 2024)

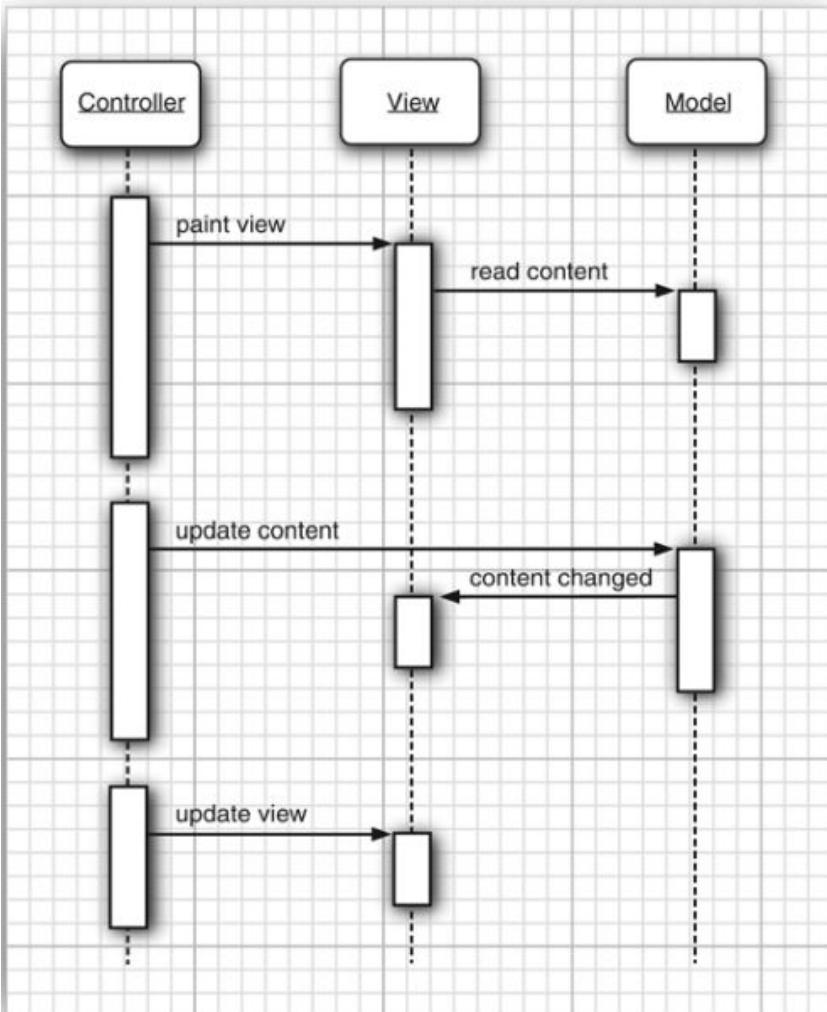
Week 9 – Swing MVC

# Model View Controller

- Design pattern for using AWT/Swing packages
  - Model: Data to be displayed/used.
  - View: How the data should appear and be displayed.
  - Controller: Event handling.



# Interaction Between Model, View and Controller



```
var button = new JButton("Blue");
ButtonModel model = button.getModel();
```

| Property Name | Value                                                                          |
|---------------|--------------------------------------------------------------------------------|
| actionCommand | The action command string associated with this button                          |
| mnemonic      | The keyboard mnemonic for this button                                          |
| armed         | true if the button was pressed and the mouse is still over the button          |
| enabled       | true if the button is selectable                                               |
| pressed       | true if the button was pressed but the mouse button hasn't yet been released   |
| rollover      | true if the mouse is over the button                                           |
| selected      | true if the button has been toggled on (used for checkboxes and radio buttons) |

# Layout Management



```
panel.setLayout(new GridLayout(4, 4));
```



## java.awt.Container 1.0

- `void setLayout(LayoutManager m)`  
sets the layout manager for this container.
- `Component add(Component c)`
- `Component add(Component c, Object constraints) 1.1`  
adds a component to this container and returns the component reference.

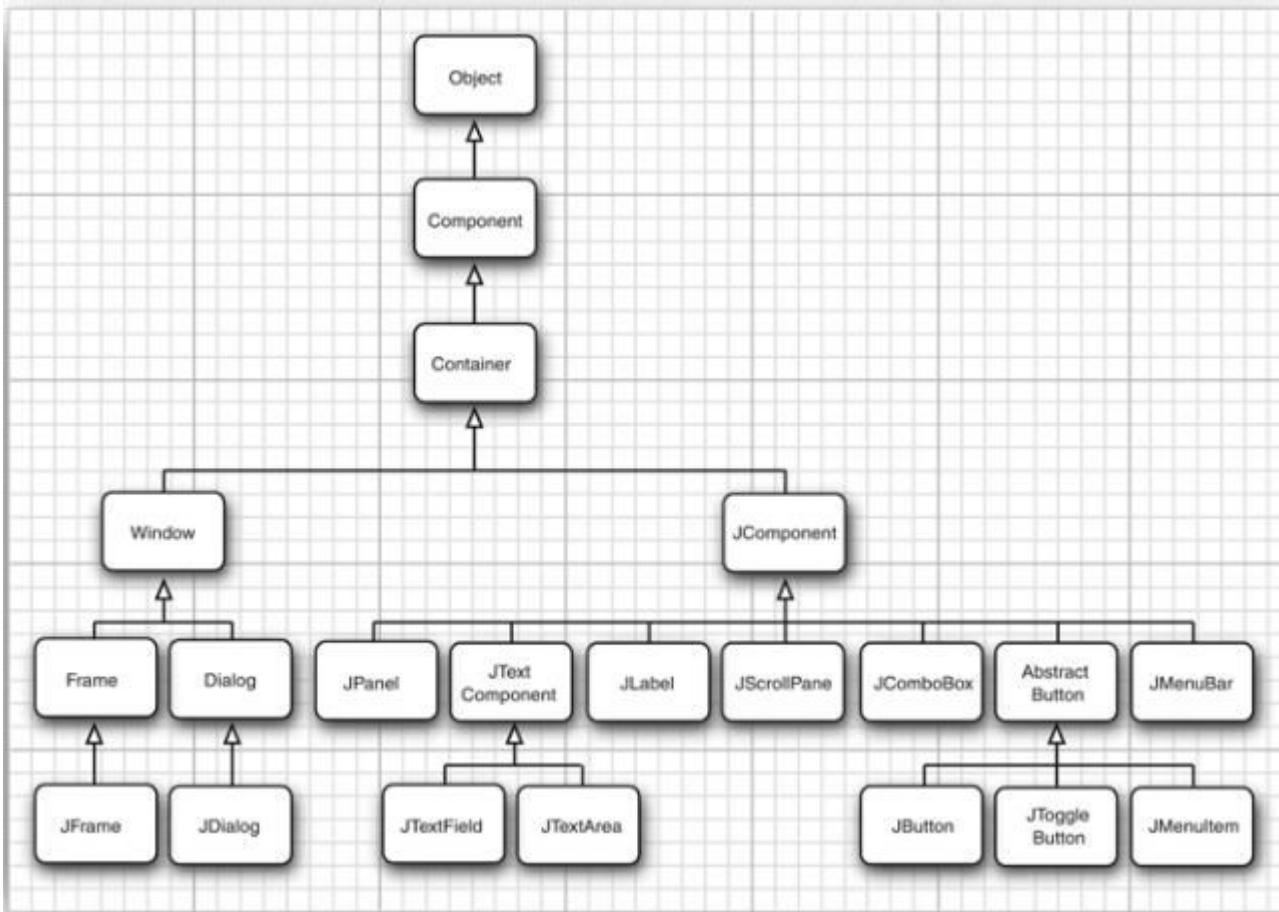
## java.awt.FlowLayout 1.0

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`  
constructs a new FlowLayout. The align parameter is one of LEFT, CENTER, or RIGHT.

## java.awt.GridLayout 1.0

- `GridLayout(int rows, int columns)`
- `GridLayout(int rows, int columns, int hgap, int vgap)`  
constructs a new GridLayout. One of rows and columns (but not both) may be zero, denoting an arbitrary number of components per row or column.

# Component Inheritances



# Text Input

```
var panel = new JPanel();
var textField = new JTextField("Default input", 20);
panel.add(textField);
```

- Default
- string

- JTextField vs  
JTextArea

Single line vs a block.

## javax.swing.text.JTextComponent 1.2

- String getText()
- void setText(String text)

gets or sets the text of this text component.

- boolean isEditable()
- void setEditable(boolean b)

gets or sets the `editable` property that determines whether the user can edit the content of this text component.

# Password Fields – Type of Text Fields

## `javax.swing.JPasswordField 1.2`

- `JPasswordField(String text, int columns)`  
constructs a new password field.
- `void setEchoChar(char echo)`  
sets the echo character for this password field. This is advisory; a particular look-and-feel may insist on its own choice of echo character. A value of 0 resets the echo character to the default.
- `char[] getPassword()`  
returns the text contained in this password field. For stronger security, you should overwrite the content of the returned array after use. (The password is not returned as a String because a string would stay in the virtual machine until it is garbage-collected.)

# Text Areas

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each
```

```
textArea.setLineWrap(true); // long lines are wrapped
```

```
textArea = new JTextArea(8, 40);
var scrollPane = new JScrollPane(textArea);
```

## javax.swing.JScrollPane 1.2

- `JScrollPane(Component c)`

creates a scroll pane that displays the content of the specified component. Scrollbars are supplied when the component is larger than the view.

## javax.swing.JTextArea 1.2

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String text, int rows, int cols)`

constructs a new text area.
- `void setColumns(int cols)`

tells the text area the preferred number of columns it should use.
- `void setRows(int rows)`

tells the text area the preferred number of rows it should use.
- `void append(String newText)`

appends the given text to the end of the text already in the text area.
- `void setLineWrap(boolean wrap)`

turns line wrapping on or off.

## javax.swing.JTextArea 1.2 (Continued)

- `void setWrapStyleWord(boolean word)`

If `word` is `true`, long lines are wrapped at word boundaries. If it is `false`, long lines are broken without taking word boundaries into account.
- `void setTabSize(int c)`

sets tab stops every `c` columns. Note that the tabs aren't converted to spaces but cause alignment with the next tab stop.

# Text Areas

```
1 package text;
2
3 import java.awt.BorderLayout;
4 import java.awt.GridLayout;
5
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8 import javax.swing.JLabel;
9 import javax.swing.JPanel;
10 import javax.swing.JPasswordField;
11 import javax.swing.JScrollPane;
12 import javax.swing.JTextArea;
13 import javax.swing.JTextField;
14 import javax.swing.SwingConstants;
15
16 /**
17  * A frame with sample text components.
18 */
19 public class TextComponentFrame extends JFrame
20 {
21     public static final int TEXTAREA_ROWS = 8;
22     public static final int TEXTAREA_COLUMNS = 20;
23
24     public TextComponentFrame()
25     {
26         var textField = new JTextField();
27         var passwordField = new JPasswordField();
28
29         var northPanel = new JPanel();
30         northPanel.setLayout(new GridLayout(2, 2));
31         northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
32
33         northPanel.add(textField);
34         northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
35         northPanel.add(passwordField);
36
37         add(northPanel, BorderLayout.NORTH);
38
39         var textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
40         var scrollPane = new JScrollPane(textArea);
41
42         add(scrollPane, BorderLayout.CENTER);
43
44         // add button to append text into the text area
45
46         var southPanel = new JPanel();
47
48         var insertButton = new JButton("Insert");
49         southPanel.add(insertButton);
50         insertButton.addActionListener(event ->
51             textArea.append("User name: " + textField.getText() + " Password: "
52                         + new String(passwordField.getPassword()) + "\n"));
53
54         add(southPanel, BorderLayout.SOUTH);
55
56     }
57 }
```

---

# Checkboxes

- Used to collect options.
- Come with labels that identify them.
- Upon checking action is triggered as an ActionEvent.



```
bold = new JCheckBox("Bold");
bold.setSelected(true);

ActionListener listener = . . .;
bold.addActionListener(listener);
italic.addActionListener(listener);

ActionListener listener = event -> {
    int mode = 0;
    if (bold.isSelected()) mode += Font.BOLD;
    if (italic.isSelected()) mode += Font.ITALIC;
    label.setFont(new Font(Font.SERIF, mode, FONTSIZE));
};
```

# Radio Buttons

- Used to select one of several options.
- Event notification on clicking the button.

- Button Label      Selected/Unselected

```
var group = new ButtonGroup();
```

```
var smallButton = new JRadioButton("Small", false);  
group.add(smallButton);
```

```
var mediumButton = new JRadioButton("Medium", true);  
group.add(mediumButton);
```

```
....
```

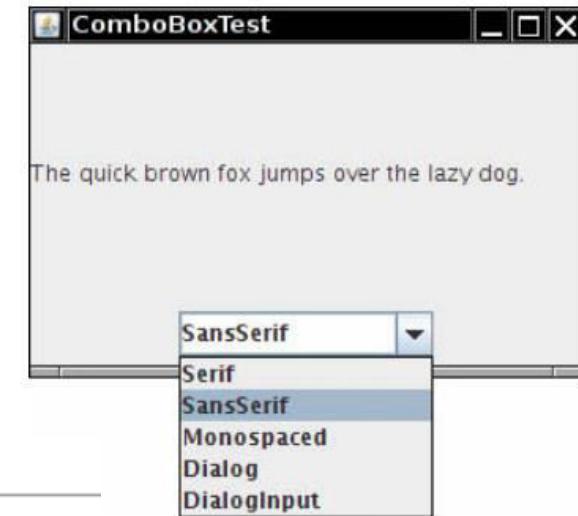


# Radio Buttons

```
1 package radioButton;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8 * A frame with a sample text label and radio buttons for selecting font size
9 */
10 public class RadioButtonFrame extends JFrame
11 {
12     private JPanel buttonPanel;
13     private ButtonGroup group;
14     private JLabel label;
15     private static final int DEFAULT_SIZE = 36;
16
17     public RadioButtonFrame()
18     {
19         // add the sample text label
20
21         label = new JLabel("The quick brown fox jumps over the lazy dog.");
22         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
23
24         add(label, BorderLayout.CENTER);
25
26         // add the radio buttons
27
28         buttonPanel = new JPanel();
29         group = new ButtonGroup();
30
31         addRadioButton("Small", 8);
32         addRadioButton("Medium", 12);
33         addRadioButton("Large", 18);
34         addRadioButton("Extra large", 36);
35
36         add(buttonPanel, BorderLayout.SOUTH);
37         pack();
38
39     /**
40      * Adds a radio button that sets the font size of the sample text.
41      * @param name the string to appear on the button
42      * @param size the font size that this button sets
43      */
44     public void addRadioButton(String name, int size)
45     {
46         boolean selected = size == DEFAULT_SIZE;
47         var button = new JRadioButton(name, selected);
48         group.add(button);
49         buttonPanel.add(button);
50
51         // this listener sets the label font size
52
53         ActionListener listener = event -> label.setFont(new Font("Serif", Font.PLAIN, size));
54
55         button.addActionListener(listener);
56     }
57 }
```

# Combo Boxes

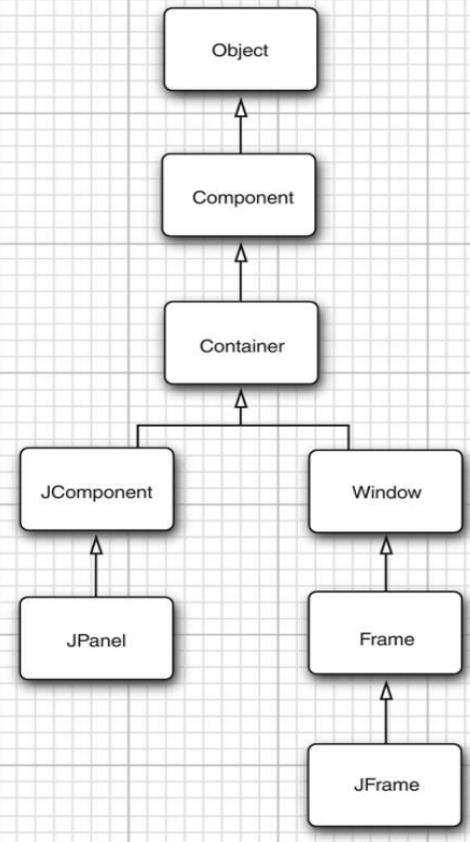
- Alternatives to give to users.
- E.g. font selections.
- Event on selection.
- JComboBox



## javax.swing.JComboBox 1.2

- boolean `isEditable()`
- void `setEditable(boolean b)`  
gets or sets the editable property of this combo box.
- void `addItem(Object item)`  
adds an item to the item list.
- void `insertItemAt(Object item, int index)`  
inserts an item into the item list at a given index.
- void `removeItem(Object item)`  
removes an item from the item list.
- void `removeItemAt(int index)`  
removes the item at an index.
- void `removeAllItems()`  
removes all items from the item list.
- Object `getSelectedItem()`  
returns the currently selected item.

# Frame Properties



## java.awt.Component 1.0

- `boolean isVisible()`
- `void setVisible(boolean b)`

gets or sets the visible property. Components are initially visible, with the exception of top-level components such as `JFrame`.

- `void setSize(int width, int height) 1.1`  
resizes the component to the specified width and height.
- `void setLocation(int x, int y) 1.1`

moves the component to a new location. The `x` and `y` coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a `JFrame`).

- `void setBounds(int x, int y, int width, int height) 1.1`  
moves and resizes this component.
- `Dimension getSize() 1.1`
- `void setSize(Dimension d) 1.1`  
gets or sets the size property of this component.

## java.awt.Window 1.0

- `void setLocationByPlatform(boolean b) 5`

gets or sets the `locationByPlatform` property. When the property is set before this window is displayed, the platform picks a suitable location.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0 (Continued)

- `Image getIconImage()`
- `void setIconImage(Image image)`

gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

## java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`

gets or sets the `resizable` property. When the property is set, the user can resize the frame.

- `String getTitle()`
- `void setTitle(String s)`

gets or sets the `title` property that determines the text in the title bar for the frame.

# Combo Boxes

```
1 package comboBox;
2
3 import java.awt.BorderLayout;
4 import java.awt.Font;
5
6 import javax.swing.JComboBox;
7 import javax.swing.JFrame;
8 import javax.swing.JLabel;
9 import javax.swing.JPanel;
10
11 /**
12 * A frame with a sample text label and a combo box for selecting font faces.
13 */
14 public class ComboBoxFrame extends JFrame
15 {
16     private JComboBox<String> faceCombo;
17     private JLabel label;
18     private static final int DEFAULT_SIZE = 24;
19
20     public ComboBoxFrame()
21     {
22         // add the sample text label
23
24         label = new JLabel("The quick brown fox jumps over the lazy dog.");
25         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
26         add(label, BorderLayout.CENTER);
27
28         // make a combo box and add face names
29
30         faceCombo = new JComboBox<>();
31         faceCombo.addItem("Serif");
32         faceCombo.addItem("SansSerif");
33         faceCombo.addItem("Monospaced");
34         faceCombo.addItem("Dialog");
35
36         // the combo box listener changes the label font to the selected face name
37         faceCombo.addActionListener(event ->
38             label.setFont(
39                 new Font(faceCombo.getItemAt(faceCombo.getSelectedIndex()),
40                     Font.PLAIN, DEFAULT_SIZE)));
41
42         // add combo box to a panel at the frame's southern border
43
44         var comboPanel = new JPanel();
45         comboPanel.add(faceCombo);
46         add(comboPanel, BorderLayout.SOUTH);
47         pack();
48     }
49
50 }
51 }
```

# Menus

- Menu bars > Menu items > Submenu items

```
var menuBar = new JMenuBar();
```

```
var editMenu = new JMenu("Edit");
```

```
menuBar.add(editMenu);
```

```
var pasteItem = new JMenuItem("Paste");
```

```
editMenu.add(pasteItem);
```

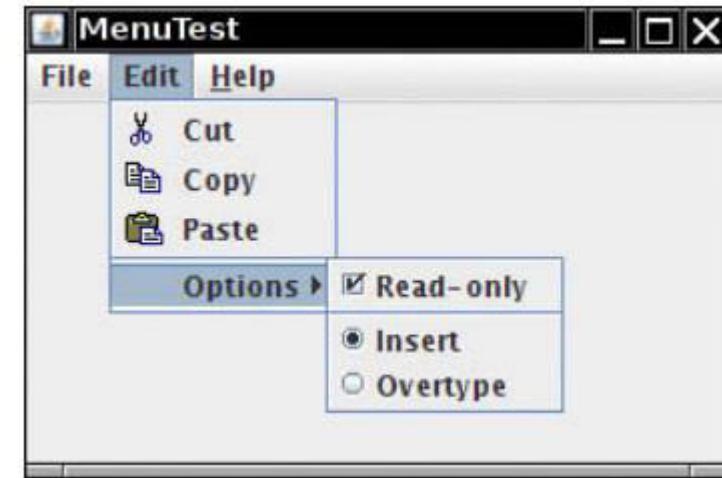
```
editMenu.addSeparator();
```

```
JMenu optionsMenu = . . .; // a submenu
```

```
editMenu.add(optionsMenu);
```

```
ActionListener listener = . . .;
```

```
pasteItem.addActionListener(listener);
```



# Menus with Icons

```
var cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));

cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

# Checkboxes and Radio Buttons in Menu Items

```
var readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

```
var group = new ButtonGroup();
var insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
var overtypeItem = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

*Advanced Programming*  
**CSE 201**  
**Instructor: Sambuddho**

(Semester: Monsoon 2024)  
Week 10 – File Streams and I/O Operations

# Reading and Writing to files.

- Input stream: Used to read sequence of bytes (could be anything “representable” as a file).
- Output stream: Used to write sequence of bytes (could be anything “representable” as a file).

# Reading and Writing to files.

- `InputStream : abstract int read()`
  - [ Return values: -1 if error, or the byte read (upcasted to an int {4-bytes}). ]
- `OutputStream : abstract void write(int b)`
  - [ Write a single byte (upcasted as an int). ]

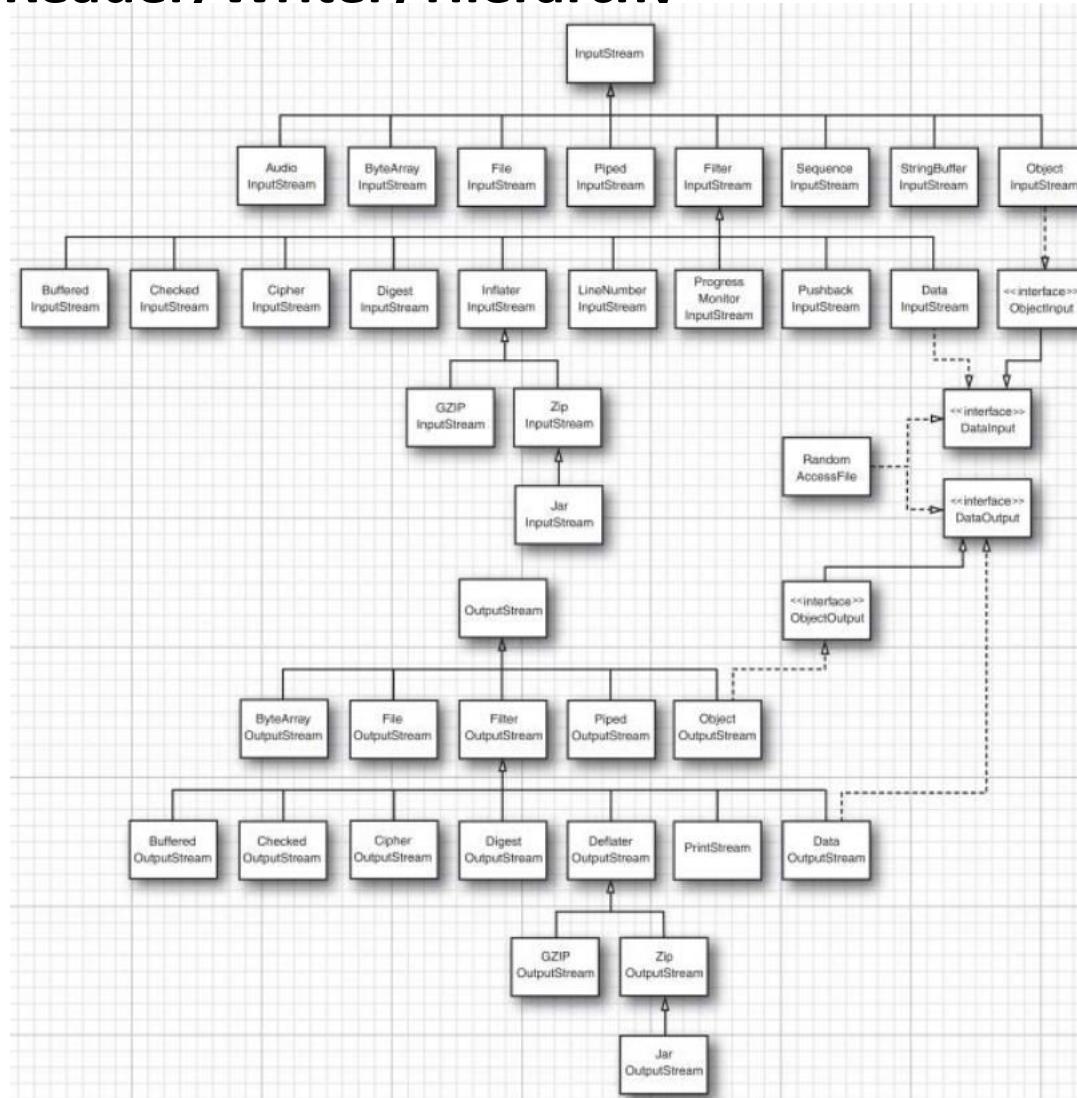
## `java.io.InputStream 1.0`

- `abstract int read()`  
reads a byte of data and returns the byte read; returns -1 at the end of the input stream.
- `int read(byte[] b)`  
reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the input stream; this method reads at most `b.length` bytes.
- `int read(byte[] b, int off, int len)`  
reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the input stream.  
*Parameters:*    `b`      The array into which the data is read  
                  `off`      The offset into `b` where the first bytes should be placed  
                  `len`      The maximum number of bytes to read
- `long skip(long n)`  
skips `n` bytes in the input stream, returns the actual number of bytes skipped (which may be less than `n` if the end of the input stream was encountered).
- `int available()`  
returns the number of bytes available, without blocking (recall that blocking means that the current thread loses its turn).
- `void close()`  
closes the input stream.
- `void mark(int readlimit)`  
puts a marker at the current position in the input stream (not all streams support this feature). If more than `readlimit` bytes have been read from the input stream, the stream is allowed to forget the marker.
- `void reset()`  
returns to the last marker. Subsequent calls to `read` reread the bytes. If there is no current marker, the input stream is not reset.
- `boolean markSupported()`  
returns true if the input stream supports marking.

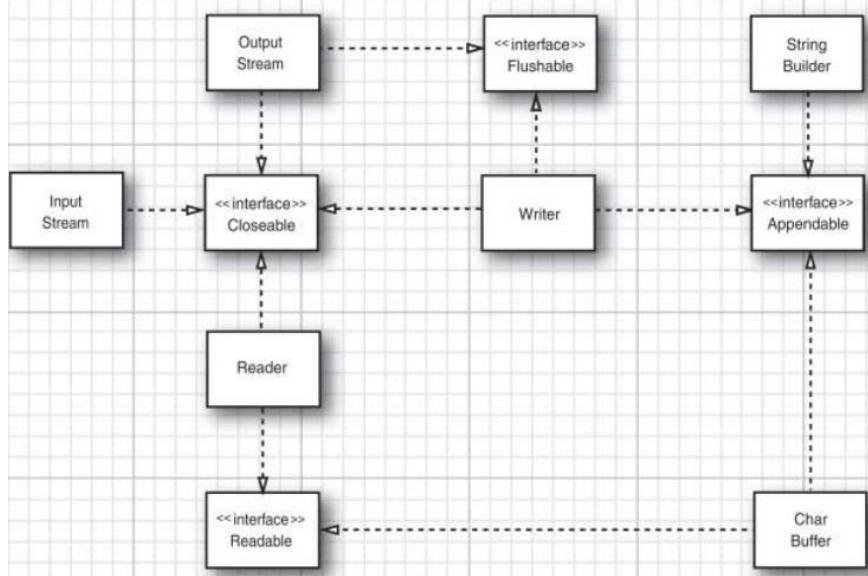
## `java.io.OutputStream 1.0`

- `abstract void write(int n)`  
writes a byte of data.
- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`  
writes all bytes or a range of bytes in the array `b`.  
*Parameters:*    `b`      The array from which to write the data  
                  `off`      The offset into `b` to the first byte that will be written  
                  `len`      The number of bytes to write
- `void close()`  
flushes and closes the output stream.
- `void flush()`  
flushes the output stream—that is, sends any buffered data to its destination.

# I/O Stream (plus Reader/Writer) Hierarchy



# I/O Stream (plus Reader/Writer) Hierarchy



## `java.io.Closeable` 5.0

- `void close()`

closes this `Closeable`. This method may throw an `IOException`.

## `java.io.Flushable` 5.0

- `void flush()`

flushes this `Flushable`.

## `java.lang.Appendable` 5.0

- `Appendable append(char c)`
- `Appendable append(CharSequence cs)`

appends the given code unit, or all code units in the given sequence, to this `Appendable`; returns this.

## `java.lang.CharSequence` 1.4

- `char charAt(int index)`  
returns the code unit at the given index.
- `int length()`  
returns the number of code units in this sequence.
- `CharSequence subSequence(int startIndex, int endIndex)`  
returns a `CharSequence` consisting of the code units stored from index `startIndex` to `endIndex - 1`.
- `String toString()`  
returns a string consisting of the code units of this sequence.

# Reading Writing from Files

FileInputStream – Read bytes from file.

FileOutputStream – Write bytes to file

```
FileInputStream fin = new FileInputStream("employee.dat");
```

```
byte b = (byte) fin.read();
```

```
DataInputStream din = . . .;  
double x = din.readDouble();
```

```
FileInputStream fin = new FileInputStream("employee.dat");  
DataInputStream din = new DataInputStream(fin);  
double x = din.readDouble();
```



```
DataInputStream din = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("employee.dat")));
```

# Reading Writing from Files -- PushbackInputStream

```
PushbackInputStream pbin = new PushbackInputStream(  
    new BufferedInputStream(  
        new FileInputStream("employee.dat")));  
  
int b = pbin.read();  
  
if (b != '<') pbin.unread(b);
```

## java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

constructs an input stream with one-byte lookahead or a pushback buffer of specified size.

- `void unread(int b)`

pushes back a byte, which is retrieved again by the next call to `read`.

*Parameters:*    `b`      The byte to be read again

## java.io.FileInputStream 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

creates a new file input stream using the file whose path name is specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.

## java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

creates a new file output stream specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) If the `append` parameter is `true`, an existing file with the same name will not be deleted and data will be added at the end of the file. Otherwise, this method deletes any existing file with the same name.

## java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

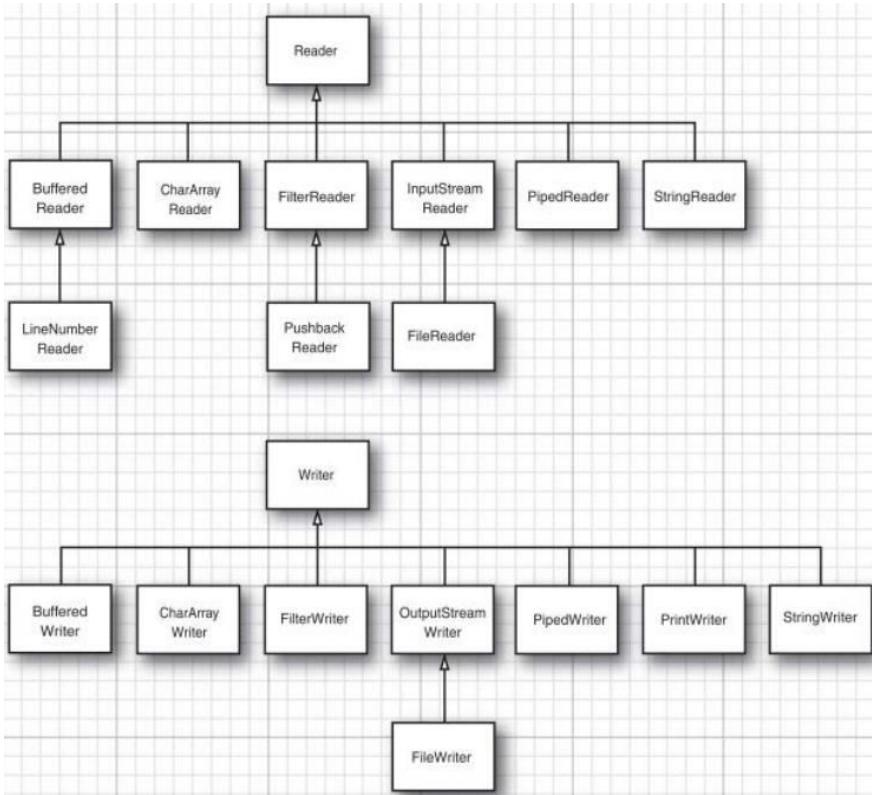
creates a buffered input stream. A buffered input stream reads bytes from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

## java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

creates a buffered output stream. A buffered output stream collects bytes to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

# Text I/O – Reader/Writer



```
Reader in = new InputStreamReader(System.in);
```

```
Reader in = new InputStreamReader(new FileInputStream("data.txt"), StandardCharsets.UTF_8);
```

- `char b = (char) in.read();`

## Constructor Summary

### Constructors

#### Constructor and Description

`InputStreamReader(InputStream in)`

Creates an InputStreamReader that uses the default charset.

`InputStreamReader(InputStream in, Charset cs)`

Creates an InputStreamReader that uses the given charset.

`InputStreamReader(InputStream in, CharsetDecoder dec)`

Creates an InputStreamReader that uses the given charset decoder.

`InputStreamReader(InputStream in, String charsetName)`

Creates an InputStreamReader that uses the named charset.

## Method Summary

### All Methods | Instance Methods | Concrete Methods

#### Modifier and Type

void

#### Method and Description

`close()`

Closes the stream and releases any system resources associated with it.

#### String

`getEncoding()`

Returns the name of the character encoding being used by this stream.

int

`read()`

Reads a single character.

int

`read(char[] cbuf, int offset, int length)`

Reads characters into a portion of an array.

boolean

`ready()`

Tells whether this stream is ready to be read.

# Text I/O – Buffered Reader

- Reader br = new BufferedReader(new BufferedInputStream(new InputStreamReader("myfile")));
- String next\_str = br.readLine(); // Read an entire line.
- char next\_char = br.read(); // read single character.
- List<String> lines = Files.readAllLines(path, charset); // read multiple lines as strings for file path.
- Alternative way to read lines of a file without the List<>.
- InputStream inputStream = . . .;
- try (BufferedReader in = new BufferedReader(new InputStreamReader(inputStream,  
StandardCharsets.UTF\_8)))
- {
- String line;
- while ((line = in.readLine()) != null)
- {
- do something with line
- }
- }

# Text I/O – File Writer

```
PrintWriter out = new PrintWriter(  
    new FileOutputStream("employee.txt"), "UTF-8");
```

```
String name = "Harry Hacker";  
double salary = 75000;  
out.print(name);  
out.print(' ');  
out.println(salary);
```



Harry Hacker 75000.0

# Reading and Writing Binary Data

- *DataInput* and *DataOutput* interfaces:
- `readChars()`, `readByte()`, `readInt()`, `readShort()`, `readLong()`, `readFloat()`, `readDouble()`, `readChar()`, `readBoolean()`, `readUTF()`.
- `writeChars()`, `writeByte()`, `writeInt()`, `writeShort()`, `writeLong()`, `writeFloat()`, `writeDouble()`, `writeChar()`, `writeBoolean()`, `writeUTF()`.
- Using *DataInputStream* and *DataOutputStream*:
- `DataInputStream in = new DataInputStream(new FileInputStream("employee.dat"));`
- `DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));`



# Reading and Writing Binary Data

## `java.io.DataInput 1.0`

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

reads in a value of the given type.

- `void readFully(byte[] b)`

reads bytes into the array `b`, blocking until all bytes are read.

*Parameters:*      `b`        The buffer into which the data are read

- `void readFully(byte[] b, int off, int len)`

reads bytes into the array `b`, blocking until all bytes are read.

*Parameters:*      `b`        The buffer into which the data are read

`off`      The start offset of the data

`len`      The maximum number of bytes to read

- `String readUTF()`

reads a string of characters in the “modified UTF-8” format.

- `int skipBytes(int n)`

skips `n` bytes, blocking until all bytes are skipped.

*Parameters:*      `n`        The number of bytes to be skipped

## `java.io.DataOutput 1.0`

- `void writeBoolean(boolean b)`
- `void writeByte(int b)`
- `void writeChar(int c)`
- `void writeDouble(double d)`
- `void writeFloat(float f)`
- `void writeInt(int i)`
- `void writeLong(long l)`
- `void writeShort(int s)`

writes a value of the given type.

- `void writeChars(String s)`

writes all characters in the string.

- `void writeUTF(String s)`

writes a string of characters in the “modified UTF-8” format.

# Random Access Files

- - Files are sequential access.
- - Random access files can help access at any location (seek at random locations) – has a *“file pointer”* much like C/C++.
- - RandomAccessFile implements both DataInput and DataOutput interfaces – use methods like readInt()/writeInt() etc.

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("employee.dat", "rw");
```

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee e = new Employee();
e.readData(in);
```

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

## Zip Archive Files

- - Used for accessing compressed (.zip) files.
- - ZipInputStream class to read a .zip file.
- - ZipOutputStream class to write to a .zip file.
- - Compressed file stream stored in blocks, aka ``entires''.

```
ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    InputStream in = zin.getInputStream(entry);
    read the contents of in
    zin.closeEntry();
}
zin.close();
```

## Zip Archive Files

- Writing to a ZIP file – use ZipOutputStream class.
- `FileOutputStream fout = new FileOutputStream("test.zip");`
- `ZipOutputStream zout = new ZipOutputStream(fout);`
- *for all files*
  - {
  - `ZipEntry ze = new ZipEntry(filename);`
  - `zout.putNextEntry(ze);`
  - `zout.write(...);`
  - `zout.closeEntry();`
  - }
  - `zout.close();`

## Path Interface and Files Class

- Used to work with the file system for operations like create file, delete file, create directory etc.
- Paths – Sequence of directory names, optionally followed by a file name. The first component is the *root* [e.g. / or C:\ ].

```
Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

(Throws InvalidPathException if it is not a valid path)

# Path Interface and Files Class

- Read path as a string from a config file.
- 

```
String baseDir = props.getProperty("base.dir");
    // May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK that baseDir has separators
```

## ***Path resolution:***

To resolve a path, e.g. to find the working directory relative to a known path:

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);      OR
```

```
Path workPath = basePath.resolve("work");
```

# Reading Files

Read bytes from a file into a byte array.

```
byte[] bytes = Files.readAllBytes(path);
```

Read all lines are a collection of strings.

```
List<String> lines = Files.readAllLines(path, charset);
```

Write bytes to a file.

```
Files.write(path, content.getBytes(charset));
```

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

Reading large files:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

```
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

## java.nio.file.Files 7

---

- static byte[] readAllBytes(Path path)
- static List<String> readAllLines(Path path, Charset charset)  
reads the contents of a file.
- static Path write(Path path, byte[] contents, OpenOption... options)
- static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)  
writes the given contents to a file and returns path.

- static InputStream newInputStream(Path path, OpenOption... options)
- static OutputStream newOutputStream(Path path, OpenOption... options)
- static BufferedReader newBufferedReader(Path path, Charset charset)
- static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)  
opens a file for reading or writing.

# Directories

Creating directory from path.

```
Files.createDirectory(path);
```

Creating intermediate directories.

```
Files.createDirectories(path);
```

Creating and empty file in a directory.

```
Files.createFile(path);
```

Creating temporary files and directories in system-specific locations.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
```

```
Path newPath = Files.createTempFile(prefix, suffix);
```

```
Path newPath = Files.createTempDirectory(dir, prefix);
```

```
Path newPath = Files.createTempDirectory(prefix);
```

## java.nio.file.Files 7

---

- static Path createFile(Path path, FileAttribute<?>... attrs)
- static Path createDirectory(Path path, FileAttribute<?>... attrs)
- static Path createDirectories(Path path, FileAttribute<?>... attrs)  
creates a file or directory. The createDirectories method creates any intermediate directories as well.
- static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)
- static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)
- static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
- static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)  
creates a temporary file or directory, in a location suitable for temporary files or in the given parent directory. Returns the path to the created file or directory.

# Copying, Moving and Deleting Files

Copy file from one location to another.

```
Files.copy(fromPath, toPath);
```

Move/rename

```
Files.move(fromPath, toPath);
```

Copy/move with target overwrite:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,  
          StandardCopyOption.COPY_ATTRIBUTES);
```

Atomic move (move correctly or don't remove sources):

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

Copy files from InputStream to path and from  
Path to OutputStream

```
Files.copy(inputStream, toPath);  
Files.copy(fromPath, outputStream);
```

Delete file

```
Files.delete(path);
```

## java.nio.file.Files 7

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)  
copies or moves from to the given target location and returns to.
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)  
copies from an input stream to a file, or from a file to an output stream, returning  
the number of bytes copied.
- static void delete(Path path)
- static boolean deleteIfExists(Path path)  
deletes the given file or empty directory. The first method throws an exception if  
the file or directory doesn't exist. The second method returns false in that case.

# Getting File Attributes

All FS attributes are encapsulated in the BasicFileAttributes and PosixFileAttributes

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

```
static Set<PosixFilePermission> getPosixFilePermissions(Path path, LinkOption... options)
    Returns a file's POSIX file permissions.  
static Path setPosixFilePermissions(Path path, Set<PosixFilePermission> perms)
    Sets a file's POSIX permissions.
```

## Method Summary

| All Methods | Instance Methods                                                                                   | Abstract Methods |
|-------------|----------------------------------------------------------------------------------------------------|------------------|
|             | Modifier and Type                                                                                  |                  |
|             | FileTime                                                                                           |                  |
|             | Object                                                                                             |                  |
|             | boolean                                                                                            |                  |
|             | FileTime                                                                                           |                  |
|             | FileTime                                                                                           |                  |
|             | long                                                                                               |                  |
|             | Method and Description                                                                             |                  |
|             | creationTime()                                                                                     |                  |
|             | Returns the creation time.                                                                         |                  |
|             | fileKey()                                                                                          |                  |
|             | Returns an object that uniquely identifies the given file, or null if a file key is not available. |                  |
|             | isDirectory()                                                                                      |                  |
|             | Tells whether the file is a directory.                                                             |                  |
|             | isOther()                                                                                          |                  |
|             | Tells whether the file is something other than a regular file, directory, or symbolic link.        |                  |
|             | isRegularFile()                                                                                    |                  |
|             | Tells whether the file is a regular file with opaque content.                                      |                  |
|             | isSymbolicLink()                                                                                   |                  |
|             | Tells whether the file is a symbolic link.                                                         |                  |
|             | lastAccessTime()                                                                                   |                  |
|             | Returns the time of last access.                                                                   |                  |
|             | lastModifiedTime()                                                                                 |                  |
|             | Returns the time of last modification.                                                             |                  |
|             | size()                                                                                             |                  |
|             | Returns the size of the file (in bytes).                                                           |                  |

## Method Summary

| All Methods | Instance Methods                     | Abstract Methods |
|-------------|--------------------------------------|------------------|
|             | Modifier and Type                    |                  |
|             | GroupPrincipal                       |                  |
|             | UserPrincipal                        |                  |
|             | Set<PosixFilePermission>             |                  |
|             | Method and Description               |                  |
|             | group()                              |                  |
|             | Returns the group owner of the file. |                  |
|             | owner()                              |                  |
|             | Returns the owner of the file.       |                  |
|             | permissions()                        |                  |
|             | Returns the permissions of the file. |                  |

# Directory Traversal/Walk

Directories are also files, so you can read all the entries of a directory and use them.

Filtering files in a directory

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
```

```
    ...
}
```

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
    // Contains all descendants, visited in depth-first order
}
```

Using directory streams.

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
{
    for (Path entry : entries)
        Process entries
}
```

# Memory Mapped I/O

- Instead of writing and reading from disk directly, map the file to a memory location and work on it like its a flat memory.

(Caveat: In reality all files are mapped to a location of memory for faster operation, aka *BufferedReader*).

Steps:

1. Get a FileChannel object of a file (like a file handle).
2. Specify the file area you want to map, using the FileChannel (1) and the *mapping mode* (FileChannel.MapMode.READ\_ONLY,FileChannel.MapMode.READ\_WRITE,FileChannel.MapMode.Private).
3. (2) provides a reference to ByteBuffer class that can be used for reading and writing like a flat file.

Buffers can be both sequential (using the get() and put() method that advance the position) or random-access using the actual location.

# Memory Mapped I/O

Sequential access

```
while (buffer.hasRemaining())
{
    byte b = buffer.get();
    ...
}
```

Random access

```
for (int i = 0; i < buffer.limit(); i++)
{
    byte b = buffer.get(i);
    ...
}
```

```
public static long checksumMappedFile(Path filename) throws IOException
{
    try (FileChannel channel = FileChannel.open(filename))
    {
        CRC32 crc = new CRC32();
        int length = (int) channel.size();
        MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);

        for (int p = 0; p < length; p++)
        {
            int c = buffer.get(p);
            crc.update(c);
        }
        return crc.getValue();
    }
}
```

# Memory Mapped I/O

## java.nio.ByteBuffer 1.4

- `byte get()`

gets a byte from the current position and advances the current position to the next byte.

- `byte get(int index)`

gets a byte from the specified index.

- `ByteBuffer put(byte b)`

puts a byte at the current position and advances the current position to the next byte. Returns a reference to this buffer.

- `ByteBuffer put(int index, byte b)`

puts a byte at the specified index. Returns a reference to this buffer.

- `ByteBuffer get(byte[] destination)`

- `ByteBuffer get(byte[] destination, int offset, int length)`

fills a byte array, or a region of a byte array, with bytes from the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are read, and a `BufferUnderflowException` is thrown. Returns a reference to this buffer.

*Parameters:*    `destination`      The byte array to be filled

`offset`        The offset of the region to be filled

`length`      The length of the region to be filled

- `ByteBuffer put(byte[] source)`

- `ByteBuffer put(byte[] source, int offset, int length)`

puts all bytes from a byte array, or the bytes from a region of a byte array, into the buffer, and advances the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are written, and a `BufferOverflowException` is thrown. Returns a reference to this buffer.

*Parameters:*    `source`        The byte array to be written

`offset`        The offset of the region to be written

`length`      The length of the region to be written

- `Xxx getXxx()`

- `Xxx getXxx(int index)`

- `ByteBuffer putXxx(Xxx value)`

- `ByteBuffer putXxx(int index, Xxx value)`

gets or puts a binary number. `Xxx` is one of `Int`, `Long`, `Short`, `Char`, `Float`, or `Double`.

- `ByteBuffer order(ByteOrder order)`

- `ByteOrder order()`

sets or gets the byte order. The value for `order` is one of the constants `BIG_ENDIAN` or `LITTLE_ENDIAN` of the `ByteOrder` class.

- `static ByteBuffer allocate(int capacity)`

constructs a buffer with the given capacity.

- `static ByteBuffer wrap(byte[] values)`

constructs a buffer that is backed by the given array.

- `CharBuffer asCharBuffer()`

constructs a character buffer that is backed by this buffer. Changes to the character buffer will show up in this buffer, but the character buffer has its own position, limit, and mark.

# *Advanced Programming*

## CSE 201

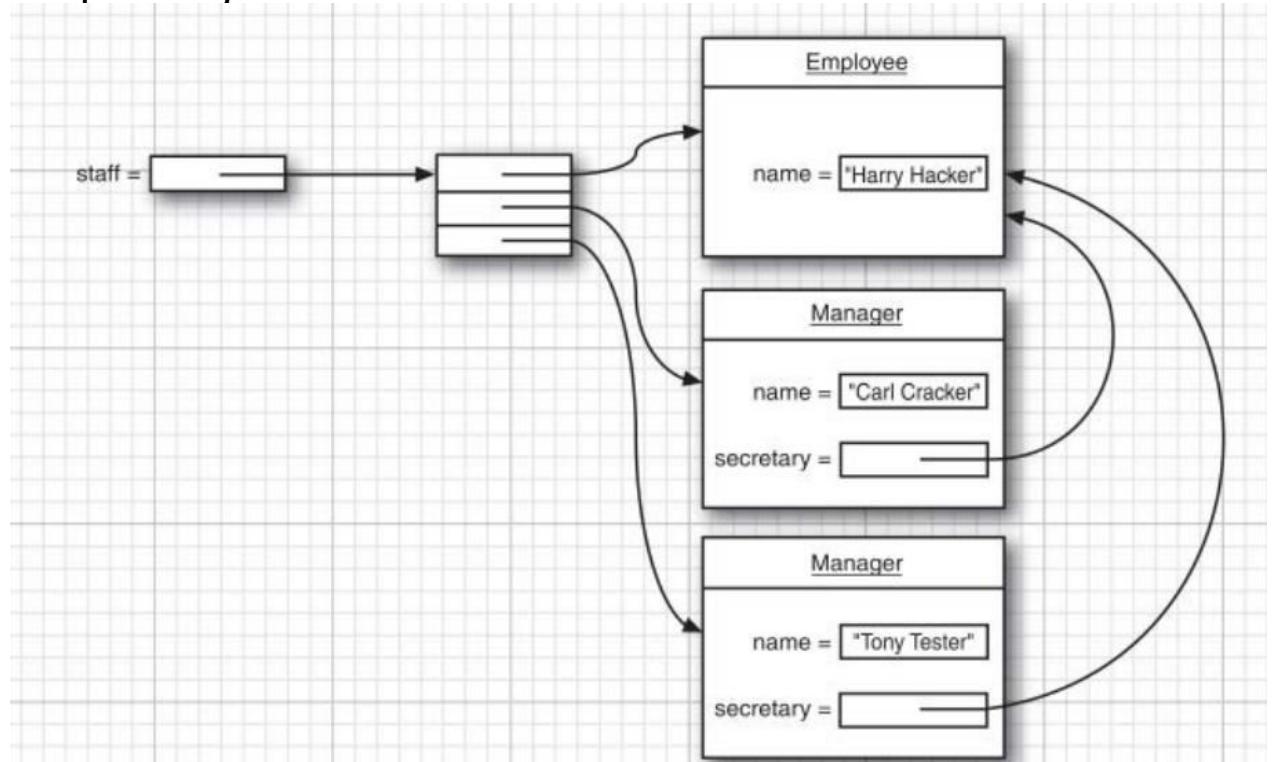
**Instructor: Sambuddho**

(Semester: Monsoon 2024)

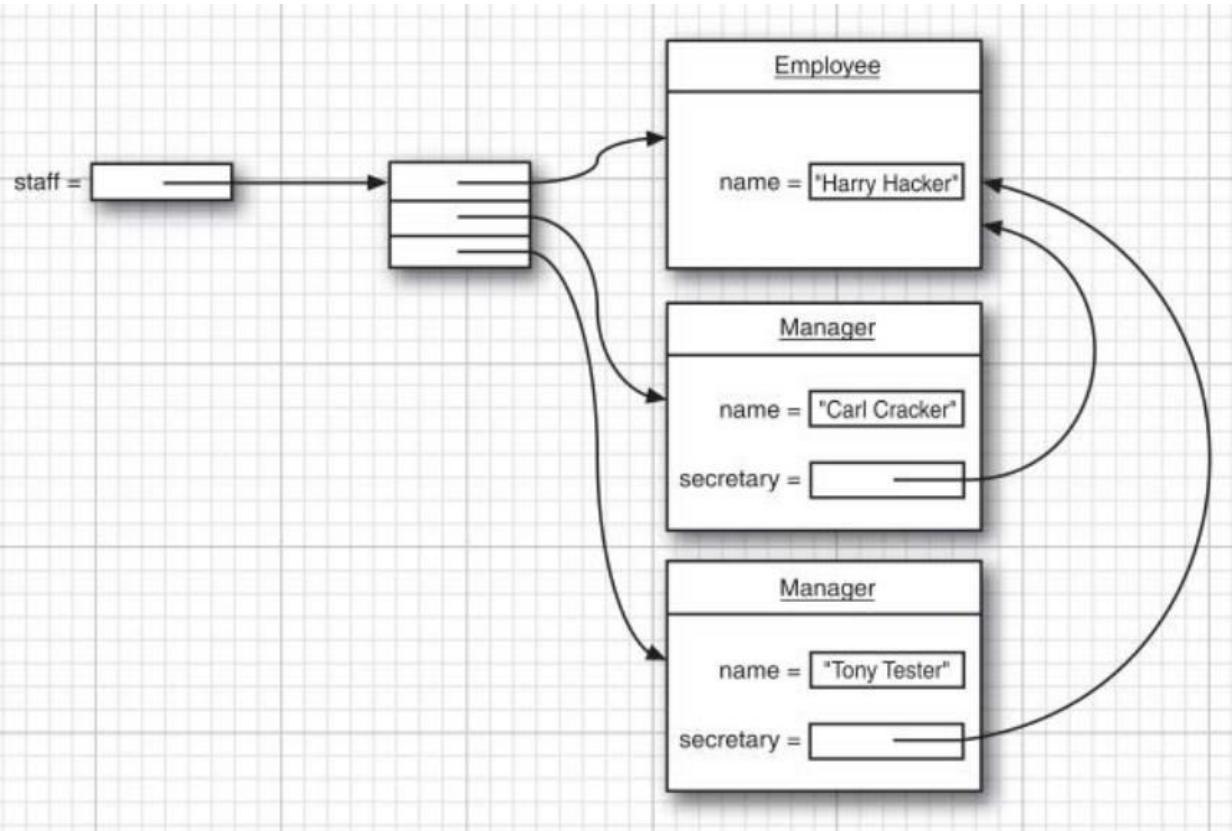
Week 11 – File Serialzability and  
JUnit Testing

# Object I/O Streams and Serialization

- So far we have mostly seen binary and ASCII file read/write.
- Issues with reading and writing complex objects with several in-built references.



# Object I/O Streams and Serialization



```
class Manager extends Employee  
{  
    private Employee secretary;  
    . . .  
}
```

```
harry = new Employee("Harry Hacker", . . .);  
Manager carl = new Manager("Carl Cracker", . . .);  
carl.setSecretary(harry);  
Manager tony = new Manager("Tony Tester", . . .);  
tony.setSecretary(harry);
```

# Reading/Writing Objects

- Writing objects to files.

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

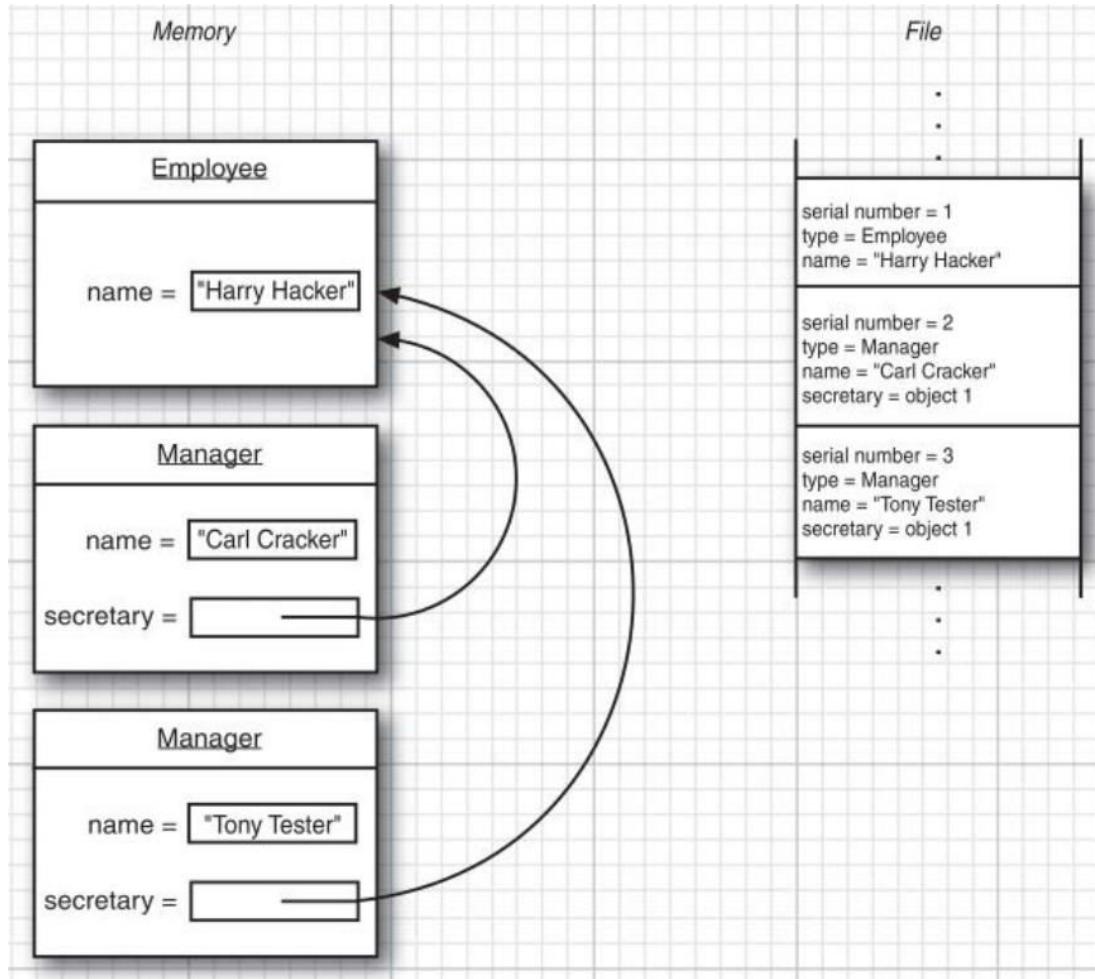
```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

- Reading objects to files.

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));
```

```
Employee e1 = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject();
```

# Reading/Writing Objects



```
class Employee implements Serializable { . . . }
```

# The Transient Types

- Certain types cannot be serialized, e.g. native types.
- Leads to code crashes/bugs.
- Mark such fields as ‘transient’; they are skipped during serialization.
- You need to define the following methods so that default serialization doesn’t take into effect.

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

# The Transient Types

```
public class LabeledPoint implements Serializable
{
    private String label;
    private transient Point2D.Double point;

    private void writeObject(ObjectOutputStream out)
        throws IOException
    {
        out.defaultWriteObject();
        out.writeDouble(point.getX());
        out.writeDouble(point.getY());
    }

    private void readObject(ObjectInputStream in)
        throws IOException
    {
        in.defaultReadObject();
        double x = in.readDouble();
        double y = in.readDouble();
        point = new Point2D.Double(x, y);
    }
}
```

# The Transient Types

- `readObject()` and `writeObject()` only need to be defined for non-serializable fields.
- `readObject()` and `writeObject()` doesn't concern with superclass information.
- Class must implement *Externalizable* interface.
- Serialization: Merely records the class of the object in the input stream.
- Externalizable: `ObjectInputStream` creates an object with no argument and then calls `readExternal()` method.

```
public void readExternal(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
    throws IOException;
```

# The Transient Types

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = LocalDate.ofEpochDay(s.readLong());
}

public void writeExternal(ObjectOutput s)
    throws IOException
{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.toEpochDay());
}
```

# Bugs and Testing

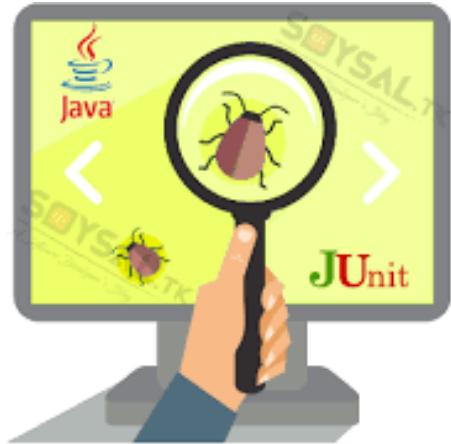
- **Software reliability:** Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTBF (mean time till failure), crash data.
- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- **Testing:** A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation)



# Manual Testing v/s Automated Testing

| Manual Testing                                                                                                                      | Automated Testing                                                                                                                                        |
|-------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Executing a test cases manually without any tool support is known as manual testing                                                 | Taking tool support and executing the test cases by using an automation tool is known as automation testing                                              |
| <b>Time-consuming and tedious</b> – Since test cases are executed by human resources, it is very slow and tedious                   | <b>Fast</b> – Automation runs test cases significantly faster than human resources                                                                       |
| <b>Huge investment in human resources</b> – As test cases need to be executed manually, more testers are required in manual testing | <b>Less investment in human resources</b> – Test cases are executed using automation tools, so less number of testers are required in automation testing |
| <b>Less reliable</b> – Manual testing is less reliable, as it has to account for human errors                                       | <b>More reliable</b> – Automation tests are precise and reliable.                                                                                        |
| <b>Non-programmable</b> – No programming can be done to write sophisticated tests to fetch hidden information                       | <b>Programmable</b> – Testers can program sophisticated tests to bring out hidden information                                                            |

# JUnit: Java Unit Testing Framework



- The Java library **JUnit** helps us to easily perform automated unit testing
- The basic idea:
  - For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail

# Sample JUnit Test

```
/* The class method to be tested */ public class Sum {  
    private int var1, var2;  
    public Sum(int v1, int v2) {var1=v1; var2=v2;} public int  
    sum () {  
        return var1 + var2;  
    }  
}
```

```
/* Junit test class */  
  
import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class MyTest {  
  
    @Test  
    public void testSum() {  
        Sum mySum = new Sum(1, 1); int sum = mySum.sum();  
        assertEquals(2, sum);  
    }  
}
```

```
/* Junit test runner class */  
  
import org.junit.runner.JUnitCore; import org.junit.runner.Result;  
import org.junit.runner.notification.Failure;  
  
public class TestRunner {  
    public static void main(String[] args) {  
        Result result= JUnitCore.runClasses(MyTest.class);  
        for (Failure failure : result.getFailures()) {  
            System.out.println(failure.toString());  
        }  
        System.out.println(result.wasSuccessful());  
    }  
}
```

*static import allows us to access the static members of a class directly without specifying the class name*

```
$ javac -cp ../path_to/junit- 4.10.jar Sum.java MyTest.java  
TestRunner.java
```

```
$ java -cp ../path_to/junit- 4.10.jar TestRunner
```

# JUnit Assertion Methods

|                                           |                                                 |
|-------------------------------------------|-------------------------------------------------|
| assertTrue ( <b>test</b> )                | fails if the boolean test is false              |
| assertFalse ( <b>test</b> )               | fails if the boolean test is true               |
| assertEquals ( <b>expected, actual</b> )  | fails if the values are not equal               |
| assertSame ( <b>expected, actual</b> )    | fails if the values are not the same (by ==)    |
| assertNotSame ( <b>expected, actual</b> ) | fails if the values <i>are</i> the same (by ==) |
| assertNull ( <b>value</b> )               | fails if the given value is <i>not</i> null     |
| assertNotNull ( <b>value</b> )            | fails if the given value is null                |
| fail ()                                   | causes current test to immediately fail         |

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("message", expected, actual)
  - Why is there no pass method?
- Detailed description: <https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html>

# What is Wrong?

```
/* The class method to be tested */ public class Sum {  
    private int var1, var2;  
    public Sum(int v1, int v2) {var1=v1; var2=v2;}  
    public void incr () { var1++; var2++;}  
}
```

```
/* Junit test class */ import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class MyTest {  
  
    @Test  
    public void testIncr() {  
        Sum mySum = new Sum(1, 1);  
        mySum.incr();  
        Sum expected = new Sum(2, 2); assertEquals(expected,  
        mySum);  
    }  
}
```

- We are passing two objects of Sumtype into the testIncr()method where the assertEqualschecks for equality
  - Missing equals()method in Sum !
  - No compilation/runtime error but test will fail

# What's Still Wrong?

```
/* The class method to be tested */ public class Sum {  
    private int var1, var2;  
    public Sum(int v1, int v2) {var1=v1; var2=v2;}  
    public void incr () {  
        var1++; var2++;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if(o!=null && getClass()==o.getClass()) { Sum s =  
            (Sum) o;  
            return ((var1==s.var1)&&(var2==s.var2));  
        }  
        return false;  
    }  
}
```

```
/* Junit test class */  
import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class MyTest {  
  
    @Test  
    public void testIncr() {  
        Sum mySum = new Sum(1, 1);  
        mySum.incr();  
        Sum expected = new Sum(3, 3); assertEquals(expected,  
            mySum); //should fail  
    }  
}
```

*testIncr(MyTest):  
expected:<Sum@2e817b38> but  
was:<Sum@c4437c4>*

- Missing `toString()` method!!

# The Correct Version

```
/* The class method to be tested */ public class Sum {  
    private int var1, var2;  
    public Sum(int v1, int v2) {var1=v1; var2=v2;} public void  
    incr () {  
        var1++; var2++;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if(o!=null && getClass()==o.getClass()) { Sum s =  
            (Sum) o;  
            return ((var1==s.var1)&&(var2==s.var2));  
        }  
        return false;  
    }  
  
    @Override  
    public String toString() {  
        return "("+Integer.toString(var1)+","  
               +Integer.toString(var2)+")";  
    }  
}
```

```
/* Junit test class */ import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
  
public class MyTest { @Test  
    public void testIncr() {  
        Sum mySum = new Sum(1, 1);  
        mySum.incr();  
        Sum expected = new Sum(3, 3); assertEquals(expected,  
        mySum); //should fail  
    }  
}
```

*testIncr(MyTest):*

*expected:<(3,3)> but was:<(2,2)>*

**Note: JUnit tests should be independent to each other as JUnit can run them in any order by using multithreading**

# Tests With a Timeout

```
@Test(timeout = 5000)
```

```
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
```

...

```
@Test(timeout = TIMEOUT)
```

```
public void name() { ... }
```

- Times out / fails after 2000 ms

# Testing for Exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails
  - Use this to test for expected errors

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.get(4);           // should fail
}
```

# Setup and Teardown

**@Before**

```
public void name() { ... }
```

**@After**

```
public void name() { ... }
```

- Methods to run before/after **each test case** method is called

**@BeforeClass**

```
public static void name() { ... }
```

**@AfterClass**

```
public static void name() { ... }
```

- Methods to **run once** before/after the entire test class runs

# JUnit Test Suites

```
/* Junit testcase class-1 */
import org.junit.Test;
import static org.junit.Assert.assertEquals; public class
MyTest1 {
    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1); int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```
/* Junit testcase class-2 */
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class MyTest2 {
    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1); mySum.incr();
        Sum expected = new Sum(2, 2);
        assertEquals(expected, mySum);
    }
}
```

```
/* Junit test suite class */

import org.junit.runner.RunWith; import
org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
    MyTest1.class, MyTest2.class
})

public class TestSuite { }
```

- **Test suite:** One class that runs many JUnit tests
  - An easy way to run all of your app's tests at once
- For this example, the classes Sum and TestRunner are still the same, as earlier. Simply replace "MyTest" in TestRunner with "TestSuite"

# Writing a test case -- Example

```
public class Calculation {  
    //method that returns maximum number  
    public static int findMax(int arr[]){  
        int max=0;  
        for(int i=1;i<arr.length;i++){  
            if(max<arr[i])  
                max=arr[i];  
        }  
        return max;  
    }  
    //method that returns cube of the given number  
    public static int cube(int n){  
        return n*n*n;  
    }  
    //method that returns reverse words  
    public static String reverseWord(String str){  
  
        StringBuilder result=new StringBuilder();  
        StringTokenizer tokenizer=new StringTokenizer(str, " ");  
  
        while(tokenizer.hasMoreTokens()){  
            StringBuilder sb=new StringBuilder();  
            sb.append(tokenizer.nextToken());  
            sb.reverse();  
  
            result.append(sb);  
            result.append(" ");  
        }  
        return result.toString();  
    }  
}
```

# Writing a test case -- Example

```
import static org.junit.Assert.assertEquals;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import com.javatpoint.logic.Calculation;

public class TestCase2 {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        System.out.println("before class");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("before");
    }

    @Test
    public void testFindMax(){
        System.out.println("test case find max");
        assertEquals(4,Calculation.findMax(new int[]{1,3,4,2}));
        assertEquals(-2,Calculation.findMax(new int[]{-12,-3,-4,-2}));
    }
}
```

```
@Test
public void testCube(){
    System.out.println("test case cube");
    assertEquals(27,Calculation.cube(3));
}

@After
public void tearDown() throws Exception {
    System.out.println("after");
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
    System.out.println("after class");
}
```

# Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add usually works, but fails after you call remove
  - make multiple calls; maybe size fails the second time only

# Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add usually works, but fails after you call remove
  - make multiple calls; maybe size fails the second time only

# Trustworthy Tests

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed
- Tests should avoid logic.
  - minimize if/else, loops, switch, etc
  - avoid try/catch
    - If it's supposed to throw, use expected= ... if not, let JUnit catch it

# *Advanced Programming*

## CSE 201

**Instructor: Sambuddho**

(Semester: Monsoon 2024)

Week 11 – Threads and Multithreaded  
Programming

# Overview

- Threads – sub-units of processes that are ``lightweight'', *i.e.* share lot of memory locations. They run within one program.
- OS schedules → JVM → (schedules) → Processes/java program → uses thread library to schedule → threads.
- In most OSes, threads are not very different from processes.
- In java they are different!

# Create a Thread

- 1. Implement run() of *Runnable* interface.

```
Runnable r = () -> { task code };
```

- 2. Construct a Thread object from the Runnable.

```
var t = new Thread(r);
```

- 3. Start the thread.

- Altern `t.start();`

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        task code  
    }  
}
```

```
public interface Runnable  
{  
    void run();  
}
```

compact1, compact2, compact3  
java.lang

## Class **Thread**

java.lang.Object  
java.lang.**Thread**

All Implemented Interfaces:  
Runnable

Direct Known Subclasses:  
ForkJoinWorker**Thread**

# Create a Thread – Example

```
1 package threads;
2
3 /**
4 * @version 1.30 2004-08-01
5 * @author Cay Horstmann
6 */
7 public class ThreadTest
8 {
9     public static final int DELAY = 10;
10    public static final int STEPS = 100;
11    public static final double MAX_AMOUNT = 1000;
12
13    public static void main(String[] args)
14    {
15        var bank = new Bank(4, 100000);
16        Runnable task1 = () ->
17        {
18            try
19            {
20                for (int i = 0; i < STEPS; i++)
21                {
22                    double amount = MAX_AMOUNT * Math.random();
23                    bank.transfer(0, 1, amount);
24                    Thread.sleep((int) (DELAY * Math.random()));
25                }
26            }
27            catch (InterruptedException e)
28            {
29            }
30    };
}
```

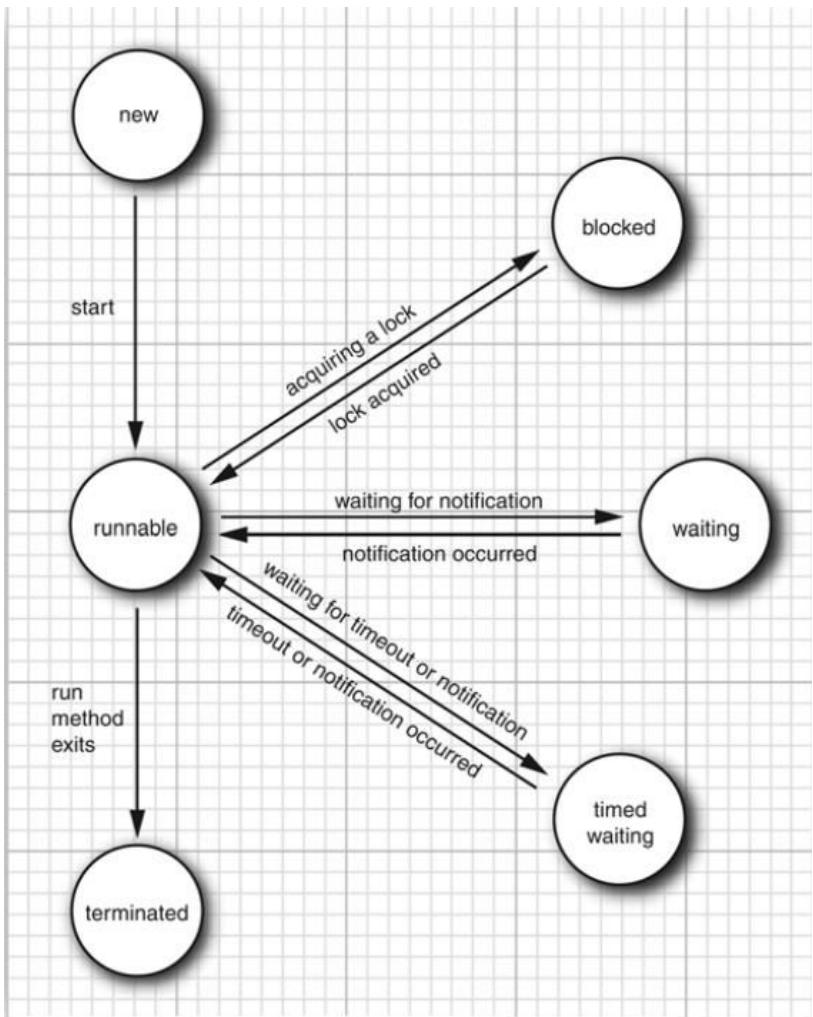
```
31
32    Runnable task2 = () ->
33    {
34        try
35        {
36            for (int i = 0; i < STEPS; i++)
37            {
38                double amount = MAX_AMOUNT * Math.random();
39                bank.transfer(2, 3, amount);
40                Thread.sleep((int) (DELAY * Math.random()));
41            }
42        }
43        catch (InterruptedException e)
44        {
45        }
46    };
47
48    new Thread(task1).start();
49    new Thread(task2).start();
50}
51}
```

---

## java.lang.Thread 1.0

- `Thread(Runnable target)`  
constructs a new thread that calls the `run()` method of the specified target.
- `void start()`  
starts this thread, causing the `run()` method to be called. This method will return immediately. The new thread runs concurrently.
- `void run()`  
calls the `run` method of the associated `Runnable`.
- `static void sleep(long millis)`  
sleeps for the given number of milliseconds.

# Thread States



- You need to periodically `yield()` a thread for others to be scheduled, because of non-preemptive scheduling.

# Thread Termination

- 1. Happens due to two reasons – normal process termination or due to uncaught exception.

`java.lang.Thread 1.0`

---

- `void join()`  
waits for the specified thread to terminate.
- `void join(long millis)`  
waits for the specified thread to die or for the specified number of milliseconds to pass.
- `Thread.State getState() 5`  
gets the state of this thread: one of NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, or TERMINATED.
- `void stop()`  
stops the thread. This method is deprecated.
- `void suspend()`  
suspends this thread's execution. This method is deprecated.
- `void resume()`  
resumes this thread. This method is only valid after `suspend()` has been invoked.  
This method is deprecated.

# Thread Interruption

- - Java threads are non-preemptive.
- - Grab a thread's attention – interruption.
  - Process terminates.
  - Exception.
  - Interrupt occurred.

interrupt

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
    do more work
}
```

```
public void interrupt()
```

- Blocked and sleeping threads cannot be interrupted. InterruptedException occurs.
- Interruption doesn't mean termination – that is system dependent if that happens.
- Only semantic – grab thread's attention!
- Thread can choose reaction to interrupt.

# Thread Interruption – Example of checking if thread was interrupted.

```
Runnable r = () -> {
    try
    {
        ...
        while (!Thread.currentThread().isInterrupted() && more work to do)
        {
            do more work
        }
    }
    catch(InterruptedException e)
    {
        // thread was interrupted during sleep or wait
    }
    finally
    {
        cleanup, if required
    }
    // exiting the run method terminates the thread
};
```

# Thread Interrupt Methods

`java.lang.Thread 1.0`

---

- `void interrupt()`

sends an interrupt request to a thread. The interrupted status of the thread is set to true. If the thread is currently blocked by a call to `sleep`, then an `InterruptedException` is thrown.

- `static boolean interrupted()`

tests whether the *current* thread (that is, the thread that is executing this instruction) has been interrupted. Note that this is a static method. The call has a side effect—it resets the interrupted status of the current thread to false.

- `boolean isInterrupted()`

tests whether a thread has been interrupted. Unlike the static `interrupted` method, this call does not change the interrupted status of the thread.

- `static Thread currentThread()`

returns the `Thread` object representing the currently executing thread.

# Daemon Threads

- Daemonizes a thread – sets it to a background thread and doesn't require joining back to parent thread.

```
t.setDaemon(true);
```

**java.lang.Thread 1.0**

- 
- void setDaemon(boolean isDaemon)

marks this thread as a daemon thread or a user thread. This method must be called before the thread is started.

# Thread Names

- Thread can have names.

```
var t = new Thread(runnable);
t.setName("Web crawler");
```

# Handlers of Uncaught Exceptions

- - `run()` method can die from unchecked exceptions.
- - There is no `catch{}` clause to which the exception can be transferred to.
- - Just before the thread dies the handler for the exception is called.
- - Handler must belong to a class that implements  
`Thread.UncaughtExceptionHandler` interface.
- `void uncaughtException(Thread t, Throwable e);`
- - Install handler using `Thread.setUncaughtExceptionHandler()` or using  
`Thread.setDefaultUncaughtExceptionHandler()`.
- - Both of these could be used for ``graceful degradation.''
-

# Handlers of Uncaught Exceptions

- ThreadGroup class implements the Thread.UncaughtExceptionHandler interface.  
Thread.uncaughtException() takes the following action.
- 1. If thread group has a parent thread then call the uncaughtException of the parent group.
- 2. Else, if the Thread.getDefaultUncaughtException != null then that the default exception handler is called.
- 3. Else if the Throwable argument of uncaughtException() is an instance of ThreadDeath (regular thread termination), then nothing happens.
- 4. Else, the thread name and stack trace is printed on System.err.
-

# Handlers of Uncaught Exceptions

## `java.lang.Thread 1.0`

---

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5`
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler() 5`  
sets or gets the default handler for uncaught exceptions.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5`
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() 5`  
sets or gets the handler for uncaught exceptions. If no handler is installed, the thread group object is the handler.

## `java.lang.Thread.UncaughtExceptionHandler 5`

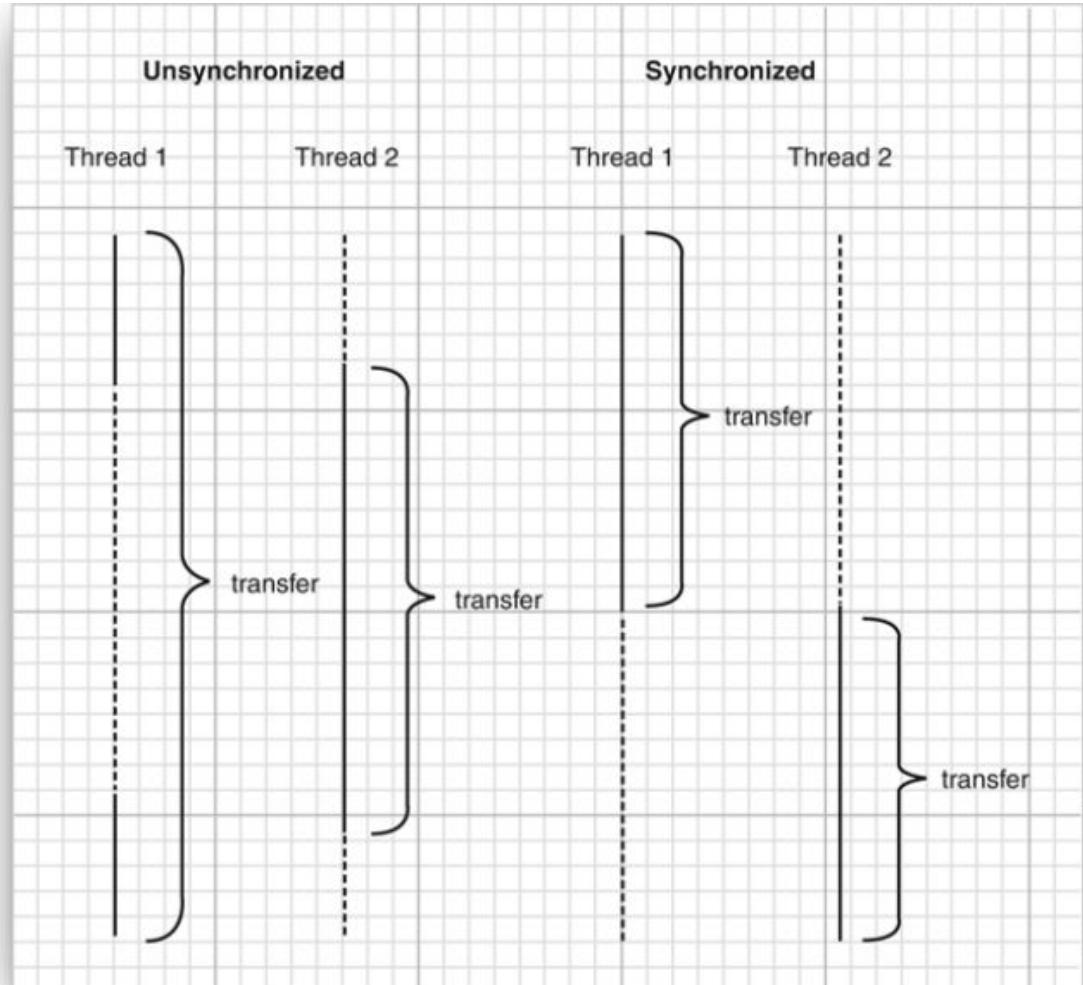
---

- `void uncaughtException(Thread t, Throwable e)`  
defined to log a custom report when a thread is terminated with an uncaught exception.

# Thread Priorities

- - Threads have priorities, determines which thread is chosen to run next by the library, *after a process yeilds or is interrupted.*
- - Default priority is the priority of the parent thread.
- Thread.setPriority(int newPriority)
- MIN\_PRIORITY (1) < NOMR\_PRIORITY (5) < MAX\_PRIORITY(10).
- Priorities are implementation dependent.

# Thread Synchronization – Locks Basics



Synchronized/serialized access versus unsynchronized.

*Basic ideas:*

Race Condition: Two parallel threads access a common object and their order of update leaves the object in a undermined state.

Solution: Serialize access to the common object aka critical section.

# Thread Synchronization – Locks Basics

ReentrantLock class used.

```
Lock mylock = new ReentrantLock();
mylock.lock();
try{
    Critical section
}
finally{
    mylock.unlock();
}
```

# Thread Synchronization – Locks Basics

```
public class Bank
{
    private Lock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
}
```

---

## java.util.concurrent.locks.Lock 5

---

- void lock()

acquires this lock; blocks if the lock is currently owned by another thread.

- void unlock()

releases this lock.

---

## java.util.concurrent.locks.ReentrantLock 5

---

- ReentrantLock()

constructs a reentrant lock that can be used to protect a critical section.

- ReentrantLock(boolean fair)

constructs a lock with the given fairness policy. A fair lock favors the thread that has been waiting for the longest time. However, this fairness guarantee can be a significant drag on performance. Therefore, by default, locks are not required to be fair.

Fair lock: Tries to prevents starvation.

# Thread Synchronization –Conditional Variables

Issues with regular locks:

- You could block on a lock and get into a situation wherein the lock variable is not released until a condition is met.
- Others cannot acquire, and your code doesn't proceed.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // wait
            . .
        }
        // transfer funds
        . .
    }
    finally
    {
        bankLock.unlock();
    }
}
```

# Thread Synchronization -- Conditional Variables

Solution: Conditional variables (class *Condition*).

- If condition is not met, go to ``sleep'', let other threads run.
- When others are done they must *signalAll()* that the condition is met and the *sufficientFunds.await()*;



*sufficientFunds.signalAll();*

```
class Bank
{
    private Condition sufficientFunds;
    ...
    public Bank()
    {
        ...
        sufficientFunds = bankLock.newCondition();
    }
}
```

# Thread Synchronization -- Conditional Variables

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            sufficientFunds.await();
        // transfer funds
        .
        .
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Note: signalAll() doesn't immediately wake up the sleeping threads but makes them runnable.

# Thread Synchronization -- synchronized method

```
public synchronized void method()
{
    method body
}
class Bank
{
    private double[] accounts;

    public synchronized void transfer(int from, int to, int amount)
        throws InterruptedException
    {
        while (accounts[from] < amount)
            wait(); // wait on intrinsic object lock's single condition
        accounts[from] -= amount;
        accounts[to] += amount;
        notifyAll(); // notify all threads waiting on the condition
    }
    public synchronized double getTotalBalance() { . . . }
}
```

The diagram illustrates the mapping between Java's built-in synchronized keyword and the Java Concurrency API. A blue arrow points from the `synchronized` block in the original code to the `lock()` and `unlock()` calls in the expanded version. Another blue arrow points from the `wait()` and `notifyAll()` calls in the original code to the `await()` and `signalAll()` calls in the expanded version.

```
public void method()
{
    this.intrinsicLock.lock();
    try
    {
        method body
    }
    finally { this.intrinsicLock.unlock(); }
}

intrinsicCondition.await();
intrinsicCondition.signalAll();
```

Issues with intrinsic locks and conditions:

### **Thread Synchronization -- synchronized method**

- Cannot be interrupted by a thread trying to acquire a lock (unlike explicit locks).
- Timeout cannot be specified when trying to acquire a lock.
- Single conditions per locks could be inefficient.

## Synchronized Blocks

```
synchronized (obj) // this is the syntax for a synchronized block  
{  
    critical section  
}
```

- Every object has a lock. The synchronized() block acquires the inherent lock of the object and then uses that to control access to a critical section.

## Thread Synchronization -- synchronized method

`java.lang.Object 1.0`

---

- `void notifyAll()`

unblocks the threads that called `wait` on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

- `void notify()`

unblocks one randomly selected thread among the threads that called `wait` on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

- `void wait()`

causes a thread to wait until it is notified. This method can only be called from within a synchronized method or block. It throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

- `void wait(long millis)`

- `void wait(long millis, int nanos)`

causes a thread to wait until it is notified or until the specified amount of time has passed. These methods can only be called from within a synchronized method or block. They throw an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock. The number of nanoseconds may not exceed 1,000,000.

## Synchronized Blocks

```
public class Bank
{
    private double[] accounts;
    private Lock lock = new Object();
```

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println(. . .);
}
```

### - Use of ad-hoc locks

```
...
public void transfer(int from, int to, int amount)
{
    synchronized (lock) // an ad-hoc lock
    {
        accounts[from] -= amount;
        accounts[to] += amount;
    }
    System.out.println(. . .);
}
```

## Final Variables

```
final var accounts = new HashMap<String, Double>();
```

Final variables are like constants and are thread safe to access as once assigned they are not changed.

## Atomics

```
public static AtomicLong nextNumber = new AtomicLong();  
// in some thread. . .  
long id = nextNumber.incrementAndGet();
```

# Thread-Safe Collections

- *Blocking queues* – If multiple threads read from or write to blocking queues then one of them blocks if it is attempting to read from an empty queue or write to a full queue.
- Useful for solving “producer/consumer” like problem scenarios.
- Several variations of queue *add* and *remove* methods – some take a timeout argument.
- Variants: `LinkedBlockingQueue` (unbounded, upper bound optional) , `ArrayBlockingQueue` (constructed with a fixed capacity), `LinkedBlockingDeque` (double-ended version of `LinkedBlockingQueue`), `PriorityBlockingQueue`,`DelayQueue`,`LinkedTransferQueue` (allows producer to wait until consumer is ready).

# Thread-Safe Collections

- Blocking Queue Method

| Method  | Normal Action                        | Action in Special Circumstances                                    |
|---------|--------------------------------------|--------------------------------------------------------------------|
| add     | Adds an element                      | Throws an <code>IllegalStateException</code> if the queue is full  |
| element | Returns the head element             | Throws a <code>NoSuchElementException</code> if the queue is empty |
| offer   | Adds an element and returns true     | Returns false if the queue is full                                 |
| peek    | Returns the head element             | Returns null if the queue is empty                                 |
| poll    | Removes and returns the head element | Returns null if the queue is empty                                 |
| put     | Adds an element                      | Blocks if the queue is full                                        |
| remove  | Removes and returns the head element | Throws a <code>NoSuchElementException</code> if the queue is empty |
| take    | Removes and returns the head element | Blocks if the queue is empty                                       |

# Blocking Queue Example

```
1 package blockingQueue;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import java.util.concurrent.*;
8 import java.util.stream.*;
9
10 /**
11  * @version 1.03 2018-03-17
12  * @author Cay Horstmann
13  */
14 public class BlockingQueueTest
15 {
16     private static final int FILE_QUEUE_SIZE = 10;
17     private static final int SEARCH_THREADS = 100;
18     private static final Path DUMMY = Path.of("");
19     private static BlockingQueue<Path> queue = new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);
20
21     public static void main(String[] args)
22     {
23         try (var in = new Scanner(System.in))
24         {
25             System.out.print("Enter base directory (e.g. /opt/jdk-11-src): ");
26             String directory = in.nextLine();
27             System.out.print("Enter keyword (e.g. volatile): ");
28             String keyword = in.nextLine();
29
30             Runnable enumerator = () -> {
31                 try
32                 {
33                     enumerate(Path.of(directory));
34                     queue.put(DUMMY);
35                 }
36             };
37
38             Runnable searcher = () -> {
39                 try
40                 {
41                     var done = false;
42                     while (!done)
43                     {
44                         Path file = queue.take();
45                         if (file == DUMMY)
46                         {
47                             queue.put(file);
48                             done = true;
49                         }
50                         else search(file, keyword);
51                     }
52                 }
53             };
54
55             new Thread(enumerator).start();
56             for (int i = 1; i <= SEARCH_THREADS; i++) {
57                 Runnable searcher = () -> {
58                     try
59                     {
60                         var done = false;
61                         while (!done)
62                         {
63                             Path file = queue.take();
64                             if (file == DUMMY)
65                             {
66                                 queue.put(file);
67                                 done = true;
68                             }
69                             else search(file, keyword);
70                         }
71                     }
72                 };
73                 new Thread(searcher).start();
74             }
75         }
76         catch (IOException e)
77         {
78             e.printStackTrace();
79         }
80         catch (InterruptedException e)
81         {
82         }
83     }
84 }
```

# Blocking Queue Example

```
        new Thread(searcher).start();
    }
}
}

/**
 * Recursively enumerates all files in a given directory and its subdirectories.
 * See Chapters 1 and 2 of Volume II for the stream and file operations.
 * @param directory the directory in which to start
 */
public static void enumerate(Path directory) throws IOException, InterruptedException
{
    try (Stream<Path> children = Files.list(directory))
    {
        for (Path child : children.collect(Collectors.toList()))
        {
            if (Files.isDirectory(child))
                enumerate(child);
            else
                queue.put(child);
        }
    }
}

/**
 * Searches a file for a given keyword and prints all matching lines.
 * @param file the file to search
 * @param keyword the keyword to search for
 */
public static void search(Path file, String keyword) throws IOException
{
    try (var in = new Scanner(file, StandardCharsets.UTF_8))
    {
        int lineNumber = 0;
        while (in.hasNextLine())
        {
            lineNumber++;
            String line = in.nextLine();
            if (line.contains(keyword))
                System.out.printf("%s:%d:%s%n", file, lineNumber, line);
        }
    }
}
```

# Blocking Queue Example – java.util.concurrent

`java.util.concurrent.ConcurrentLinkedQueue<E>` 5

- `ConcurrentLinkedQueue<E>()`

constructs an unbounded, nonblocking queue that can be safely accessed by multiple threads.

`java.util.concurrent.ConcurrentSkipListSet<E>` 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

constructs a sorted set that can be safely accessed by multiple threads. The constructor requires that the elements implement the Comparable interface.

`java.util.concurrent.ConcurrentHashMap<K, V>` 5

`java.util.concurrent.ConcurrentSkipListMap<K, V>` 6

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

constructs a hash map that can be safely accessed by multiple threads. The default for the initial capacity is 16. If the average load per bucket exceeds the load factor, the table is resized. The default is 0.75. The concurrency level is the estimated number of concurrent writer threads.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

constructs a sorted map that can be safely accessed by multiple threads. The first constructor requires that the keys implement the Comparable interface.

- Thread-safe collections (implementations of maps and queues).

# Atomic Update of Map Entries

- Thread unsafe way:

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // ERROR--might not replace oldValue
```

Safer approach:

```
var map = new ConcurrentHashMap<String, AtomicLong>;
String word = ...
...
AtomicLong oldvalue;
...
map.replace(word,oldvalue,new AtomicLong(newvalue));
```

# Bulk Operations on Concurrent Hash Maps.

Three kinds of operations:

- *search* applies a function to each key/value pairs until a function yields a non-null result and then the search terminates, the search result is returned.
- *reduce* combines all key/value pairs using provided accumulation function.
- *forEach* applies a function to all key/value pairs.

Operations:

- *operationKeys*, *operationValues*, *operation*, *operationEntries*.

```
searchKeys(long threshold, BiFunction<? super K, ? extends U> f)
searchValues(long threshold, BiFunction<? super V, ? extends U> f)
search(long threshold, BiFunction<? super K, ? super V, ? extends U> f)
searchEntries(long threshold, BiFunction<Map.Entry<K, V>, ? extends U> f)
```

# Bulk Operations on Concurrent Hash Maps.

Some examples.

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

For each map entry print key/value pair

```
map.forEach(threshold,  
           (k, v) -> System.out.println(k + " -> " + v));
```

*Transformer* → *consumer* variant. Pass output of transformer to consumer.

```
map.forEach(threshold,  
           (k, v) -> k + " -> " + v, // transformer  
           System.out::println); // consumer
```

# Bulk Operations on Concurrent Hash Maps.

Some examples.

*reduce* operations on key/value pairs.

```
Long sum = map.reduceValues(threshold, Long::sum);
```

*reduce* operations *transformer* → *accumulator* variant.

```
Integer maxlen = map.reduceKeys(threshold,  
    String::length, // transformer  
    Integer::max); // accumulator
```

# Parallel Array Algorithms

Arrays class has number of parallelized operations, e.g.  
Arrays.parallelSort(), Arrays.parallelSetAll() etc.

```
var contents = new String(Files.readAllBytes(  
    Path.of("alice.txt")), StandardCharsets.UTF_8); // read file into string  
String[] words = contents.split("[\\P{L}]+"); // split along nonletters  
Arrays.parallelSort(words);
```

Sort with a comparator method.

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

Compute an operator on all array values.

```
Arrays.parallelSetAll(values, i -> i % 10);  
// fills values with 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```