



**Birla Institute of Technology & Science, Pilani**

**K. K. Birla Goa Campus**

**First Semester 2024-2025**

**Course Handout (Part II)**

In addition to Part-I (General Handout for all courses appended to the time table), this portion gives further specific details regarding the course

**Course ID: CS F213**

**Course Title: Object Oriented Programming**

**Instructor: Neena Goveas ([neena@goa.bits-pilani.ac.in](mailto:neena@goa.bits-pilani.ac.in)) (D259) VOIP 403**

**Sujith Thomas**

### **1. Course Objectives and Scope**

This course provides the students with an understanding of the object-oriented programming paradigm. The topics include Introduction to Object Oriented Programming, Classes and Methods, Encapsulation, Inheritance, Polymorphism, and Multithreaded Programming. The covered topics will be further explained in lab sessions using Java as the programming language. After successfully completing this course, students will have a good understanding of object oriented analysis and design process, and will be able to demonstrate object-oriented concepts in Java programming language.

### **2. Text Books**

T1: The Complete Reference Java J2SE, 5th Edition, Herbert Schildt, Tata McGraw Hill Publishing, 2005

T2: Objects First with Java: A Practical Introduction Using BlueJ, David J. Barnes and Michael Kolling, Pearson Education, 5th Edition, 2012

### **3. Reference Books**

R1: Head First Java, Bert Bates, O'Reilly, 2<sup>nd</sup> Edition, 2005

R2: Core Java Volume I - Fundamentals, Cay Horstmann, Pearson Education, 8<sup>th</sup> Edition 2008

### **4. Course Plan**

No. of Lectures	Topic	Reference
1	Introduction and review	Class Notes + Course Handout
3	Introduction to Object Oriented Programming, Class definition, Object, Principles of OOP, Introduction to Java program syntax, Compiling & execution of Java program	T1 Ch2, T2 Ch1, Class Notes
2	Primitive data types, Type conversion and casting, Arrays, Operators, Control statements, Minor differences between C & Java	T1 Ch3, Ch4, Ch5 T2 Ch2
2	Class fundamentals, Objects, Constructors and Methods, Garbage collection	T1 Ch6, T2 Ch2
3	UML diagrams	R2 Ch4
2	Variables of class and null type, Method overloading, Object as parameters, Argument passing, Access Specifiers	T1 Ch7
2	Static variable and static methods, Accessors and Mutators, Introducing final: Final methods, Final classes, Introducing nested and inner class,	T1 Ch7

2	Revisiting arrays, Exploring string class, Variable length arguments	T1 Ch7
4	Inheritance, Keyword: super, Instance variable hiding, Multilevel hierarchy, Method overriding, Abstract classes, Final with inheritance.	T1 Ch8, T2 Ch8, Ch9
2	Packages, Importing packages, Creating packages, Access protection, Interfaces, Defining and implementing interfaces	T1 Ch9
2	Exception handling fundamentals, Exception types, Try and catch, Nested try statements, Java's built-in exceptions, Keywords: throw, throws, and finally	T1 Ch10, T2 Ch12
4	Collections: Set, List, Map, Iterator	Class Notes
3	Thread model and basics, Creating new threads, The Main thread, Thread synchronization	T1 Ch11
4	File handling in Java, I/O Classes and Interfaces, Stream classes	T1 Ch19
4	Advanced Topics in OOP, UML and Java	Class Notes

## 5. Evaluation Components

Component	Duration	Date	% Weightage	Remarks
Lab	Continuous.		20%	Open Book
Mid test		Oct 4 9:30 -11:00	25%	Closed Book
Group Project	Continuous		20%	Open Book
End-term exam		Dec 4, FN	35%	Closed Book

6. **Office Hours:** Wednesday, 3:00-4:00 PM. Students must do offline discussions only during office hours. Appointments via email for other timeslots are to be taken.
7. **Notices:** All notices concerning this course will be displayed on the News forum of Quanta. Keep an eye on ID/AUGSD notices as well.
8. **Make-up Policy:** Cases approved by the Faculty Incharge of Instruction Division will be granted for a make-up exam. No make-up for the regular lab/lab quiz/ Group project components. Zero marks will be awarded in case of absence or missing component (lab or exam).
9. **Evaluation Policy:** Any attempt of cheating or plagiarism in tests or labs will attract disciplinary committee action.

**Instructor In-charge**  
Prof Neena Goveas

ID Number	Name	Grp No	PC No.
2022A7PS1350G	SHARMA Jr.,GAURANG MADHAV	0	Z1_01
2022B1A70060G	,,ANSHUMAN SRINIVAS PHUKAN	0	Z1_02
2022B3A70110G	,,SHIVAM NABIN AGARWALA	0	Z1_03
2022B3A70920G	DESHWAL,HARSH	0	Z1_04
2022B4A70420G	,,YASHRAAJ JATANIA	0	Z1_05
2022B4A70990G	,,ANISH BANSAL	0	Z1_06
2022B5A70070G	,,SACHIL SANTOSH ABBIMANE	0	Z1_07
2022B5A71060G	,,VIMARSH APURVBHAI SHAH	0	Z1_08
2022B5A71230G	,,NAMAN GUPTA	0	Z1_09
2023A7PS0010G	BHATT,TANUSH BIREN	0	Z1_10
2023A7PS0020G	,,ABHIRAM MANDADI	0	Z1_11
2023A7PS0030G	,,VIJAYANT JOSHI	0	Z1_12
2023A7PS0330G	,,DAKSH JANGRA	0	Z1_13
2023A7PS0340G	,,PRANAV R	0	Z1_14
2023A7PS0350G	,,RISHI JAIN	0	Z1_15
2023A7PS0360G	,,LALIT KISHORE V	0	Z1_16
2023A7PS0370G	,,OJASWI PRATAP SINGH	0	Z1_17
2023A7PS0380G	,,ANKUR KAMBOJ	0	Z1_18
2023A7PS0390G	,,SHASHWAT PATNI	0	Z1_19
2023A7PS0400G	,,GOUTHAM REDDY ARAVA	0	Z1_20
2023A7PS0410G	,,ADITYA KRISHNA JHA	0	Z1_21
2023A7PS0420G	,,PARTH LANGAR	0	Z1_22
2023A7PS0440G	WADHAWAN,ASHMAN	0	Z1_23
2023A7PS0450G	,,RAHUL SHIRIRANG GORHE	0	Z1_24
2023A7PS0460G	,,SOHAM SUNIL KALBURGI	0	Z1_25
2023A7PS0470G	,,AADI PANDEY	0	Z1_26
2023A7PS0480G	,,VEDANT AGARWAL	0	Z1_27
2023A7PS1020G	,,RISHITA JAIN	0	Z1_28
2021A7PS1211G	A,ADITHI	1	Z2_01
2022B3A70031G	,,ANIKET SONAWANE	1	Z2_02
2022B3A70501G	,,ARNAV NIHIL JAIN	1	Z2_03
2022B4A70771G	,,ARJUN PARDAL	1	Z2_04
2022B5A70151G	,,ANAND JAT	1	Z2_05
2022B5A70941G	,,ANAND SHANKAR HARIHARAN	1	Z2_06
2022B5A71241G	,,RISHABH SAHU	1	Z2_07
2023A7PS0021G	,,SANIA KOTHARI	1	Z2_08
2023A7PS0331G	GUPTA,RACHIT	1	Z2_09
2023A7PS0341G	,,RAHUL KUMAR YADAV	1	Z2_10
2023A7PS0351G	MISHRA,DEV RISHI	1	Z2_11
2023A7PS0361G	,,PRIYANSHU TALWAR	1	Z2_12
2023A7PS0371G	,,MITTAL,HARSHVARDHAN	1	Z2_13
2023A7PS0381G	DEDHIA,SOUMYA DIPESH	1	Z2_14

2023A7PS0391G	,RAHUL LAKKIMSETTY	1	Z2_15
2023A7PS0401G	BORKAR,NILAY SADANAND	1	Z2_16
2023A7PS0411G	,VISHESH AGRAWAL	1	Z2_17
2023A7PS0421G	GUPTA,VANAD	1	Z2_18
2023A7PS0431G	,PRAKHAR RANJAN	1	Z2_19
2023A7PS0441G	SHARMA,VEDANSH	1	Z2_20
2023A7PS0451G	,HARSHIT NEGI	1	Z2_21
2023A7PS0461G	,ADITYA NAMDEO	1	Z2_22
2023A7PS0471G	,ARYAN CHRIS LOPEZ	1	Z2_23
2023A7PS0481G	,VIVEK RISHI PANCHAGNULA	1	Z2_24
2023A7PS1021G	.RAOOT,SARTHAK SHASHIKANT	1	Z2_25
2022B1A70092G	,AYUSH AGARWAL	2	Z1_29
2022B1A70332G	,SIMRAN CHRISTY SERRAO	2	Z1_30
2022B1A71372G	,ASHMITA DUTTA	2	Z1_31
2022B3A70222G	,CHINMAYA SREERAM	2	Z1_32
2022B3A70852G	,GAURAV SRINIVAS	2	Z1_33
2022B3A70972G	,VANSHAJ BHUDOLIA	2	Z1_34
2022B3A71162G	,DEV CHAUDHARI	2	Z1_35
2022B3A71382G	,AMISH K SINGHAL	2	Z1_36
2022B4A70922G	,SAVYA MISHRA	2	Z1_37
2022B4A70992G	,PRAKHAR PRADHAN	2	Z1_38
2022B4A71542G	,SAAMIR AFRAAZ	2	Z1_39
2023A7PS0012G	,KRISH SANJAY SINGHVI	2	Z1_40
2023A7PS0332G	,ADITYA NITIN JAGTAP	2	Z1_41
2023A7PS0342G	,ESHA DHRITHI KOTHURI	2	Z1_42
2023A7PS0352G	RAMRAJ,VISHRUT	2	Z1_43
2023A7PS0362G	,ARYAN KUMAR SINGH	2	Z1_44
2023A7PS0372G	,RAJSHRI RAJKISHORE SINGH	2	Z1_45
2023A7PS0382G	,ASHMIT AGARWAL	2	Z1_46
2023A7PS0392G	,MANIT TANWAR	2	Z1_47
2023A7PS0402G	,MAYANK JOSHI	2	Z1_48
2023A7PS0412G	,GURANURAG SINGH TUNG	2	Z1_49
2023A7PS0422G	,AYUSH HASMUKHBHAI KAKADIYA	2	Z1_50
2023A7PS0432G	,KANAV GARG	2	Z1_51
2023A7PS0442G	,AASHMAN ANAND PATWARDHAN	2	Z1_52
2023A7PS0452G	,PAWANI PURWAR	2	Z1_53
2023A7PS0462G	,RAGHAV MAHESHWARI	2	Z1_54
2023A7PS0472G	,NAMAN KUSHWAH	2	Z1_55
2023A7PS0482G	,PRANSHUL ARORA	2	Z1_56
2023A7PS1022G	,RIDDHIMA MAZUMDER	2	Z1_57
2021B3A71613G	,M S ASWIN	3	Z1_58
2022B1A71033G	,MAYUKH SAHA	3	Z1_59
2022B2A71383G	,JAY MAHESHWARI	3	Z1_60

2022B3A70233G	,VIDHI BHUTANI	3	Z1_61
2022B3A70283G	NETO,DWAYNE PEREIRA	3	Z1_62
2022B3A70313G	,AYUSH SUBHASH JOSHI	3	Z1_63
2022B3A70343G	NAIDU,KONDALA PADMANABHAM AIS	3	Z1_64
2022B3A70373G	,VADDADI SURYA SAMEER DATTA	3	Z1_65
2022B3A70393G	,DEBANJANAA KUNDU	3	Z1_66
2022B3A70593G	,SWAYAM SHAH	3	Z1_67
2022B4A70073G	,VEDANT VASUDEV KAMATH	3	Z1_68
2022B5A70423G	,VINAYAK VISHWANATHAN	3	Z1_69
2023A7PS0023G	,BHARGAVA VENKATA NIMISHAKAVI	3	Z1_70
2023A7PS0333G	,MANAN HARSHAD NANDHA	3	Z1_71
2023A7PS0343G	,SOHAM DAMBALKAR	3	Z1_72
2023A7PS0353G	R,DARSH	3	Z1_73
2023A7PS0363G	SODHI,TEJAS SINGH	3	Z1_74
2023A7PS0373G	,ARNAV BAJAJ	3	Z1_75
2023A7PS0383G	,AENK SINGHAL	3	Z1_76
2023A7PS0393G	,RIG MAHAJAN	3	Z1_77
2023A7PS0403G	,ARNAV RUSTAGI	3	Z1_78
2023A7PS0413G	,VIYOM GUPTA	3	Z1_79
2023A7PS0423G	,DIVYAM GUPTA	3	Z1_80
2023A7PS0433G	BETHMANGALKAR,ARYAN	3	Z1_81
2023A7PS0443G	,ASHMIT RANA	3	Z1_82
2023A7PS0453G	,VARUN SETH	3	Z1_83
2023A7PS0463G	MENE,ADITYA PRASAD	3	Z1_84
2023A7PS0473G	,ADITYA NAGARSEKAR	3	Z1_85
2023A7PS0483G	,NAVNEET KUMAR SINGH	3	Z1_86
2023A7PS1013G	,DIPESH	3	Z1_87
2023A7PS1023G	,SOHAN MUTRA	3	Z1_88
2023A7PS1153G	,RIA ARORA	3	Z1_89
2022B1A70384G	,DEVANSH MEHTA	4	Z1_90
2022B1A71064G	,PRANAD PRANSHU	4	Z1_91
2022B1A71374G	,ASHMIT GUPTA	4	Z1_92
2022B3A70114G	,ANIKET KINI	4	Z1_93
2022B3A70204G	SINGHAL,ARYAN	4	Z1_94
2022B3A70634G	,APOORV DUBEY	4	Z1_95
2022B3A70674G	,ANUSHKA DATTATRAY PANDIT	4	Z1_96
2022B3A70784G	,MANAS CHOUDHARY	4	Z1_97
2022B3A70994G	,RAGHAV HARSH WAHAL	4	Z1_98
2022B3A71204G	DESHPANDE Jr.,PRANAV J	4	Z1_99
2022B4A70534G	,NAMAN KAUSHIK SHAH	4	Z1_100
2022B5A70594G	,TANISH HARISH SHETTY	4	Z1_101
2022B5A71414G	SUGLA,MANBHAV	4	Z1_102
2023A7PS0014G	,NEHA MAHESH	4	Z1_103

2023A7PS0324G	,ISHANYA SHARMA	4	Z1_104
2023A7PS0334G	,AARYA ANIL DANDAGAVAL	4	Z1_105
2023A7PS0344G	,SKANDHA PRASANNA KUNTAGOD	4	Z1_107
2023A7PS0354G	,RITIK DWIVEDI	4	Z1_108
2023A7PS0364G	,ARSHKHAN PATHAN	4	Z1_109
2023A7PS0384G	RAI,VIGHNESH KUMAR	4	Z1_110
2023A7PS0394G	,ANSHUL RAKESH	4	Z1_111
2023A7PS0404G	,JAYANT CHOUDHARY	4	Z1_112
2023A7PS0414G	,SUDHANSU SUDHAKAR KULKARNI	4	Z1_113
2023A7PS0424G	,SAATHVIK KRISHNA DONNIPADU	4	Z1_114
2023A7PS0434G	LAHOTI,PRATHAM PRAKASH	4	Z1_115
2023A7PS0444G	,KESHAV KAVRA	4	Z1_116
2023A7PS0454G	,SATVIK AGRAWAL	4	Z1_117
2023A7PS0464G	,VARUN BHATT	4	Z1_118
2023A7PS0474G	,DIVYANSHU AGARWAL	4	Z1_119
2023A7PS0484G	.SHARMA,RAHUL	4	Z1_120
2023A7PS1014G	,ISHAAN SEMWAL	4	Z1_121
2023A7PS1024G	,ESHAN SAIKIA	4	Z1_122
2022B2A71415G	,MAYANK	5	Z2_26
2022B2A71505G	,VED ARYAN	5	Z2_27
2022B3A70045G	,DHRUV AHUJA	5	Z2_28
2022B3A70055G	,DEVNEEL ANOOP DAGA	5	Z2_29
2022B3A70095G	SINGH KOCHHAR,ARSHDEEP SINGH AK	5	Z2_30
2022B3A70385G	,KSHITIJ VERMA	5	Z2_31
2022B3A70465G	,HARSHMAN SINGH CHADHA	5	Z2_32
2022B4A70075G	JAWALE,ANSHUL	5	Z2_33
2022B4A71015G	,ANANYA VASHISTHA	5	Z2_34
2022B5A70685G	,KANAD CHAKRADHAR SASWADE	5	Z2_35
2023A7PS0045G	,RIYA RAVI GANDHI	5	Z2_36
2023A7PS0325G	,VARUN CHIRUNAGULA	5	Z2_37
2023A7PS0335G	,HARDIK TYAGI	5	Z2_38
2023A7PS0345G	,JAINIL ALPESH SHAH	5	Z2_39
2023A7PS0355G	,DHRUV GUPTA	5	Z2_40
2023A7PS0365G	JAIN,AARAV	5	Z2_41
2023A7PS0375G	,SIDDHANT KEDIA	5	Z2_42
2023A7PS0385G	,VANSH MEHRA	5	Z2_43
2023A7PS0395G	,ISHAAN SOMANI	5	Z2_44
2023A7PS0405G	,PIYUSH SRIVASTAVA	5	Z2_45
2023A7PS0415G	,RAGAV KRISHNA RAMESH	5	Z2_46
2023A7PS0435G	,SAIRAM REDDY SAMALA	5	Z2_47
2023A7PS0455G	.CHATTERJEE,RITABAN	5	Z2_48
2023A7PS0465G	,DEVARSH KHATRI	5	Z2_49
2023A7PS0475G	,SHOURYA MISHRA	5	Z2_50

2023A7PS0485G	,KRISH JIGNESH SHAH	5	Z2_51
2023A7PS1015G	,RITWIK GUHA	5	Z2_52
2023A7PS1025G	JAIN,ABHINANDAN	5	Z2_53
2023A7PS1155G	AGRAWAL,VARUN	5	Z2_54
2021B5A71196G	,ABHISHEK SENSHARMA	6	Z2_55
2022B1A70216G	MAHAJAN,VEDASHREE PARAG	6	Z2_56
2022B1A70896G	.SACHAN,KARAN	6	Z2_57
2022B3A70256G	,ANUSHKA JAIN	6	Z2_58
2022B3A70296G	,HAAZIQ MOHAMED A	6	Z2_59
2022B3A70366G	JHA,PEEYUSH KUMAR	6	Z2_60
2022B3A70406G	,ANSHUL BHAVIN SHAH	6	Z2_61
2022B3A70536G	JOSE,JOE	6	Z2_62
2022B4A70636G	,ABHISHEK KUMAR	6	Z2_63
2022B4A71206G	UPPAL,SPARSH	6	Z2_64
2022B4A71426G	,SHLOK MANGESH MEHENDALE	6	Z2_65
2022B5A70206G	,SHAASHWAT SAGAR KALE	6	Z2_66
2022B5A70386G	,AADI DAFTARDAR	6	Z2_67
2022B5A71436G	,AMARTYA KUMAR	6	Z2_68
2022B5A71546G	.KOTASTHANE,ARYAN SADANAND	6	Z2_69
2023A7PS0016G	,ANUSHKA REDDY DOODIPALA	6	Z2_70
2023A7PS0326G	,ARYAN SINGH	6	Z2_71
2023A7PS0336G	,DEV RAMAN TANEJA	6	Z2_72
2023A7PS0346G	,ARINJAY JAIN	6	Z2_73
2023A7PS0356G	,JAYANT CHANDWANI	6	Z2_74
2023A7PS0366G	SAWADIA,JANAVI	6	Z2_75
2023A7PS0376G	,JASHAN GUPTA	6	Z2_76
2023A7PS0386G	,PRIYANSH AJMERA	6	Z2_77
2023A7PS0396G	.GOYAL,PALAK	6	Z2_78
2023A7PS0406G	,MEET PARMAR	6	Z2_79
2023A7PS0416G	,BHAVYA BAJAJ	6	Z2_80
2023A7PS0426G	,SAUROJYOTI PAUL	6	Z2_81
2023A7PS0436G	,ANSH SHARMA	6	Z2_82
2023A7PS0446G	,RISHIT LAVANIA	6	Z2_83
2023A7PS0456G	,ARIN SAMEER RODAY	6	Z2_84
2023A7PS0466G	,ARNAV ADIVI	6	Z2_85
2023A7PS0476G	,AALHAD SAWANE	6	Z2_86
2023A7PS0486G	,SHUBHAY GOKHRU	6	Z2_87
2023A7PS1016G	,NIHIRA CHAPALGAONKAR	6	Z2_88
2023A7PS1026G	RAI,K.N.	6	Z2_89
2023A7PS1146G	,M P ASHISH BHAT	6	Z2_90
2022B1A70037G	,ATHARVA MANDHANIYA	7	Z3_01
2022B1A70897G	,ASHMIT ARYA	7	Z3_02
2022B3A70197G	,JAWAHAR RANGANATHAN	7	Z3_03

2022B3A70677G	,,VINEET KUMAR	7	Z3_04
2022B3A70847G	,,TANAY SUDARSHAN	7	Z3_06
2022B3A70857G	,,ARNAV JAIN	7	Z3_07
2022B3A71087G	RASTOGI,ATHARV	7	Z3_08
2022B3A71097G	,,DIVYAM GUPTA	7	Z3_09
2022B4A70377G	,,SAHIL MAHESHWARI	7	Z3_10
2022B4A71427G	,,VANSH SHARMA	7	Z3_11
2022B5A71067G	,,NEERAJ PARAG KULKARNI	7	Z3_12
2023A7PS0017G	,,ANANYA PURWAR	7	Z3_13
2023A7PS0337G	,,TANISHQ GUPTA	7	Z3_14
2023A7PS0347G	,,ADITYA AGARWAL	7	Z3_15
2023A7PS0357G	,,JAYESH SHARMA	7	Z3_16
2023A7PS0367G	,,ANIKETHA HANDE P R	7	Z3_17
2023A7PS0377G	,,ANANYA KHAITAN	7	Z3_18
2023A7PS0387G	,,ABBAS MUFADDAL DUDHIYAWALA	7	Z3_19
2023A7PS0397G	,,KARTIK GARG	7	Z3_20
2023A7PS0417G	,,PRIYANSHU SINGAWAT	7	Z3_21
2023A7PS0427G	DESHMUKH,ANIRBAN	7	Z3_22
2023A7PS0437G	,,VISHESHE NARANG	7	Z3_23
2023A7PS0447G	,,KRIITH SETHI	7	Z3_24
2023A7PS0457G	,,ARNAV RUTVIJ PARIKH	7	Z3_25
2023A7PS0467G	,,HARIS JAMAL	7	Z3_26
2023A7PS0477G	,,SUPRIYA PRIYadarshni	7	Z3_27
2023A7PS1017G	,,AMEY KULKARNI	7	Z3_28
2023A7PS1027G	,,AADITYA HEMANSHU VERNENKER	7	Z3_29
2022B3A70218G	,,SWASTIK PURSHOTTAM DAVE	8	Z3_30
2022B3A70328G	,,SHAAN HASAN	8	Z3_31
2022B3A70418G	,,ADITHI HARISHANKAR	8	Z3_32
2022B3A70628G	,,SHUBHAM SAHOO	8	Z3_33
2022B3A70728G	,,MANAS AVINASH DESAI	8	Z3_34
2022B4A70588G	,,PARTH JAYANANDAN	8	Z3_35
2022B4A71428G	,,SIDDHANT KHUNTETA	8	Z3_36
2023A7PS0328G	,,ARYAMAN SINGH	8	Z3_37
2023A7PS0338G	,,WARREN BONIFACIO TAVARES	8	Z3_38
2023A7PS0348G	,,ARCHIT SAPRA	8	Z3_39
2023A7PS0358G	SHAH,HARSHAVARDHAN	8	Z3_40
2023A7PS0368G	,,SWAYAM LAKHOTIA	8	Z3_41
2023A7PS0378G	,,HRISHANT BHEDA	8	Z3_42
2023A7PS0388G	,,UTKARSH PANDEY	8	Z3_43
2023A7PS0398G	,,PUJIT CHHABRA	8	Z3_44
2023A7PS0408G	,,KRITI JALAN	8	Z3_45
2023A7PS0418G	BHATIA,NILESH	8	Z3_46
2023A7PS0428G	WILLIAMS,DHRUVA MENON	8	Z3_47

2023A7PS0438G	,,ARUNABH SINGH	8	Z3_48
2023A7PS0448G	,,ANANYA VEERARAGHAVAN	8	Z3_49
2023A7PS0458G	,,PRAKHAR BHANDARI	8	Z3_50
2023A7PS0468G	,,KHUSHI MISHRA	8	Z3_51
2023A7PS0478G	,,VRAJ URVISHKUMAR SHAH	8	Z3_52
2023A7PS0488G	,,VAISHNAVI YUVARAJ POTEKAR	8	Z3_53
2023A7PS0568G	,,KATHAN PRAJAPATI	8	Z3_54
2023A7PS1018G	,,AYUSH CHOPRA	8	Z3_55
2023A7PS1028G	,,PARTH JAROLI	8	Z3_56
2022B3A70059G	,,AGARWAL KSHITIJ AGARWAL	9	Z2_91
2022B3A70139G	,,MOKSH C PATIL	9	Z2_92
2022B3A70769G	,,NISHAD NITIN KOTKAR	9	Z2_93
2022B4A70429G	,,ANSHU ANIL OSTWAL	9	Z2_94
2022B4A71429G	,,ISHITA HEMANT ADSUL	9	Z2_95
2022B4A71539G	,,SARTHAK ARORA	9	Z2_96
2022B5A71309G	,,MRIGANKA THAKUR	9	Z2_97
2023A7PS0009G	,,IRIS ELDO	9	Z2_98
2023A7PS0019G	,,PRANAVI PENUMETCHA	9	Z2_100
2023A7PS0029G	,,ADVAIT ANIL KUMAR	9	Z2_101
2023A7PS0329G	,,NIKSHAY BICHALA	9	Z2_102
2023A7PS0339G	GUPTA,ISHAAN	9	Z2_103
2023A7PS0359G	,,NAVYA AGARWAL	9	Z2_104
2023A7PS0369G	,,MOHD AMMAR SHEIKH	9	Z2_105
2023A7PS0379G	MEHTA,ISHAN KAUSTUBH	9	Z2_106
2023A7PS0389G	,,KARTIKEYA SINHA	9	Z2_107
2023A7PS0399G	ANAMALA,RAMA JAYANTH KUMAR RE	9	Z2_108
2023A7PS0409G	,,ADITYA BIRLA	9	Z2_109
2023A7PS0419G	,,HIMANSHU GUPTA	9	Z2_110
2023A7PS0429G	,,NEEL KIRAN NAIK	9	Z2_111
2023A7PS0439G	,,PARTH JHALANI	9	Z2_112
2023A7PS0449G	.RAHUL,RAJ	9	Z2_113
2023A7PS0459G	,,ADVAY ANEESH MATAPURKAR	9	Z2_114
2023A7PS0469G	,,ISHAAN ATTAM	9	Z2_115
2023A7PS0489G	UTHIRAPANDIAN,KAAVIYA	9	Z2_116
2023A7PS0569G	,,ANJALI KAMATH	9	Z2_117
2023A7PS1019G	,,DHRUTHI	9	Z2_118

# Programming using Java

## Java Basics

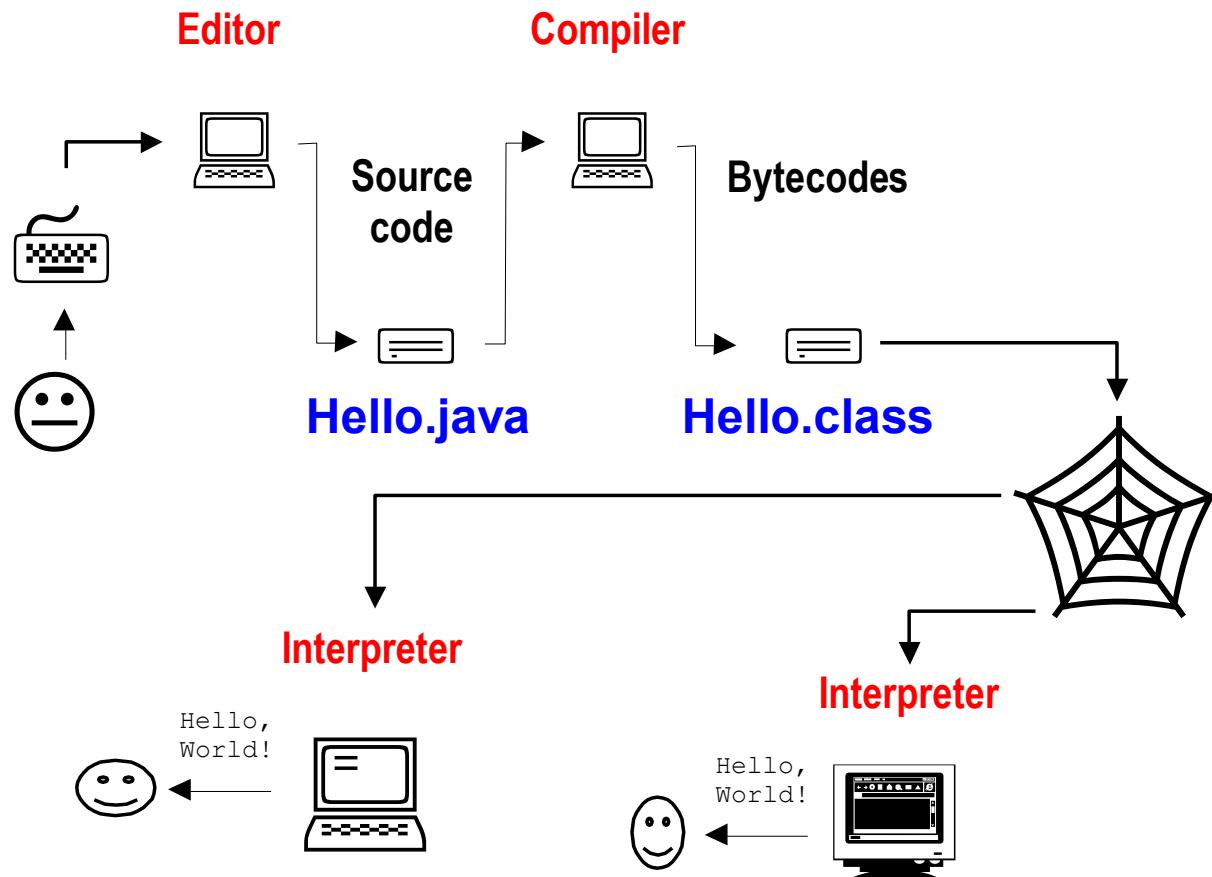
# Java Technology

There is more to Java than the language.

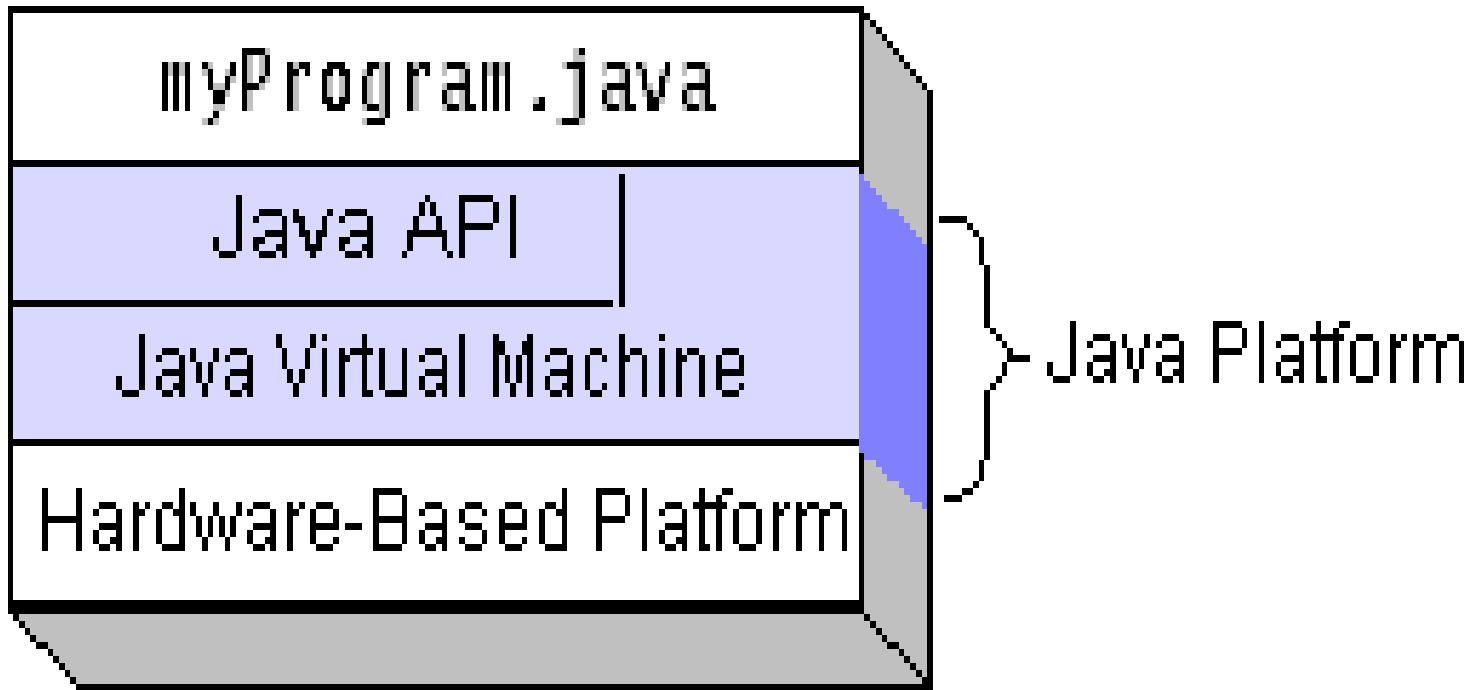
Java Technology consists of:

- 1) Java Programming Language
- 2) Java Virtual Machine (JVM)
- 3) Java Application Programming Interfaces (APIs)

# Java's Compiler + Interpreter



# Java Execution Platform



# Start a Java Application

Java

C

“Hello.java”

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

Class Name

Main method

Main Function

“hello.c”

```
void main() {  
    printf("Hello  
    World\n");  
}
```

> javac Hello.java

Compile

> java Hello

Run

output

Hello World

> cc hello.c -o hello

> hello

# Java Virtual Machine

- The .class files generated by the compiler are not executable binaries
  - so Java combines compilation and interpretation
- Instead, they contain “byte-codes” to be executed by the Java Virtual Machine
- This approach provides platform independence, and greater security

## Lab on 3<sup>rd</sup>: Code Template to be used:

```
import java.util.*;  
public class PatternPrinting {  
    public static void pattern(int n)  
    {  
        //write your code here  
    }  
    // Driver code  
    public static void main(String[] args)  
    {  
        Scanner sc= new Scanner(System.in);  
        System.out.print("Enter number ");  
        int n= sc.nextInt();  
        pattern(n);  
    }  
}
```

# HelloWorld program

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

# Comments are almost like C

```
/* This kind of comment can span  
multiple lines  
*/
```

```
//This kind is to the end of the line
```

```
/**  
 * This kind of comment is a special  
 * 'javadoc' style comment  
 */
```

# Primitive data types are like C

- Main data types are `int`, `double`, `boolean`, `char`  
`byte`, `short`, `long`, `float`
- `boolean` has values `true` and `false`
- Declarations look like C, for example,
  - `double x, y;`
  - `int count = 0;`

# Variables and Assignments

## ■ Variables types

- char      16 bits Unicode character data
- boolean    Boolean Variable
- byte       8 bits signed integer
- short      16 bits signed integer
- int         32 bits signed integer
- long        64 bits signed integer
- float      32 bits signed floating point number
- double     64 bits signed floating point number

# Variables and Assignments

## Variable Declaration

```
type varName;           float x, y, z;
```

## Value assignments

```
varName = value;
```

int num = 100;

long m = 21234234L

double type : .5 0.8 9e-2 -9.3e-5

float type : 1.5e-3f;

# Strings and Characters

- String : sequence of character

String s = “Enter an integer value: ” ;

Char c = ‘A’; ←————— Unicode

- Concatenation Operator ‘+’

String s = “Lincoln said: ” + “\” Four score ago\”” ;

Result :

Lincoln said: “Four score ago”

# Expressions are like C

- Assignment statements mostly look like those in C
  - you can use `=, +=, *=` etc.
- Arithmetic uses the familiar `+ - * / %`
- Java also has `++` and `--`
- Java has boolean operators `&& || !`
- Java has comparisons `< <= == != >= >`
- Java does *not* have pointers or pointer arithmetic

# Arithmetic Operators

## ■ Operators

➤ + - \* / %

➤ += -= \*= /= %=

➤ ++ --

$5 / 2 \rightarrow 2$  Why isn't it 2.5 ?

$5 \% 2 \rightarrow 1$

$4 / 2 \rightarrow 2$

$4 \% 2 \rightarrow 0$

➤ count \* num + 88 / val – 19 % count

➤ j += 6; ➔ j = j + 6;

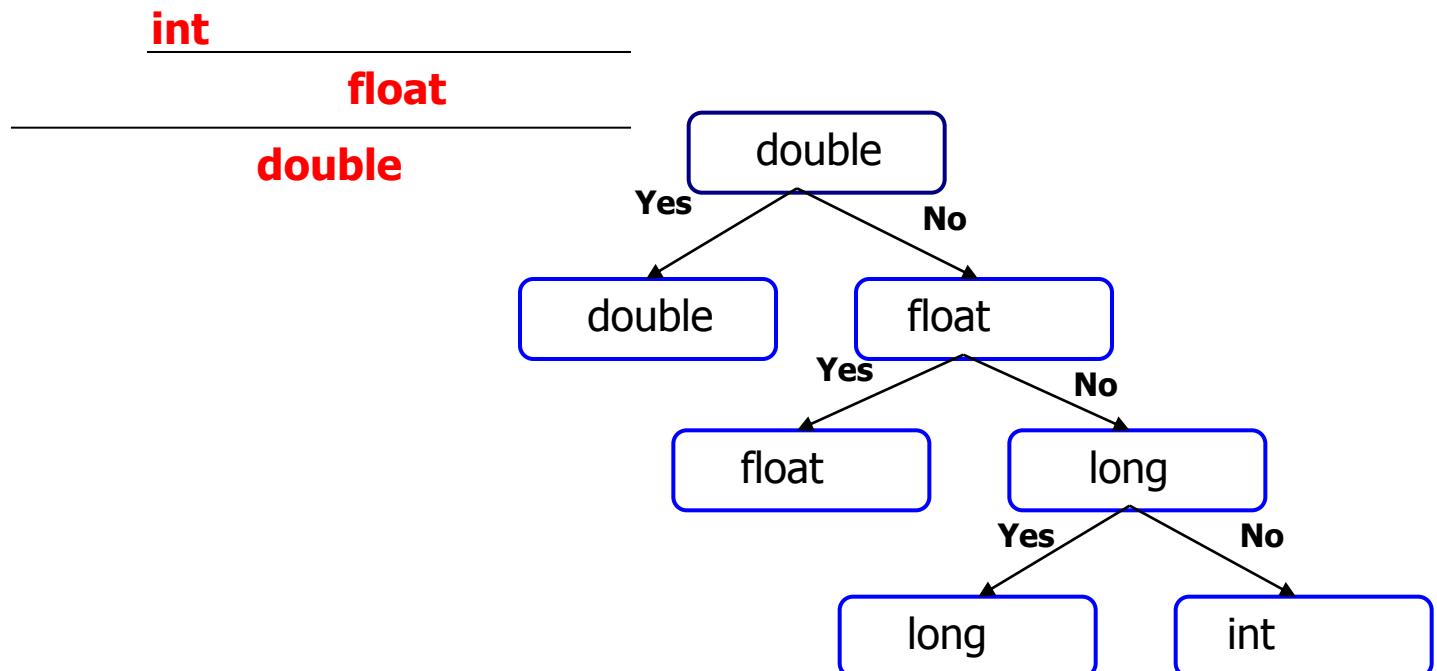
# Operator Precedence

Operator	Association	Precedence
( ) [] . ! ~ ++ -- + - (Data Type) * / % + - << >> >>> < <= > >= instance == != & ^   &&    ?: = += -= *= /= %= &= ^=  = <<= >>= >>>=	Left Assoc. Right Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Left Assoc. Right Assoc. Right Assoc.	(High)  (Low)

# Type Conversions in Expression

```
char ch;    int i;    float f;    double outcome;  
ch = '0';   i = 10;   f = 10.2f;  
outcome = ch * i / f;
```

---



# Type Conversions in Assignment

- Widening Conversion

```
byte b = 127;
```

```
int i;
```

```
i = b;
```

- Wrong Conversion

~~byte b;~~~~int i = 127;~~~~b = i;~~

- Conversion (But data may be lost)

```
byte b;
```

```
int i = 127;
```

```
b = (byte) i;
```

**Type Casting**

# Increment & Decrement Operator

## ■ Operator

- ++, --
- Prefix operator

```
n = 1;  
x = ++n;    // x=2, n=2
```

```
n = 1;  
x = n++;    // x=1, n=2
```

## ➤ Postfix operator

- Cannot be used with expressions, only with variables
- Cannot be applied to the real types

```
(a + b)++ // error
```

# Back-Slash Codes

\b back space

\n new line

\r carriage return

\f form feed

\t tab

\” double quotation

\’ single quotation

\0 null

\\\ back slash

\uxxxx Unicode (x: hexa decimal)

```
class SpecialCharacters {
```

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

```
        System.out.print("\u00a0  
        \u00a1 \u00a2 \u00a3");
```

```
    }
```

```
}
```

# Bitwise Operators

## Operator

- &, |, <<, >>, >>>, ^, ~
- Operand should be integer type
- Precedence

Operator	Precedence
<<    ~ >> >>> & ^ 	(H) ↑ ↓(L)

# Ternary Operator

## Operator

➤ Expr1 ? Expr2 : Expr3 (3 Terms Operator)

```
max = x > y ? x : y ;
```

```
if (x > y) max = x;  
else max = y;
```

```
m = a > b ? (c > a ? c : a) : (c > b ? c : b) ;
```

# Assignment Operators

Expr 1 = Expr 1 op Expr2

Expr1 op= Expr 2

## Operator

- Arithmetic operator : + - \* / %
- Bitwise operator : & | ^ << >> >>>

sum = sum + i ;

sum += i ;

x = x \* y + 1;

x \*= y + 1;



x = x \* (y+1)

# Cast Operators

## ■ Data Type Casting Operator

**(Data Type) Expression**

➤ Cast operator : ( )

(int) 3.75	==>	3
(float) 3	==>	3.0
(float) (1 / 2)	==>	0.0
(float)1/2	==>	0.5

# Control statements are like C

- if ( $x < y$ ) smaller =  $x$ ;
- if ( $x < y$ ){  
    smaller =  $x$ ; sum +=  $x$ ;}  
else { smaller =  $y$ ; sum +=  $y$ ; }
- while ( $x < y$ ){  
     $y = y - x$ ; }
- do {  $y = y - x$ ; }  
while ( $x < y$ )
- for (int  $i = 0$ ;  $i < \text{max}$ ;  $i++$ )sum +=  $i$ ;
- BUT: conditions must be **boolean** !

# Control statements II

```
switch (n + 1) {  
    case 0: m = n - 1; break;  
    case 1: m = n + 1;  
    case 3: m = m * n; break;  
    default: m = -n; break;  
}
```

- Java also introduces the **try** statement, about which more later

# Arrays

## Definition of One Dimensional Array

*type VarName[]*

int ia[];

## Assign Range to One Dimensional Array

*varName = new type[size]*

ia = new int[10];

## Declaration of One Dimensional Array with Range

*type varName[] = new type[size]*

int ia[] = new int[10];

# Java Keywords

## ■ 50 Java Keywords

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient*
class	goto*	protected	try
const*	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	strictfp
assert (New in 1.5)		enum (New in 1.5)	

# Simple Java Program

A class to display a simple message:

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

# Running the Program

Type the program, save as **Hello.java**.

In the command line, type:

```
> ls  
Hello.java  
> javac Hello.java  
> ls  
Hello.java, Hello.class  
> java Hello  
Hello World!
```

# Explaining the Process

- 1) creating a source file - a source file contains text written in the Java programming language, created using any text editor on any system.
- 2) compiling the source file Java compiler (javac) reads the source file and translates its text into instructions that the Java interpreter can understand. These instructions are called bytecode.
- 3) running the compiled program Java interpreter (java) installed takes as input the bytecode file and carries out its instructions by translating them on the fly into instructions that your computer can understand.

# Java Program Explained

**Hello** is the name of the class:

*class Hello{*

main is the method with one parameter args and no results:

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

println is a method in the standard System class:

*System.out.println("First Java program.");*

*}*

*}*

# Main Method

The main method must be present in every Java application:

1) *public static void main(String[] args)* where:

- a) public means that the method can be called by any object
- b) static means that the method is shared by all instances
- c) void means that the method does not return any value

2) When the interpreter executes an application, it starts by calling its main method which in turn invokes other methods in this or other classes.

3) The main method accepts a single argument – a string array, which holds all command-line parameters.

# Getting Started:

## (1) Create the source file:

➤ open a text editor, type in the code which defines a class (*HelloWorldApp*) and then save it in a file (*HelloWorldApp.java*) file and class name are case sensitive and must be matched exactly (except the .java part)

Example Code: HelloWorldApp.java

```
/**  
 * The HelloWorldApp class implements an application  
 * that displays "Hello World!" to the standard output  
 */  
public class HelloWorldApp {  
    public static void main(String[] args) {  
        // Display "Hello World!"  
        System.out.println("Hello World!");  
    }  
}
```



Java is CASE SENSITIVE!

# Getting Started:

## (2) Compile the program:

- compile HelloWorldApp.java by using the following command:

```
javac HelloWorldApp.java
```

it generates a file named HelloWorldApp.class

# Getting Started:

## (3) Run the program:

➤ run the code through:

java HelloWorldApp

➤ Note that the command is `java`, not `javac`,

➤ and you refer to

`HelloWorldApp`, not `HelloWorldApp.java` or

`HelloWorldApp.class`

# Modify:

## (4) Modify the program:

- Declare 2 variables a and b of type int.
- Multiply them
- Print the output as:

a \* b = Value obtained

- You can copy paste any c program you have made and it will work (by taking care of issues discussed in the previous slides)

# Java isn't C!

- In C, almost everything is in functions
- In Java, almost everything is in classes
- There is often only one class per file
- There *must* be only one **public** class per file
- The file name *must* be the same as the name of that public class, but with a **.java** extension

# THE JAVA BUZZWORDS

- Object oriented
- Network Savvy
- Robust
- Secure
- Architecture Neutral
- Portable
- Interpreted
- High Performance
- Multithreaded
- Dynamic

# Java Syntax and Style

## Arrays

# Programmer-Defined Names

- In addition to reserved words, Java uses standard names for library packages and classes:  
`String, Graphics, javax.swing, JApplet,  
JButton, ActionListener, java.awt`
- The programmer gives names to his or her classes, methods, fields, and variables.

# Names

- Syntax: A name can include:
  - upper- and lowercase letters
  - digits
  - underscore characters
- Syntax: A name cannot begin with a digit.
- Style: Names should be descriptive to improve readability.

# Names

- Programmers follow strict style conventions.
- Style: Names of classes begin with an uppercase letter, subsequent words are capitalized:  
`public class FallingCube`
- Style: Names of methods, fields, and variables begin with a lowercase letter, subsequent words are capitalized.

```
private final int delay = 30;  
public void dropCube()
```

# Names

- Method names often sound like verbs:  
`setBackground`, `getText`, `dropCube`, `start`
- Field names often sound like nouns:  
`cube`, `delay`, `button`, `whiteboard`
- Constants sometimes use all caps:  
`PI`, `CUBESIZE`
- It is OK to use standard short names for temporary “throwaway” variables:  
`i`, `k`, `x`, `y`, `str`

# Syntax vs. Style

- Syntax is part of the language. The compiler checks it.
- Style is a convention widely adopted by software professionals.
- The main purpose of style is to improve the **readability of programs**.

# Style

- Arrange code on separate lines; insert blank lines between fragments of code.
- Use comments.
- Indent blocks within braces.

# Style (cont'd)

Before:

```
public boolean  
moveDown(){if  
(cubeY<6*cubeX)  
{cubeY+=yStep;  
return true;}else  
return false;}
```

After:

```
public boolean moveDown()  
{  
    if (cubeY < 6 * cubeX)  
    {  
        cubeY += yStep;  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

Compiles  
fine!

# Style (cont'd)

```
public void fill (char ch)
{
    int rows = grid.length, cols = grid[0].length;
    int r, c;
    for (r = 0; r < rows; r++)
    {
        for (c = 0; c < cols; c++)
        {
            grid[r][c] = ch;
        }
    }
}
```

Add blank lines  
for readability

Add spaces around operators  
and after semicolons

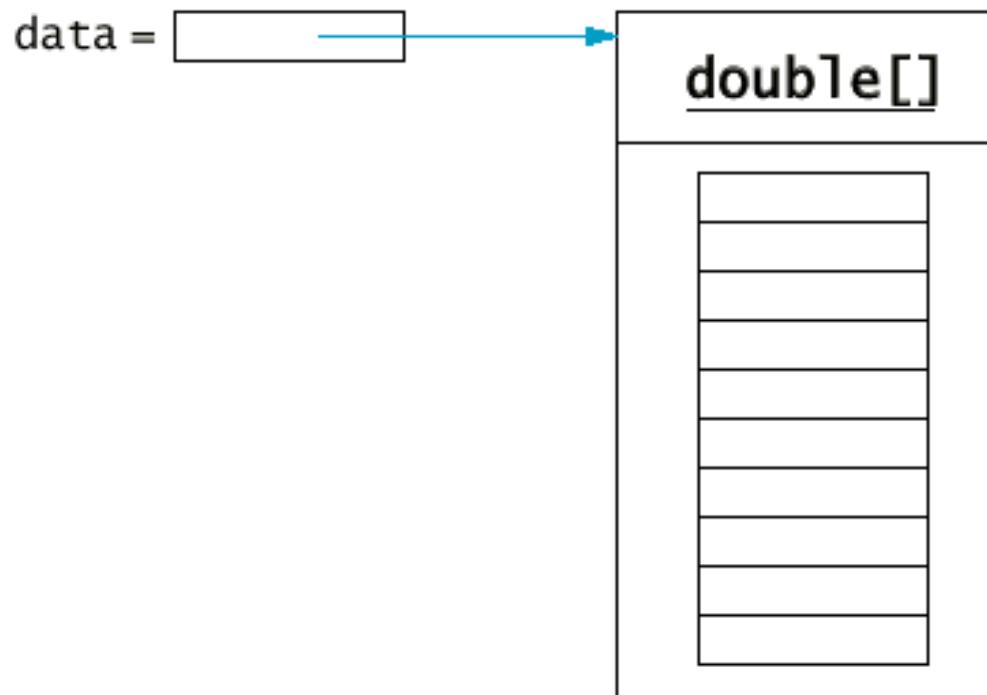
# Java Arrays

# Introduction

- Array is a useful and powerful aggregate data structure presence in modern programming languages
- Arrays allow easy access and manipulation to the values/objects that they store
- Arrays are indexed by a sequence of integers

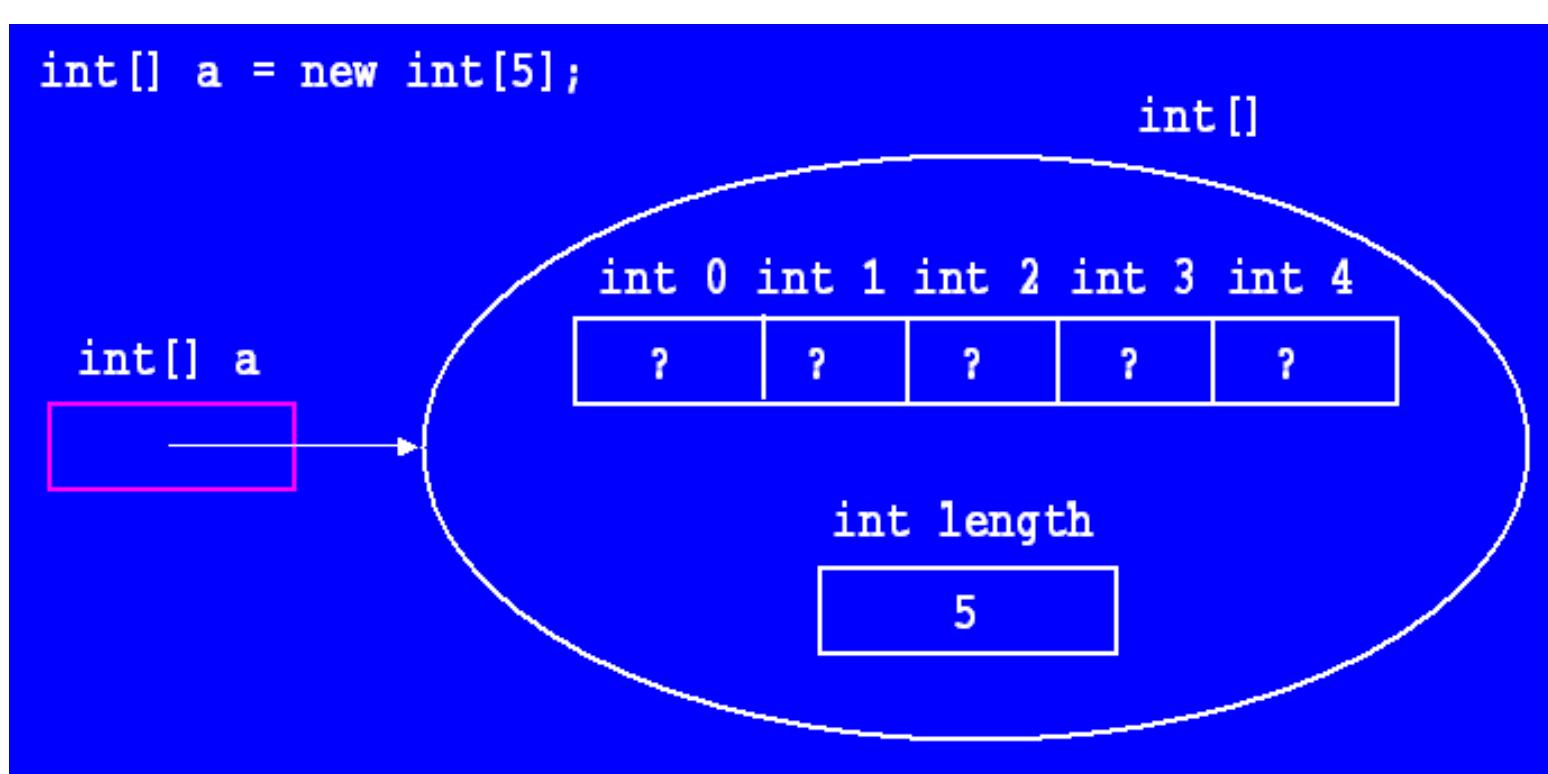
# Arrays

- new is used to construct a new array:  
`new double[10]`
- Store 10 double type variables in an array of doubles  
`double[] data = new double[10];`



# integer Arrays

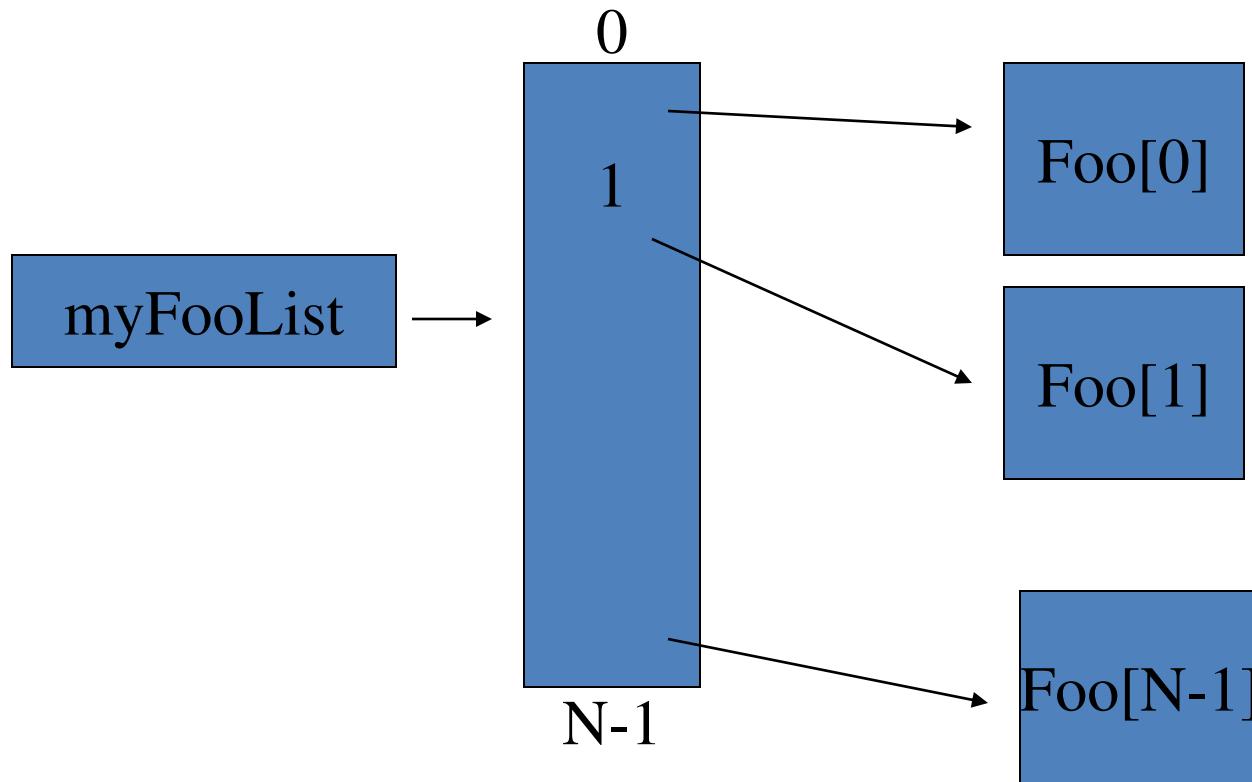
```
int[] a = new int[5];
```



# Array of Object References

```
class Foo() { ....}
```

```
Foo[ ] myFooList = new Foo[N];
```



# Arrays

- *fixed length*
- Element of specific type or references to objects
- [ ] is used to access array elements  
`data[4] = 29.95;`
- Use length **attribute** to get array length.
  - **data.length**
  - (Not a method!)

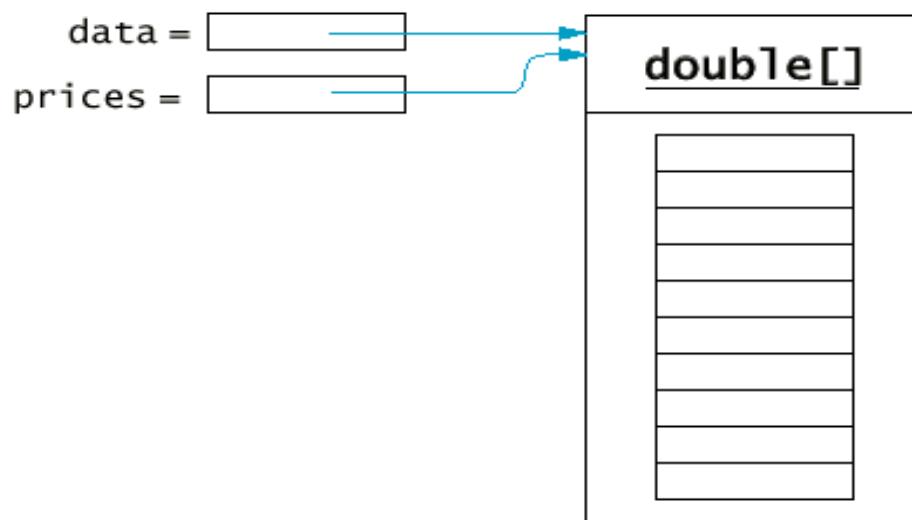
# Array

- homogeneous data structure: each of its members stores the same type (either primitive or reference)
- indices go from 0 to one less than the length of the array
- each array object stores a **public final int length** instance variable that stores the length of the array
- we can access the value stored in this field, in the example above, by writing **a.length**

# Copying Arrays

Copying an array reference yields a second reference to the same array

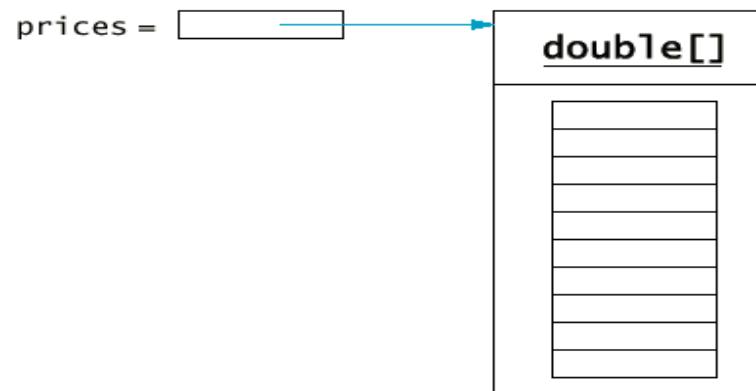
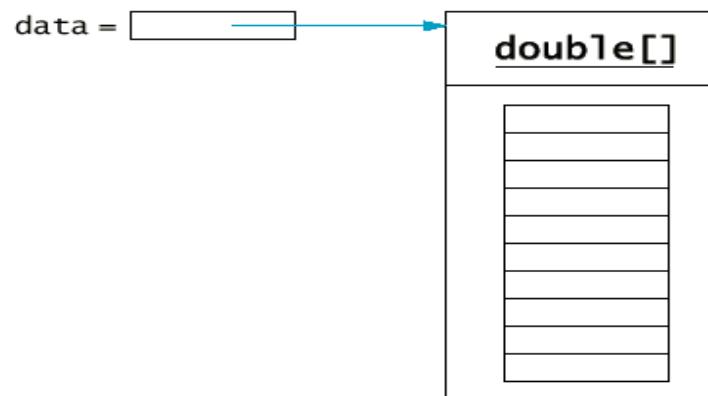
```
double[] data = new double[10];  
// fill array . . .  
double[] prices = data;
```



# Cloning Arrays

- Use `clone` to make true copy

```
double[] prices = (double[])data.clone();
```



# Swapping Array Elements

- Suppose you want to swap two elements in the array, say entries with indices  $i$  and  $j$ . Assuming we are dealing with an array of ints
  - `int temp = A[i]; // save a copy of A[i] in temp`
  - `A [i] = A[j]; // copy the content of A[j] to A[i]`
  - `A[j] = temp; // copy the content of temp to A[j]`
- Note that :  $A[i]= A[j]$  and  $A[j] = A[i]$  do not swap content
- Exercise: Reverse an array using swaps

# Accessing Arrays

- `int[] a = new int[]{4, 2, 0, 1, 3};`
- `system.out.println( a[0] );`
- `if (a[5] == 0) ...some statement`
- if the value computed for the index **is less than 0, or greater than OR EQUAL TO** the length of the array
  - trying to access the member at an illegal index causes Java to throw the
  - `ArrayIndexOutOfBoundsException` which contains a message showing what index was attempted to be accessed

# Programming using Java

## Java Classes and Objects: A Preview

# Objectives:

- Get an introduction to Classes and Objects
- Get a general idea of how a small program is put together

# Classes and Source Files

- A class defines a class of objects.

Convention:  
a class name  
starts with a  
capital letter

Class name:

File name:

SomeClass

SomeClass.java

Ramblecs

Ramblecs.java

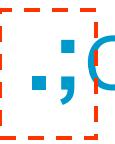
FallingCube

FallingCube.java



Same upper / lower case letters

# Files and Folders

- `javac` automatically looks for classes (.java or .class files) in the current folder, or, if `classpath` is set, in folders listed in the `classpath` string.
- A missing file may be reported as a syntax error when compiling another file.
- If you set `classpath`, include the current folder. It is denoted by a dot. For example:  
  
`.;;C:\javamethods\EasyIO`
- IDE helps take care of the file locations.

```
public class SomeClass
```

```
{
```

- Fields

Attributes / variables that define the object's state. Can hold numbers, characters, strings, other objects. Usually **private**.

- Constructors

Code for constructing a new object and initializing its fields. Usually **public**.

- Methods

```
}
```

Actions that an object can take. Can be **public** or **private**.

*private*: visible only inside this class  
*public*: visible in other classes

```
public class FallingCube
{
    private final int cubeSize;
    private int cubeX, cubeY;      // Cube coordinates
    ...
    private char randomLetter;    // Cube letter

    public FallingCube(int size)
    {
        cubeSize = size;
        ...
    }

    public void start()
    {
        cubeX = 0;
        cubeY = -cubeSize;
        ...
    }
    ...
}
```

Fields

Constructor

The name of a constructor is always the same as the name of the class.

Methods

# Fields

# Fields

You name it!

private (or public) [static] [final]

**datatype name;**

Usually  
private

May be present:  
means the field is  
shared by all objects  
in the class

May be present:  
means the field  
is a constant

int, double, etc., or an  
object: String, JButton,  
FallingCube, Timer

# Field qualifier

- Form of field declaration
  - **[qualifier] DataType fieldNames;**
    - qualifier : public, protected, private, static, final, volatile

```
int anInteger, anotherIntegrer;  
public String usage;  
static long idNum = 0;  
public static final double earthWeight = 5.97e24;
```

# Field

## Access Modifier

- Access Permission Level from other classes
- public, protected, private
- *Default/no declaration*: package

Modifier	Class	Sub-Class	Same Package	All Class
private	O	X	X	X
package	O	X	O	X
protected	O	O	O	X
public	O	O	O	O

# Fields

- May have *primitive data types*:  
**int, char, double, etc.**

```
private int cubeX, cubeY; // cube coordinates
```

...

```
private char randomLetter; // cube letter
```

# Fields

- May be objects of different types:

```
private FallingCube cube;
```

```
private Timer t;
```

```
private static final String letters;
```

# Constructors

# Constructors

- Constructors are like methods for creating objects of a class.
- Most constructors initialize the object's fields.
- Constructors may take parameters.
- A class may have several constructors that differ in the number or types of their parameters.
- All of a class's constructors have the same name as the class.

# Constructors

## Constructor Summary

### [JButton\(\)](#)

Creates a button with no set text or icon.

### [JButton\(Action a\)](#)

Creates a button where properties are taken from the Action supplied.

### [JButton\(Icon icon\)](#)

Creates a button with an icon.

### [JButton\(String text\)](#)

Creates a button with text.

### [JButton\(String text, Icon icon\)](#)

Creates a button with initial text and an icon.

go = new JButton("Go");

# Constructors (cont'd)

- Call them using the `new` operator:

```
cube = new FallingCube(CUBESIZE);
```

...

Calls `FallingCube`'s constructor with  
`CUBESIZE` as the parameter

```
t = new Timer(delay, this)
```

Calls `Timer`'s constructor with `delay` and  
`this` (i.e. this object) as the parameters  
(see Java docs for `javax.swing.Timer`)

# Class constructors

- The purpose of the constructor is to initialize the instance variables
  - Default constructor
  - Constructor overloading
- \*\* forgetting to call a constructor
- \*\* constructor is invoked only when an object is first created. You cannot call the constructor to reset an object.

# Constructors

- If a class has more than one constructor, they are “overloaded” and must have different numbers and/or types of arguments.
- Programmers often provide a “no-args” constructor that takes no arguments.
- If a programmer does not define any constructors, Java provides one default no-args constructor, which allocates memory and sets fields to the default values.

# Constructors (cont'd)

```
public class Fraction
```

```
{
```

```
    private int num, denom;
```

```
    public Fraction ()
```

```
{
```

```
    num = 0;
```

```
    denom = 1;
```

```
}
```

“No-args”  
constructor

```
    public Fraction (int n)
```

```
{
```

```
    num = n;
```

```
    denom = 1;
```

```
}
```



```
    public Fraction (int p, int q)
```

```
{
```

```
    num = p;
```

```
    denom = q;
```

```
    reduce ();
```

```
}
```

```
    public Fraction (Fraction other)
```

```
{
```

```
    num = other.num;
```

```
    denom = other.denom;
```

```
}
```

```
...
```

```
}
```

Copy  
constructor

# Constructors

- Constructors of a class can call each other using the keyword `this` — a good way to avoid duplicating code:

```
public class Fraction
{
    ...
    public Fraction (int n)
    {
        this (n, 1);
    }
    ...
}
```

```
...
→ public Fraction (int p,
    int q)
{
    num = p;
    denom = q;
    reduce ();
}
...
```

# Operator new

- Constructors are invoked using `new`

`Fraction f1 = new Fraction ( );` —| 0 / 1

`Fraction f2 = new Fraction (5);` —————| 5 / 1

`Fraction f3 = new Fraction (4, 6);` —| 2 / 3

`Fraction f4 = new Fraction (f3);` —| 2 / 3

# Operator new (cont'd)

- You must create an object before you can use it; the `new` operator is a way to do it

`Fraction f;`

`f` is set to `null`

`f = new Fraction (2, 3);`

Now `f` refers to a valid object

`f = new Fraction (3, 4);`

Now `f` refers to another object (the old object is “garbage-collected”)

# The this Keyword

## this Keyword

*this.varname*

## Invocation of Constructors

*this(args);*

```
class Point3D {  
    double x;  
    double y;  
    double z;  
  
    Point3D(double x, double y, double z)  
    {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
}
```

```
class ThisKeywordDemo {  
    public static void main(String args[])  
    {  
        Point3D p = new Point3D(1.1, 3.4, -2.8);  
        System.out.println("p.x = " + p.x);  
        System.out.println("p.y = " + p.y);  
        System.out.println("p.z = " + p.z);  
    }  
}
```

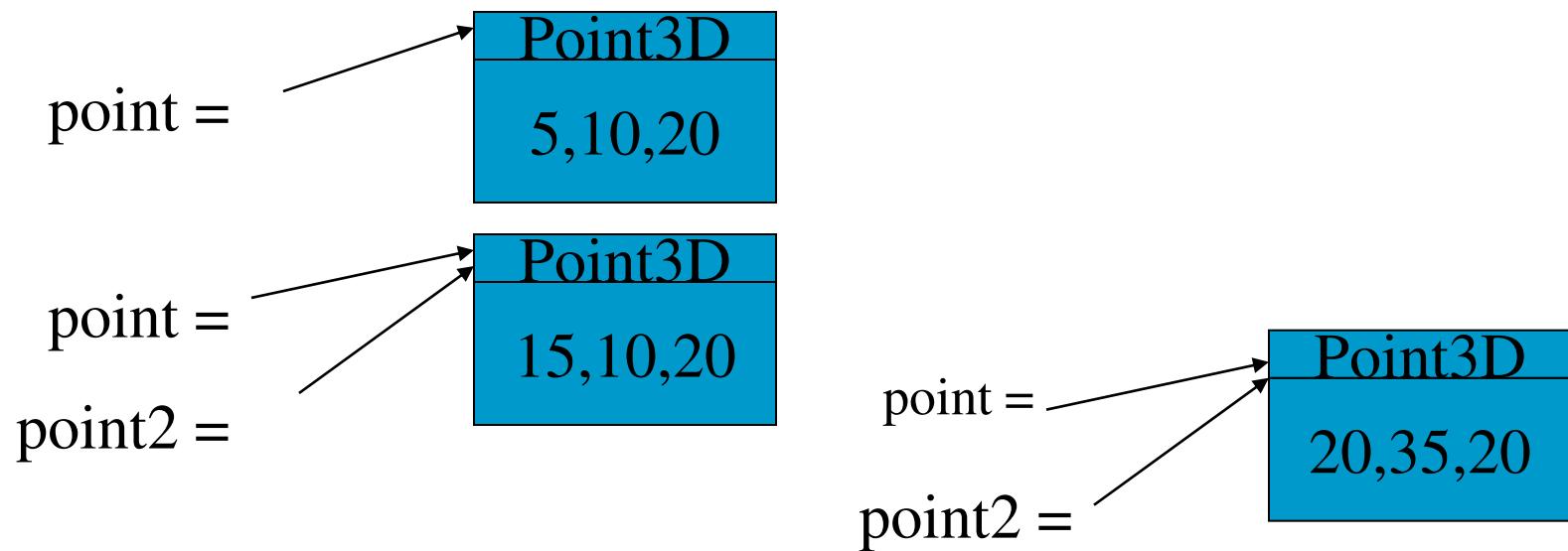
# Calling one constructor from another

```
public class OOPClass
{
    int no_of_stud;
    public OOPClass( int i)
    {
        no_of_stud = i;
    }
    public OOPClass()
    {
        this(50);
    }
}
```

this(...) has to be the first line in the constructor

# Object references

- A variable holds a memory location of an object.



# null references

- The null value is unique in that it can be assigned to a variable of any reference type whatsoever.
- Ex:
  - String middleInitial = null;
  - Car c = null;
- Compare objects
  - if (c == null).....

# Methods

# Methods

- Call them for a particular object:

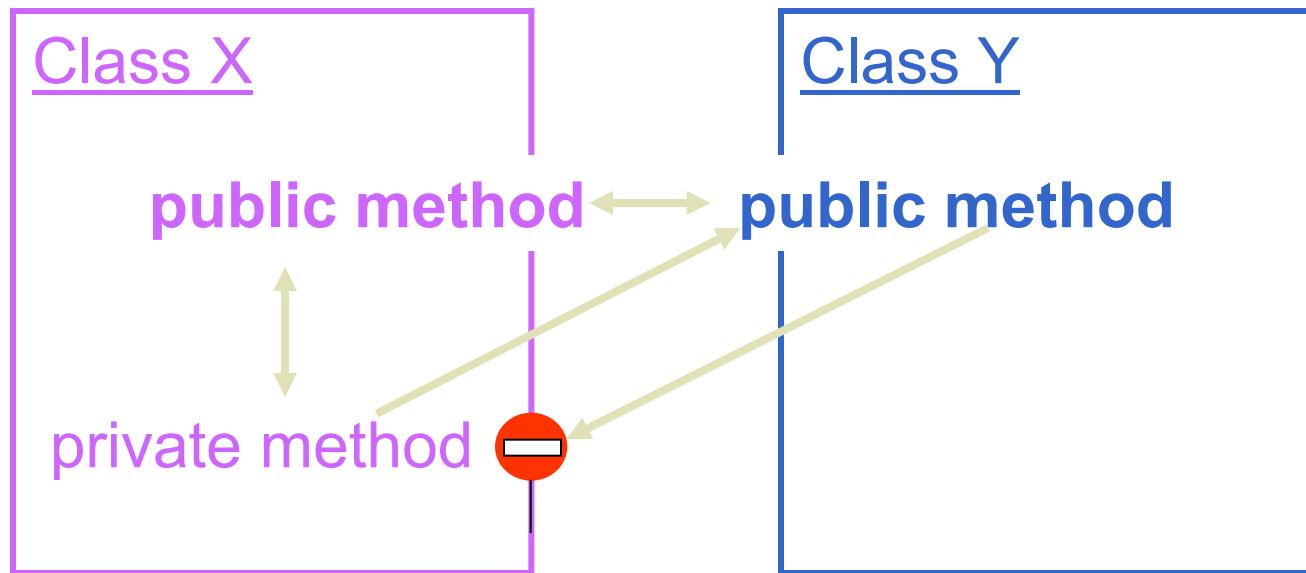
`cube.start();`

But call *static (“class”) methods* for the whole class, not a specific object:

`y = Math.sqrt (x);`

# Methods

- Constructors and methods can call other public and private methods of the same class.
- Constructors and methods can call only accessible methods of another class.



# Methods: Java Style

- Method names start with lowercase letters.
- Method names usually sound like verbs.
- The name of a method that returns the value of a field often starts with `get`:  
`getWidth`, `getX`

The name of a method that sets the value of a field often starts with `set`:  
 `setLocation`, `setText`

# Method

- Method Qualifier
  - Access Modifier
    - Access Permission Level to Method from Other Class
    - Same as that of access modifier in field
  - **static**
    - static method, class method
    - Same role of Global function
    - Use only the static field of correspond class or the static method
    - Can be referred by only class name

```
ClassName.methodName;
```

# Method

## – **final**

- Final method
- Method which cannot be redefined in subclass

## – **synchronized**

- Synchronization method
- Control the threads so that only one thread can always access the target

## – **native**

- To use the implementation written in other programming languages such as C language

# Overloaded Methods

- Methods of the same class that have the same name but different numbers or types of arguments are called ***overloaded methods***.
- Use overloaded methods when they perform similar tasks:

```
public void move (int x, int y) { ... }
```

```
public void move (double x, double y)  
{ ... }
```

```
public void move (Point p) { ... }
```

```
public Fraction add (int n) { ... }
```

```
public Fraction add (Fraction other) {  
... }
```

# Overloaded Methods (cont'd)

- The compiler treats overloaded methods as completely different methods.
- The compiler knows which one to call based on the number and the types of the arguments:

```
public class Circle {
```

```
...  
public void move (int x, int y)  
{ ... }
```

```
public void move (Point p)  
{ ... }
```

```
...
```

```
Circle circle = new Circle(5);
```

```
circle.move (50, 100);
```

```
...  
Point center =
```

```
new Point(50, 100);
```

```
circle.move (center);
```

```
...
```

# Method Overloading

- In case of method overloading, compilers do the following :
  - Seek the method having the same parameter type
  - Seek the method having the parameter which can be converted by basic type casting

# Static

# Static Fields

- A **static field** (a.k.a. *class field* or *class variable*) is shared by all objects of the class.
- A static field can hold a constant shared by all objects of the class:
- A **non-static field** (a.k.a. *instance field* or *instance variable*) belongs to an individual object.

# Static Fields

- Static fields are stored with the class code, separately from non-static fields that describe an individual object.
- Public static fields, usually global constants, are referred to in other classes using “dot notation”: `ClassName.constName`

```
double area = Math.PI * r * r;  
setBackground(Color.blue) ;  
c.add(btn, BorderLayout.NORTH) ;  
System.out.println(area) ;
```

# Static Methods

- Static methods can access and manipulate a class's static fields.
- Static methods cannot access non-static fields or call non-static methods of the class.
- Static methods are called using “dot notation”: `ClassName.statMethod(...)`

```
double x = Math.random();
double y = Math.sqrt (x);
System.exit();
```

# Instance Methods

- Non-static methods are also called *instance methods*.
- An instance method is called for a particular object using “dot notation”:  
`objName . instMethod(...);`
- Instance methods can access ALL fields and call ALL methods of their class — both class and instance fields and methods.

# Static (Class) vs. Non-Static (Instance)

```
public class MyClass
```

```
{
```

```
    public static final int statConst;  
    private static int statVar;
```

```
    private int instVar;
```

```
    ...
```

```
    ...
```

```
    public static int statMethod(...)
```

```
{
```

```
        statVar = statConst;  
        statMethod2(...);
```

```
        instVar = ...;
```

```
        instMethod(...);
```

```
}
```



All OK



```
    public int instMethod(...)
```

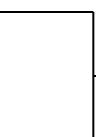
```
{
```

```
    statVar = statConst;  
    inst Var = statConst;  
    instVar = statMethod(...);  
    statVar = instMethod2(...);
```

```
    ...
```

```
}
```

OK



Error!



```
    public int instMethod2(...)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

```
}
```

# Static vs. Non-Static (cont'd)

- Note: `main` is **static** and therefore cannot access non-static fields or call non-static methods of its class:

```
public class Hello
{
    private String message = "Hello,
World";
    public static void main (String[ ]
args)
    {
        System.out.println (message);
    }
}
```

Error:  
non-static  
variable  
`message` is  
used in static  
context (`main`)

# Static Initialization Statement

- The Statement to be executed at the same time when the system initialize the static variable in the class
- From

```
static { <statement> }
```

# Static Initialization Statements

```
public class Test {  
    static{  
        System.out.println("Static");  
    }  
    {  
        System.out.println("Non-static block");  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        Test t2 = new Test();  
    }  
}
```

Static  
Non-static block  
Non-static block

# Static Initialization Statements

```
public class Main{  
    //instance variable initializer  
    String s = "abc";  
    //constructor  
    public Main() {  
        System.out.println("constructor called");    }  
    static int i, j=4;  
    //static initializer  
    static {  
        System.out.println("static initializer called");  
        System.out.println(j);    }  
    //instance initializer  
    { System.out.println("instance initializer called"); }  
    public static void main(String[] args) {  
        new Main();  
        new Main();    }  
}
```

Output:

static initializer called  
4

instance initializer called  
constructor called  
instance initializer called  
constructor called

# Static Initialization Statement

## Execution Order

- Order of initialization of static init. Statement and static variable : existing order in the program

```
class Initializers {  
    static { i = j + 2; }      // Error  
    static int i, j;  
    static j = 4;  
    //...  
}
```

- Executed earlier than constructor

# return

- A method, unless `void`, returns a value of the specified type to the calling method.
- The `return` statement is used to immediately quit the method and return a value:

```
return expression;
```

The type of the return value or expression must match the method's declared return type.

# return

- A method can have several `return` statements; then all but one of them must be inside an `if` or `else` (or in a `switch`):

```
public someType myMethod (...)  
{  
    ...  
    if (...)  
        return <expression1>;  
    else  
        return <expression2>;  
    ...  
    return <expression3>;  
}
```

# return

- A boolean method can return true, false, or the result of a boolean expression:

```
public boolean myMethod  
(...)  
{  
    ...  
    if (...)  
        return true;  
    ...  
    return n % 2 == 0;  
}
```

# return

- A `void` method can use a `return` statement to quit the method early:

```
public void myMethod (...)  
{  
    ...  
    if (...)  
        return;  
    ...  
}
```

No need for a redundant `return` at the end

# return

- If its return type is a class, the method returns a reference to an object (or `null`).
- Often the returned object is created in the method using `new`. For example:

```
public Fraction inverse ()  
{  
    if (num == 0)  
        return null;  
    return new Fraction (denom, num);  
}
```

- The returned object can also come from the arguments or from calls to other methods.

# The main Method

# The Main Method - Concept

## □ **main** method

- the system locates and runs the main method for a class when you run a program
- other methods get execution when called by the main method explicitly or implicitly
- must be public, static and void

# The Main Method - Getting Input from the Command Line

- When running a program through the `java` command, you can provide a list of strings as the real arguments for the `main` method. In the `main` method, you can use `args[index]` to fetch the corresponding argument

```
class Greetings {  
    public static void main (String args[]) {  
        String name1 = args[0];  
        String name2 = args[1];  
        System.out.println("Hello " + name1 + "&" + name2);  
    }  
}
```

➤ `java Greetings Jacky Mary`  
`Hello Jacky & Mary`

- Note: What you get are strings! You have to convert them into other types when needed.

# Passing Arguments

- Primitive data types are always passed “by value”: the value is copied into the parameter

```
public class MyMath  
{  
    ...  
    public double square (double x)  
    {  
        x *= x;      ← x here is a copy of the  
        return x;    argument. The copy is  
    }                changed, but...  
}
```

```
MyMath calc = new MyMath();  
double x = 3.0;  
double y = calc.square (x);  
System.out.println (x + " " + y);  
...
```

... the original **x** is unchanged.  
Output: 3 9

# Passing Objects as Arguments

- Objects are always passed as references:  
the address is copied, not the object.

```
Fraction f1 = new Fraction (1, 2);  
Fraction f2 = new Fraction (5, 17);  
Fraction f3 = f1.add (f2);
```

```
public class Fraction  
{
```

```
...  
public Fraction add (Fraction f)  
{
```

```
    Fraction sum;
```

```
...
```

```
}
```

```
}
```

f1: addr1

f2: addr2

copy

5/17

f: addr2

sum:

# import

- Full library class names include the package name. For example:

`java.awt.Color`

`javax.swing.JButton`

- `import` statements at the top of your program let you refer to library classes by their short names:

```
import javax.swing.JButton;
```

*Fully-qualified  
name*

...

```
JButton go = new JButton("Click here");
```

# import (cont'd)

- You can import names for all the classes in a package by using a wildcard `.*`:

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

Imports all classes from `awt`, `awt.event`, and `swing` packages

- `java.lang` is imported automatically into all classes; defines `System`, `Math`, `Object`, `String`, and other commonly used classes.

# Practice problems

- Rectangle
- Person
- Group- 1, 3, 5,
- Group- 0, 2, 4,
- Fraction
- Point3D
- Group- 7, 9
- Group- 6, 8

# Fraction

# Fraction- Methods

```
public class Fraction  
{
```

```
    private int num, denom;
```

```
    public Fraction reduce( )  
    {
```

```
}
```

```
    public Fraction invert(Fraction a){  
    }
```



```
    public Fraction (int p, int q)  
    {  
        num = p;  
        denom = q;  
        // reduce  
    }
```

```
    public Fraction add(Fraction a,  
Fraction b){
```

```
}
```

```
    public Fraction add (Fraction a, int n)  
    { ... }  
}
```

# Fraction Methods

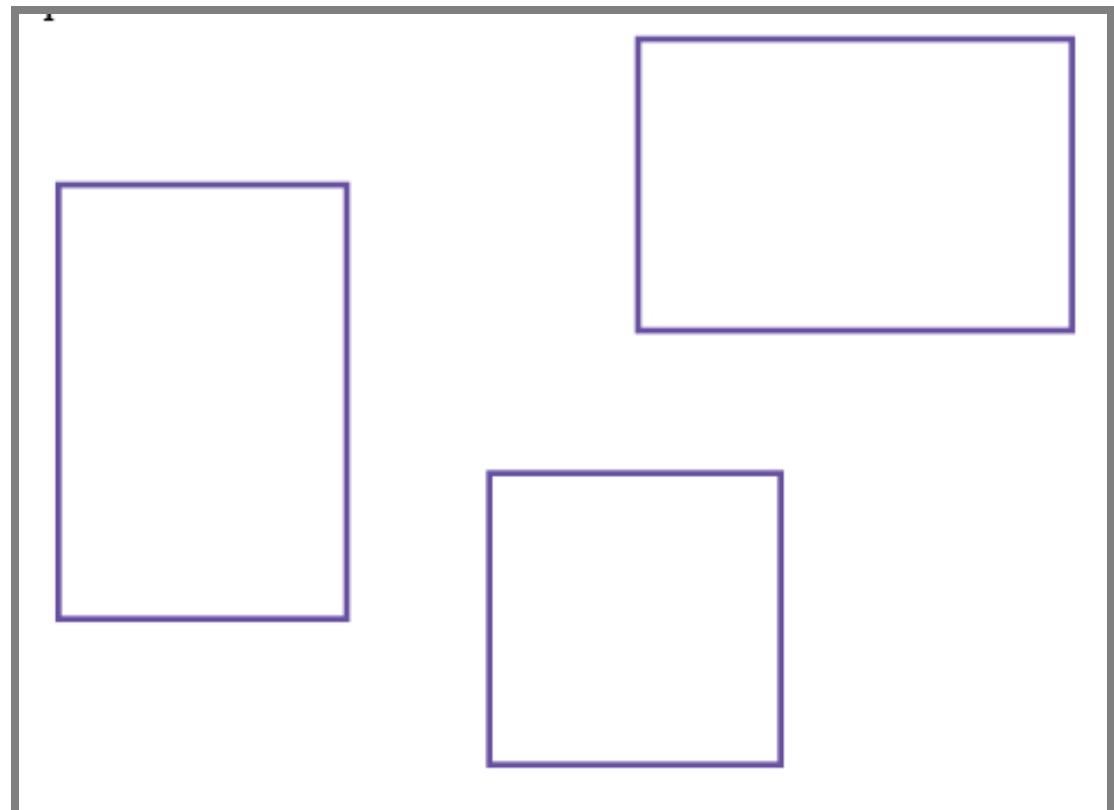
```
public class Fraction
{ int num, denom;
  public Fraction (int p, int q)
  {
    num = p;
    denom = q;
  }
  public add ( Fraction f)...
  public Fraction invert (Fraction f)...
}
```

```
class FractionDemo {
  public static void main(String args[]) {
    //Create 3 Fraction objects,
    // initialize 2 with 4 hardcoded values
    (i,j,k,l)
    // Print them on screen
    // add the two Fraction objects
    and assign it to third Fraction
    //invert method should verify if
    numerator is zero.
  }
}
```

# Rectangle

# Rectangular Shapes and Rectangle Objects

- Objects of type Rectangle *describe* rectangular shapes



Rectangular  
Shapes

# Rectangular Shapes and Rectangle Objects

- A `Rectangle` object isn't a rectangular shape—it is an object that contains a set of numbers that describe the rectangle

<u>Rectangle</u>	
x =	5
y =	10
width =	20
height =	30
<u>Rectangle</u>	
x =	35
y =	30
width =	20
height =	20
<u>Rectangle</u>	
x =	45
y =	0
width =	30
height =	20

Rectangular  
Objects

# Rectangle Class

---

Give code for Rectangle class

- It should have 4 fields of int type for x, y, width and height
- It should have 2 constructors
  - No argument constructor which sets all field values to 0
  - Four argument constructor which sets all field values.
- Translate method: which will move a Rectangle to the coordinates passed as arguments.

Give code for a RectangleDemo class

- Create Four Rectangle objects
- Identify if any of them overlap !!!!

# Constructing Objects

```
[new Rectangle(5, 10, 20, 30)]
```

[Detail:

1. The new operator makes a Rectangle object
2. It uses the parameters (in this case, 5, 10, 20, and 30) to initialize the data of the object
3. It returns the object

[Usually the output of the new operator is stored in a variable]

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

# Constructing Objects

- The process of creating a new object is called *construction*
- The four values 5, 10, 20, and 30 are called the *construction parameters*
- Some classes let you construct objects in multiple ways

```
new Rectangle()  
// constructs a rectangle with its top-left corner  
// at the origin (0, 0), width 0, and height 0
```

# Self Check

---

- How do you construct a square with center (100, 100) and side length 20?

# Answers

---

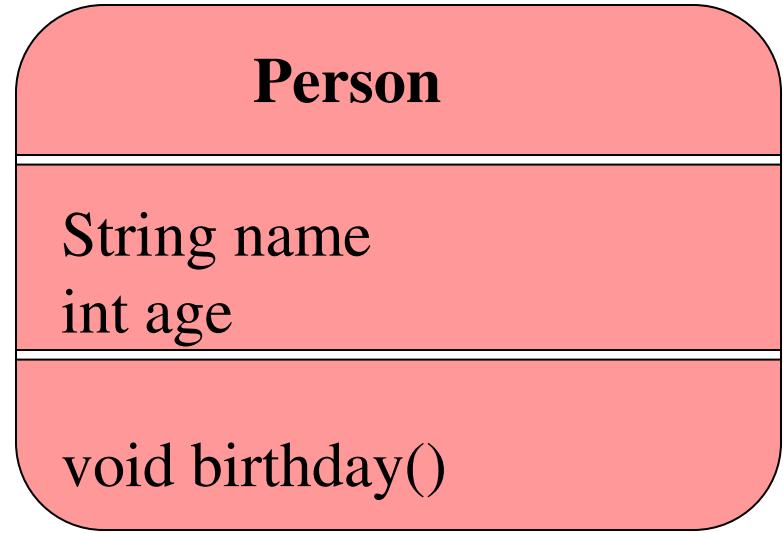
[

```
new Rectangle(90, 90, 20, 20)
```

# Person

# Person class

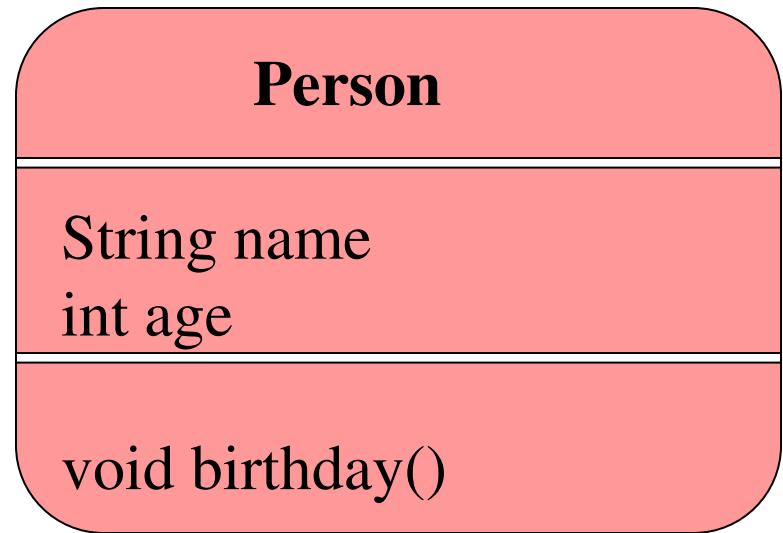
```
class Person {  
    String name;  
    int age;  
  
    void birthday () {  
        age++;  
        System.out.println(name+ 'is now '  
                           +age);  
    }  
}
```



# An example of a class

Create a class **HelloWorldPerson**  
which will:

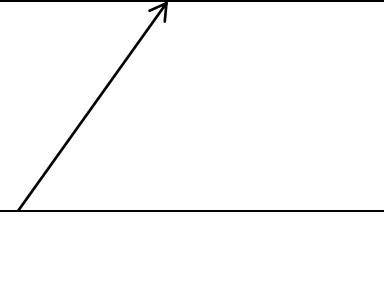
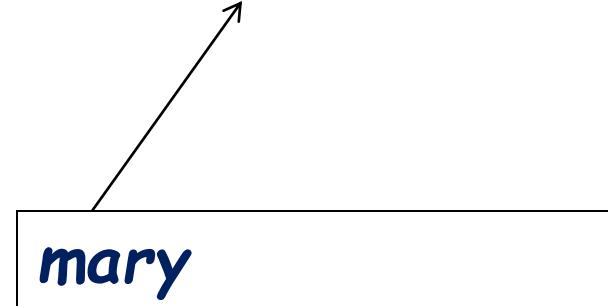
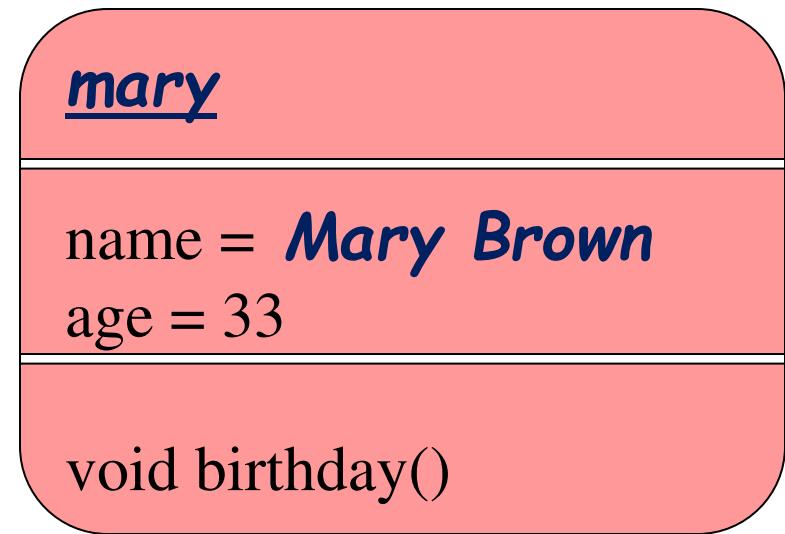
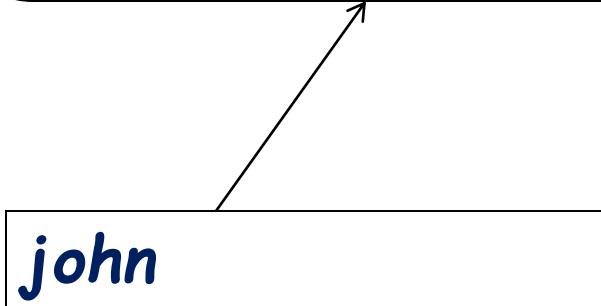
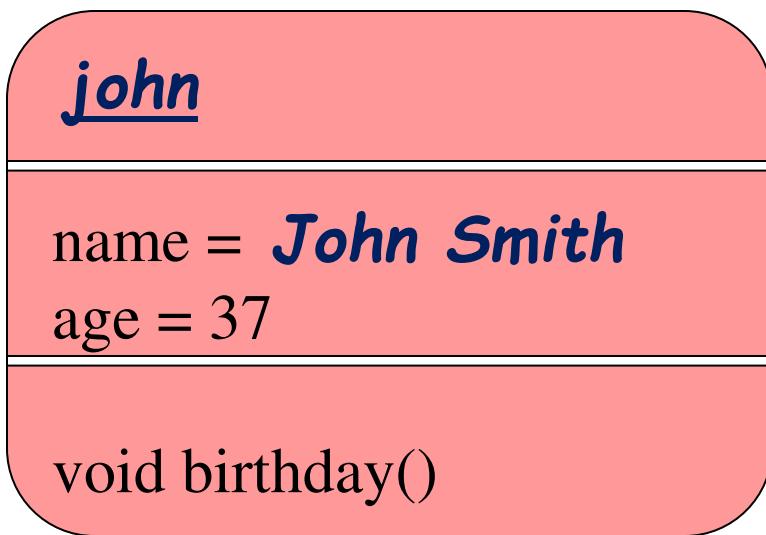
- Create 5 person objects and add to a Person array (hard code values)
- Print as output the names of all the Persons in the array
- Sort the array in increasing order of the ages of Person



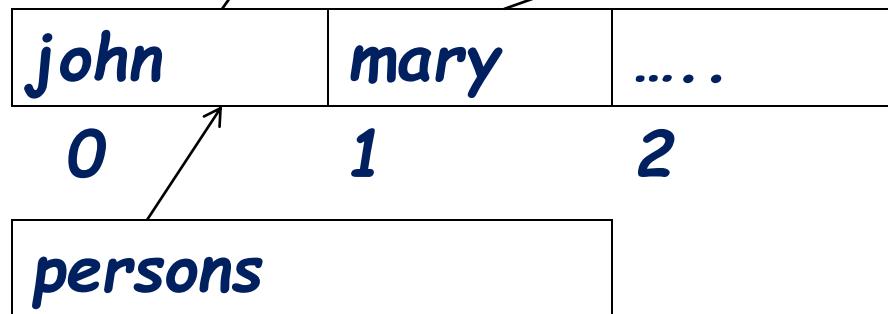
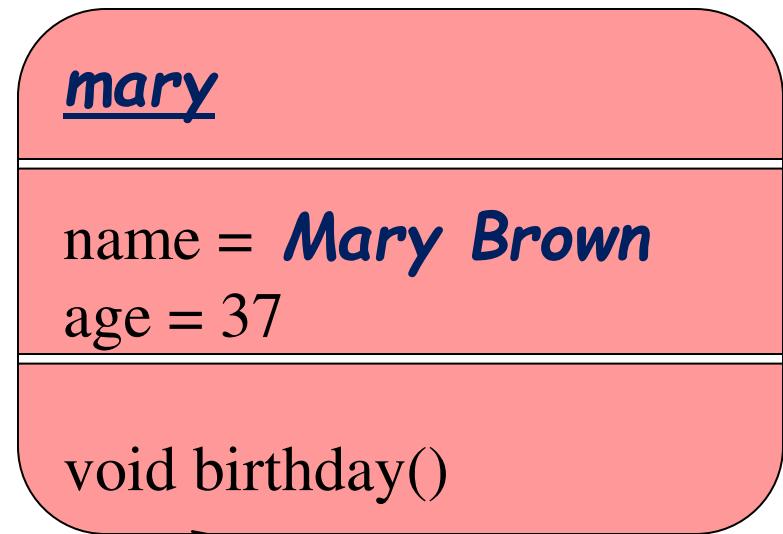
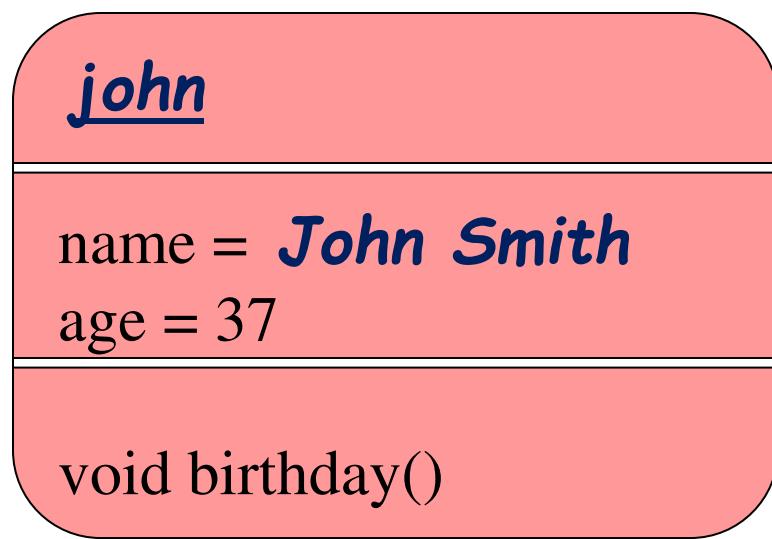
# Creating and using objects

```
public class HelloWorldPerson {  
    public static void main(String[] args){  
        Person john;          // Declaring object  
        john = new Person ( ); // creating object  
        john.name = "John Smith";  
        john.age = 37;  
  
        Person mary = new Person ( );  
        mary.name = "Mary Brown";  
        mary.age = 33;  
        mary.birthday ( );  
    }  
}
```

# Examples of Objects of type Person



# Array of Objects of type Person



# Array of Objects of type Person

```
Person [] persons= new Person[10];  
persons[0] = john;
```

....

# Point3D

# Point3D

```
class Point3D {  
    double x;  
    double y;  
    double z;  
    Point3D(double x, double y, double z)  
{  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
  
    move(double x, double y, double z){ }  
    move(Point3D p){ }  
}
```

```
class ThisKeywordDemo {  
    public static void main(String args[])  
{  
    Point3D p = new Point3D(1.1, 3.4, -2.8);  
    System.out.println("p.x = " + p.x);  
    System.out.println("p.y = " + p.y);  
    System.out.println("p.z = " + p.z);  
}  
}
```

# Practice problems

- Rectangle
- Person
- Group- 1, 3, 5,
- Group- 0, 2, 4,
- Fraction
- Point3D
- Group- 7, 9
- Group- 6, 8

# Fraction

# Fraction- Methods

```
public class Fraction  
{
```

```
    private int num, denom;
```

```
    public Fraction reduce( )  
    {
```

```
}
```

```
    public Fraction invert(Fraction a){
```

```
}
```



```
    public Fraction (int p, int q)  
    {  
        num = p;  
        denom = q;  
        // reduce  
    }
```

```
    public Fraction add(Fraction a,  
Fraction b){
```

```
}
```

```
    public Fraction add (Fraction a, int n)  
    { ... }  
}
```

# Fraction Methods

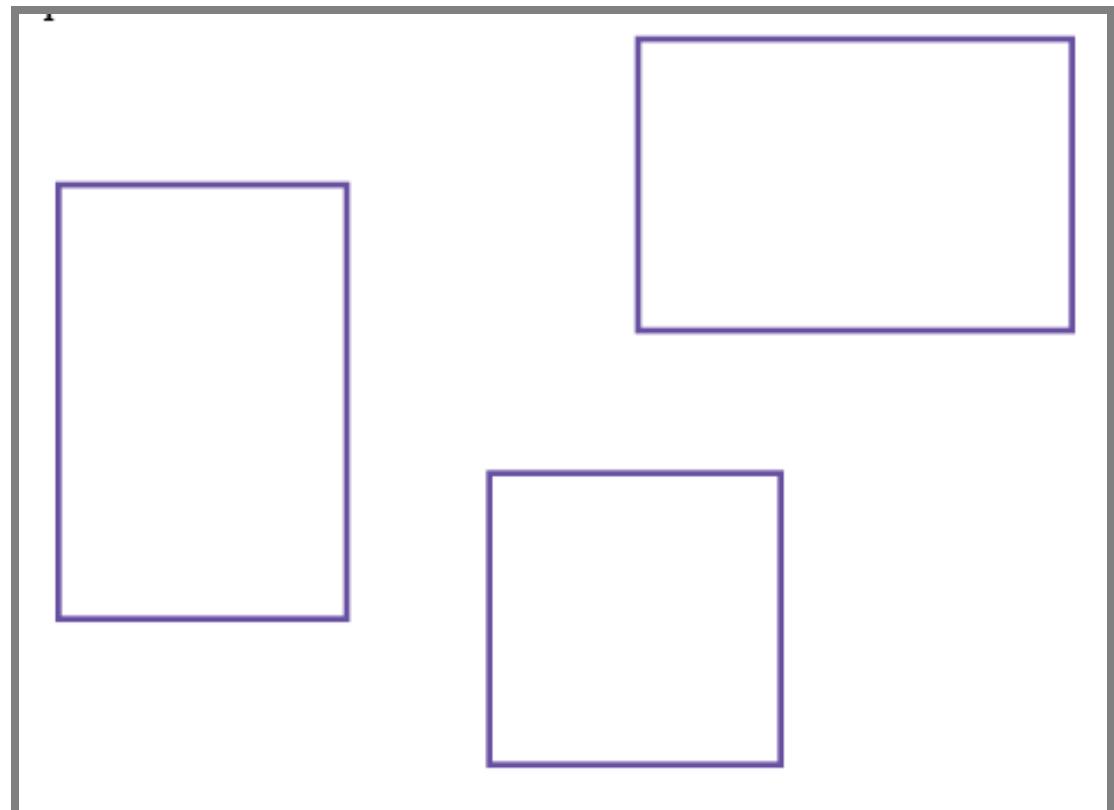
```
public class Fraction
{ int num, denom;
  public Fraction (int p, int q)
  {
    num = p;
    denom = q;
  }
  public add ( Fraction f)...
  public Fraction invert (Fraction f)...
}
```

```
class FractionDemo {
  public static void main(String args[]) {
    //Create 3 Fraction objects,
    // initialize 2 with 4 hardcoded values
    (i,j,k,l)
    // Print them on screen
    // add the two Fraction objects
    and assign it to third Fraction
    //invert method should verify if
    numerator is zero.
  }
}
```

# Rectangle

# Rectangular Shapes and Rectangle Objects

- Objects of type Rectangle *describe* rectangular shapes



Rectangular  
Shapes

# Rectangular Shapes and Rectangle Objects

- A `Rectangle` object isn't a rectangular shape—it is an object that contains a set of numbers that describe the rectangle

<u>Rectangle</u>	
x =	5
y =	10
width =	20
height =	30
<u>Rectangle</u>	
x =	35
y =	30
width =	20
height =	20
<u>Rectangle</u>	
x =	45
y =	0
width =	30
height =	20

Rectangular  
Objects

# Rectangle Class

---

Give code for Rectangle class

- It should have 4 fields of int type for x, y, width and height
- It should have 2 constructors
  1. No argument constructor which sets all field values to 0
  2. Four argument constructor which sets all field values.
- Translate method: which will move a Rectangle to the coordinates passed as arguments.

Give code for a RectangleDemo class

- Create Four Rectangle objects
- Identify if any of them overlap !!!!

# Constructing Objects

```
[new Rectangle(5, 10, 20, 30)]
```

[Detail:

1. The new operator makes a Rectangle object
2. It uses the parameters (in this case, 5, 10, 20, and 30) to initialize the data of the object
3. It returns the object

[Usually the output of the new operator is stored in a variable]

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

# Constructing Objects

- The process of creating a new object is called *construction*
- The four values 5, 10, 20, and 30 are called the *construction parameters*
- Some classes let you construct objects in multiple ways

```
new Rectangle()  
// constructs a rectangle with its top-left corner  
// at the origin (0, 0), width 0, and height 0
```

# Self Check

---

- How do you construct a square with center (100, 100) and side length 20?

# Answers

---

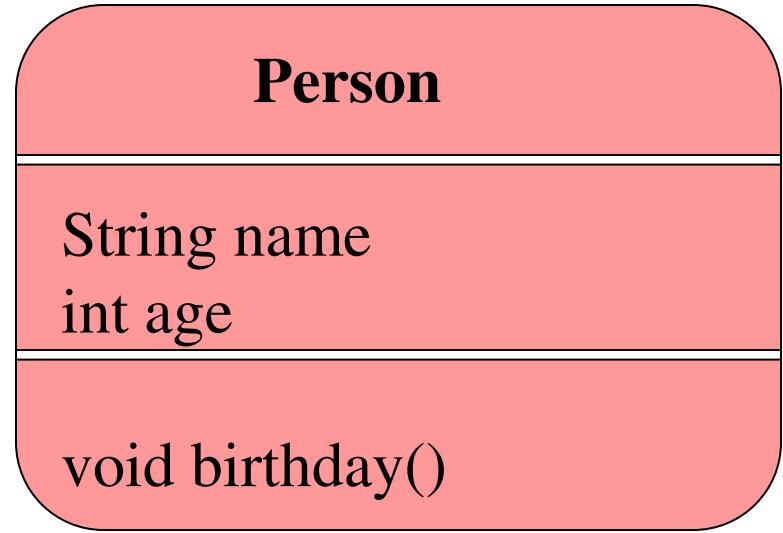
[

```
new Rectangle(90, 90, 20, 20)
```

# Person

# Person class

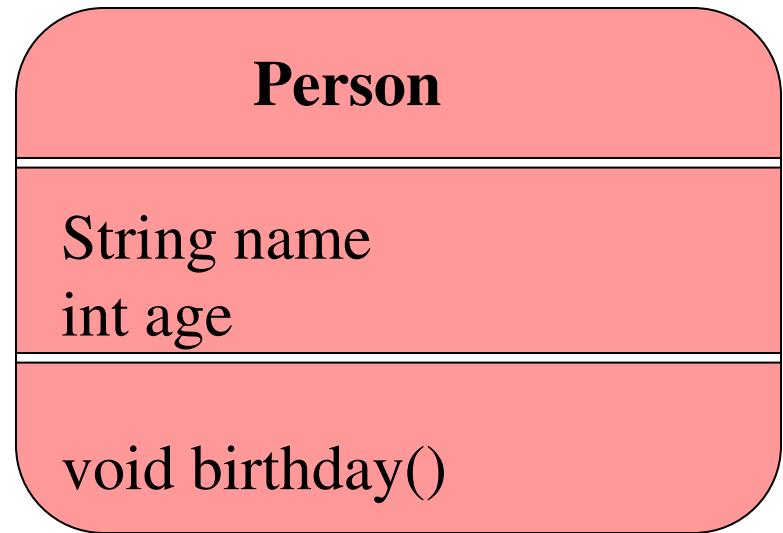
```
class Person {  
    String name;  
    int age;  
  
    void birthday () {  
        age++;  
        System.out.println(name+ 'is now '  
                           +age);  
    }  
}
```



# An example of a class

Create a class **HelloWorldPerson**  
which will:

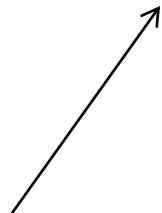
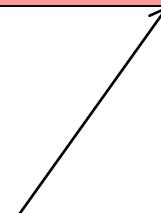
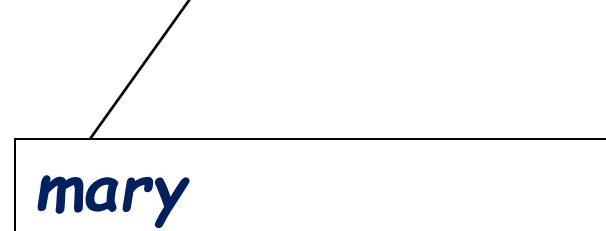
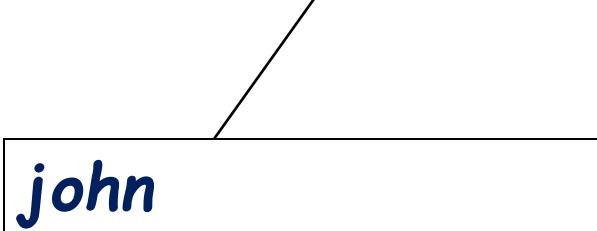
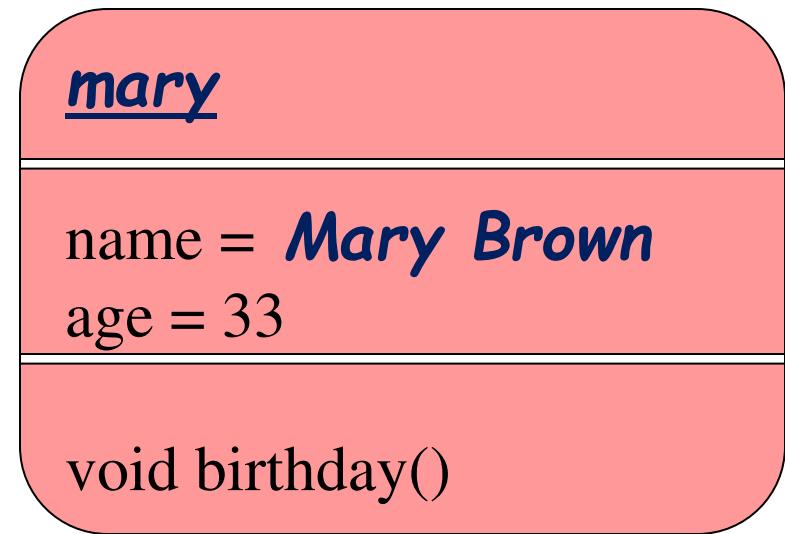
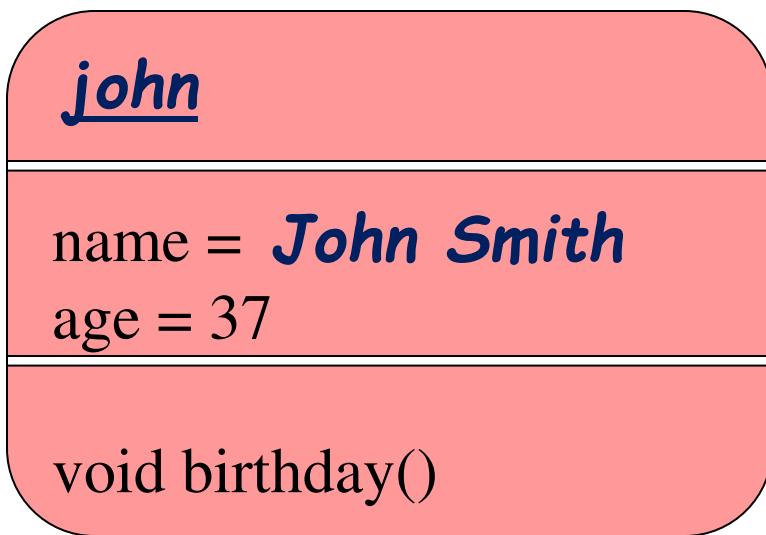
- Create 5 person objects and add to a Person array (hard code values)
- Print as output the names of all the Persons in the array
- Sort the array in increasing order of the ages of Person



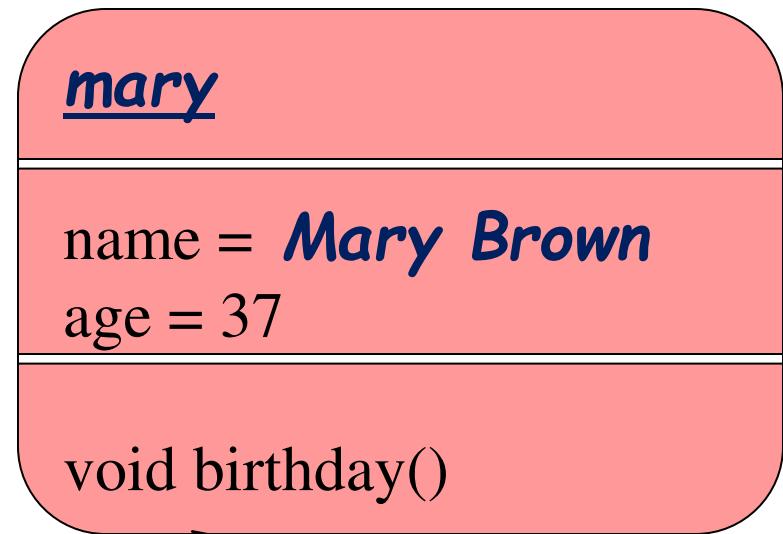
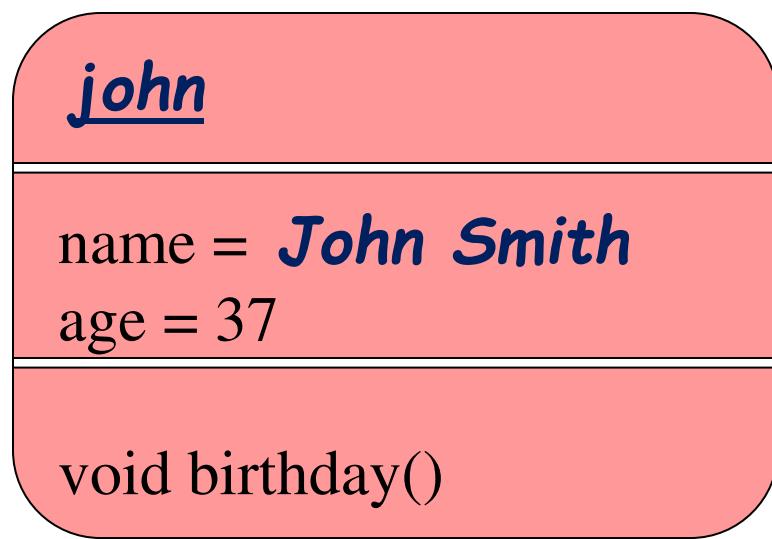
# Creating and using objects

```
public class HelloWorldPerson {  
    public static void main(String[] args){  
        Person john; // Declaring object  
        john = new Person ( ); // creating object  
        john.name = "John Smith";  
        john.age = 37;  
  
        Person mary = new Person ( );  
        mary.name = "Mary Brown";  
        mary.age = 33;  
        mary.birthday ( );  
    }  
}
```

# Examples of Objects of type Person



# Array of Objects of type Person



# Array of Objects of type Person

```
Person [] persons= new Person[10];  
persons[0] = john;
```

....

# Point3D

# Point3D

```
class Point3D {  
    double x;  
    double y;  
    double z;  
    Point3D(double x, double y, double z)  
{  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
  
    move(double x, double y, double z){ }  
    move(Point3D p){ }  
}
```

```
class ThisKeywordDemo {  
    public static void main(String args[])  
{  
    Point3D p = new Point3D(1.1, 3.4, -2.8);  
    System.out.println("p.x = " + p.x);  
    System.out.println("p.y = " + p.y);  
    System.out.println("p.z = " + p.z);  
}  
}
```

# The **String** Class

# Strings

---

- A string is a sequence of characters
- Strings are objects of the `String` class
- String constants:

```
"Hello, World!"
```

- String variables:

```
String message = "Hello, World!";
```

- String length:

```
int n = message.length();
```

- Empty string:

```
""
```

## Constructing a String:

```
String message = "Welcome to Java!"
```

String is an Object in Java.

# Constructing Strings

```
Strings newString = new String(message);
```

```
String message = new String("Welcome to Java!");
```

Since strings are used frequently, Java provides a **shorthand notation** for creating a string:

```
String message = "Welcome to Java!";
```

# Concatenation

- Use the `+` operator:

```
String name = "Dave";  
String message = "Hello, " + name;  
// message is "Hello, Dave"
```

- If one of the arguments of the operator is a string, the other is converted to a string

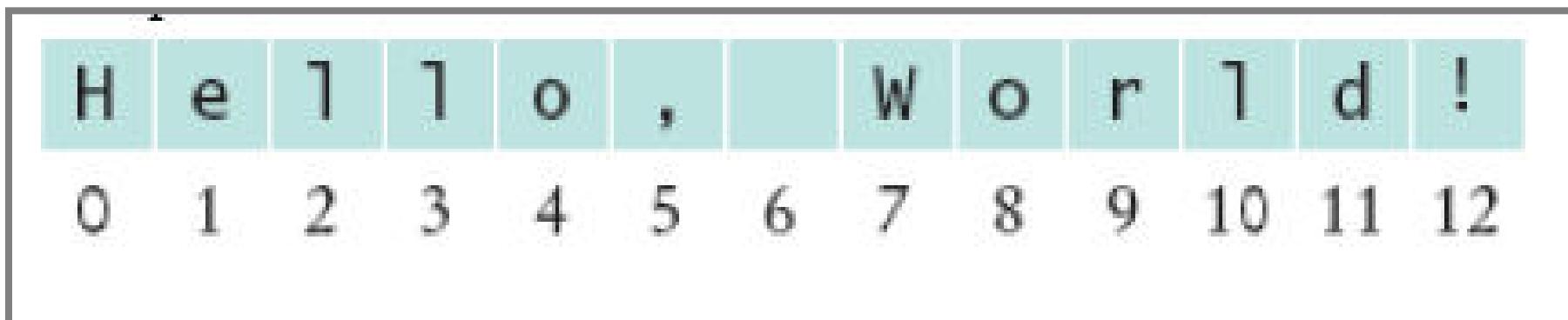
```
String a = "Agent";  
int n = 7;  
String bond = a + n;  
  
// bond is "Agent7"
```

# String

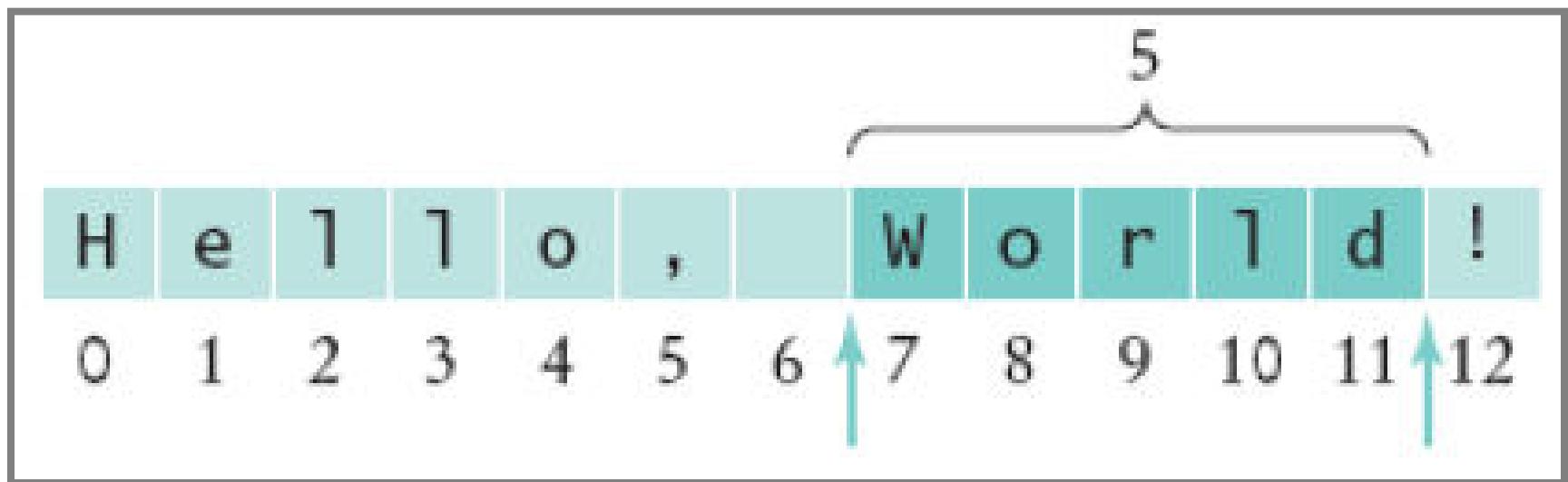
- +String()
- +String(value: String)
- +String(value: char[])
- +charAt(index: int): char
- +compareTo(anotherString: String): int
- +compareToIgnoreCase(anotherString: String): int
- +concat(anotherString: String): String
- +endsWithSuffix(suffix: String): boolean
- +equals(anotherString: String): boolean
- +equalsIgnoreCase(anotherString: String): boolean
- +indexOf(ch: int): int
- +indexOf(ch: int, fromIndex: int): int
- +indexOf(str: String): int
- +indexOf(str: String, fromIndex: int): int
- +intern(): String
- +regionMatches(toffset: int, other: String, offset: int, len: int): boolean
- +length(): int
- +replace(oldChar: char, newChar: char): String
- +startsWith(prefix: String): boolean
- +subString(beginIndex: int): String
- +subString(beginIndex: int, endIndex: int): String
- +toCharArray(): char[]
- +toLowerCase(): String
- +toString(): String

# Substrings

- String greeting = "Hello, World!";  
String sub = greeting.substring(0, 5);
- Supply start and “past the end” position
- First position is at 0, last position is (5-1) -> Hello



# Substrings



# Self Check

---

- Assuming the String variable `s` holds the value "Agent", what is the effect of the assignment `s = s + s.length()`?
- Assuming the String variable `river` holds the value "Mississippi", what is the value of `river.substring(1, 2)`? Of `river.substring(2, river.length() - 3)`?

# Answers

---

- s is set to the string Agent5
- The strings "i" and "ssissi"

# Retrieving Individual Characters in a String

- Do not use `message[0]`
- Use `message.charAt(index)`
- Index starts from 0

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
message	W	e	l	c	o	m	e		t	o		J	a	v	a

`message.charAt(0)`      `message.length()` is 15      `message.charAt(14)`

# String Concatenation

```
String s3 = s1.concat(s2);
```

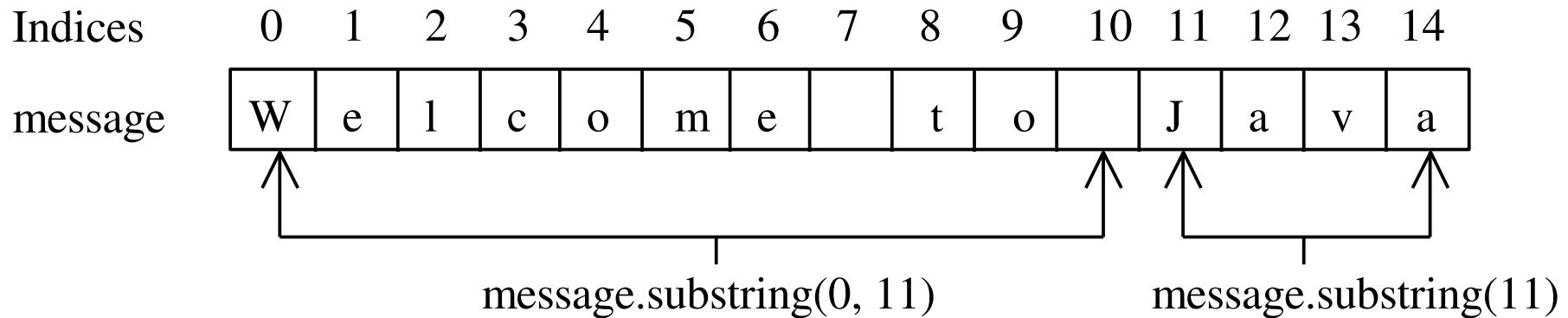
```
String s3 = s1 + s2;
```

# Extracting Substrings

String is an immutable class; its values cannot be changed individually.

```
String s1 = "Welcome to Java";
```

```
String s2 = s1.substring(0, 11) + "HTML";
```



# Strings Are Immutable

A String object is **immutable**, whose contents cannot be changed.

To improve efficiency and save memory, Java Virtual Machine stores two String objects into the same object, if the two String objects are created with the same string literal using the shorthand notation. Therefore, the shorthand notation is preferred to create strings.

# String ==

```
String s = "Welcome to Java!";
```

```
String s1 = new String("Welcome to Java!");
```

```
String s2 = s1.intern();
```

```
System.out.println("s1 == s is " + (s1 == s));
```

```
System.out.println("s2 == s is " + (s2 == s));
```

```
System.out.println("s1 == s2 is " + (s1 == s2));
```

displays

s1 == s is false

s2 == s is true

s1 == s2 false

# String practice problem

Given Strings s1, s2, s3, Give code using the String class methods

- Order the strings Lexicographically Gr 3,6
- Checking whether a string is a palindrome Gr 5, 2
- Convert s1 into substrings by breaking where there are spaces Gr 7, 0
- Create a string from s1 inserting "ab" wherever there is a 'a' Gr 9, 4
- Set a boolean array barry[] based on whether these strings start with "at". Gr 1, 8

- Create questions which use the String class methods.

# String Comparisons: equals

```
String s1 = "Welcome";
String s2 = "welcome";

if (s1.equals(s2)){
    // s1 and s2 have the same contents
}

if (s1 == s2) {
    // s1 and s2 have the same reference
}
```

# compareTo(Object object)

```
if (s1.compareTo(s2) > 0) {  
    // s1 is greater than s2  
}
```

# String Conversions

The contents of a string cannot be changed once the string is created. But you can convert a string to a new string using the following methods:

- `toLowerCase`
- `toUpperCase`
- `trim`
- `replace(oldChar, newChar)`

# Finding a Character or a Substring in a String

"Welcome to Java!".indexOf('W')) returns 0.

"Welcome to Java!".indexOf('x')) returns -1.

"Welcome to Java!".indexOf('o', 5)) returns 9.

"Welcome to Java!".indexOf("come")) returns 3.

"Welcome to Java!".indexOf("Java", 5)) returns 11.

"Welcome to Java!".indexOf("java", 5)) returns -1.

# Convert Character and Numbers to Strings

`valueOf` methods for converting a character, an array of characters, and numeric values to strings. These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`.

Convert a double value to a string  
`String.valueOf(5.44)`.

# The Character Class

## Character

- +Character(value: char)
- +charValue(): char
- +compareTo(anotherCharacter: Character): int
- +equals(anotherCharacter: Character): boolean
- +isDigit(ch: char): boolean
- +isLetter(ch: char): boolean
- +isLetterOrDigit(ch: char): boolean
- +isLowerCase(ch: char): boolean
- +isUpperCase(ch: char): boolean
- +toLowerCase(ch: char): char
- +toUpperCase(ch: char): char

# Examples

charObject.compareTo(new Character('a')) returns 1  
charObject.compareTo(new Character('b')) returns 0  
charObject.compareTo(new Character('c')) returns -1  
charObject.compareTo(new Character('d')) returns -2  
charObject.equals(new Character('b')) returns true  
charObject.equals(new Character('d')) returns false

# Counting Each Letter in a String

Write a program that counts the number of occurrence of each letter in a string. Assume the letters are not case-sensitive.

# The **StringBuffer** Class

The StringBuffer class is an alternative to the String class. In general, a string buffer can be used wherever a string is used.

StringBuffer is more flexible than String. You can add, insert, or append new contents into a string buffer. However, the value of a string is fixed once the string is created.

# StringBuffer

- +append(data: char[]): StringBuffer
- +append(data: char[], offset: int, len: int): StringBuffer
- +append(v: *aPrimitiveType*): StringBuffer
- +append(str: String): StringBuffer
- +capacity(): int
- +charAt(index: int): char
- +delete(startIndex: int, endIndex: int): StringBuffer
- +deleteCharAt(int index): StringBuffer
- +insert(index: int, data: char[], offset: int, len: int): StringBuffer
- +insert(offset: int, data: char[]): StringBuffer
- +insert(offset: int, b: *aPrimitiveType*): StringBuffer
- +insert(offset: int, str: String): StringBuffer
- +length(): int
- +replace(int startIndex, int endIndex, String str): StringBuffer
- +reverse(): StringBuffer
- +setCharAt(index: int, ch: char): void
- +setLength(newLength: int): void
- +substring(start: int): StringBuffer
- +substring(start: int, end: int): StringBuffer

# StringBuffer Constructors

- `public StringBuffer()`  
No characters, initial capacity 16 characters.
- `public StringBuffer(int length)`  
No characters, initial capacity specified by  
the length argument.
- `public StringBuffer(String str)`  
Represents the same sequence of characters  
as the string argument. Initial capacity 16  
plus the length of the string argument.

# Appending New Contents into a String Buffer

```
StringBuffer strBuf = new StringBuffer();
strBuf.append("Welcome");
strBuf.append(' ');
strBuf.append("to");
strBuf.append(' ');
strBuf.append("Java");
```

# Object Oriented Programming

Wrapper classes, Java Type System, The Object class

# Primitive Types

Type	Description	Size
<code>int</code>	The integer type, with range <code>-2,147,483,648 . . . 2,147,483,647</code>	<b>4 bytes</b>
<code>byte</code>	The type describing a single byte, with range <code>-128 . . . 127</code>	<b>1 byte</b>
<code>short</code>	The short integer type, with range <code>-32768 . . . 32767</code>	<b>2 bytes</b>
<code>long</code>	The long integer type, with range – <code>9,223,372,036,854,775,808 . . .</code> <code>-9,223,372,036,854,775,807</code>	<b>8 bytes</b>

*Continued...*

# Primitive Types

Type	Description	Size
<code>double</code>	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
<code>float</code>	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
<code>char</code>	The character type, representing code units in the Unicode encoding scheme	2 bytes
<code>boolean</code>	The type with the two truth values <code>false</code> and <code>true</code>	1 byte

# Primitives & Wrappers

- Java has a *wrapper* class for each of the eight primitive data types:

Primitive Type	Wrapper Class	Primitive Type	Wrapper Class
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short

# Use of the Wrapper Classes

- Java's *primitive* data types (boolean, int, etc.) are not classes.
- Wrapper classes are used in situations where objects are required, such as for elements of a Collection:

```
List<Integer> a = new ArrayList<Integer>();  
methodRequiringListOfIntegers(a);
```

# Use of the Wrapper Classes: Methods for conversion

- Value => Object                    **valueOf**
- Object => Value  
                                          **intValue , booleanValue**
- String => value  
                                          **parseInt , parseBoolean , ...**

# Value => Object: Wrapper Object Creation

- *Wrapper.valueOf()* takes a value (or string) and returns an object of that class:

```
Integer i1 = Integer.valueOf(42);  
Integer i2 = Integer.valueOf("42");
```

```
Boolean b1 = Boolean.valueOf(true);  
Boolean b2 = Boolean.valueOf("true");
```

```
Long n1 = Long.valueOf(42000000L);  
Long n1 = Long.valueOf("42000000L");
```

# Object => Value

- Each wrapper class Type has a method typeValue to obtain the object's value:

```
Integer i1 = Integer.valueOf(42) ;  
Boolean b1 = Boolean.valueOf("false") ;  
System.out.println(i1.intValue()) ;  
System.out.println(b1.booleanValue()) ;
```

=>

42

false

# String => value

- The Wrapper class for each primitive *type* has a method `parseType()` to parse a string representation & return the literal value.

`Integer.parseInt("42")`      => 42

`Boolean.parseBoolean("true")`    => true

`Double.parseDouble("2.71")`      => 2.71

//...

- Common use: Parsing the arguments to a program:

# Parsing argument lists

```
// Parse int and float program args.  
public parseArgs(String[] args) {  
    for (int i = 0; i < args.length; i++) {  
  
        ...println(Integer.parseInt(args[i]));  
    }  
}
```

# Many useful utility methods: Integer

int	hashCode()
static int	numberOfLeadingZeros(int i)
static int	numberOfTrailingZeros(int i)
static int	reverse(int i)
static int	reverseBytes(int i)
static int	rotateLeft(int i, int distance)
static int	rotateRight(int i, int distance)
static String	toBinaryString(int i)
static String	toHexString(int i)
static String	toOctalString(int i)
static String	toString(int i, int radix)

# Double & Float: Utilities for Arithmetic Operations:

- Constants

`POSITIVE_INFINITY, NEGATIVE_INFINITY`

- Constant `NaN` = Not-a-Number (NaN) value.
- Methods `isNaN()`, `isInfinite()`

# Class Object

# Class Object

- **Object** is the root of the class hierarchy
  - Every *class* has **Object** as a superclass
- All classes inherit the methods of Object
  - But may override them

# Class Object

TABLE 3.2

Methods of Class `java.lang.Object`

Method	Behavior
<code>Object clone()</code>	Makes a copy of an object.
<code>boolean equals(Object obj)</code>	Compares this object to its argument.
<code>int hashCode()</code>	Returns an integer hash code value for this object.
<code>String toString()</code>	Returns a string that textually represents the object.

# The Method `toString`

- You should always override `toString` method if you want to print object state
  - If you do *not* override it:
    - `Object.toString` will return a `String`
    - Just not the `String` you want!
- Example: `ArrayBasedPD@ef08879`  
... The name of the class, @, instance's hash code

# Always override `toString()`

“When practical, the `toString` method should return all of the interesting info contained in the object.”

Note that `toString` should never print anything

# **toString() called automatically**

```
System.out.println( "Answer = " + 42 );
```

**toString() method is  
called automatically**

```
System.out.println( d1 );
```

```
System.out.println( d1.topFace() );
```

```
System.out.println( d1.toString() );
```

**unnecessary, adds clutter**

# Practice problems

- Take an input (using main) an array of Strings. Give methods using library functions which will
  - Give sum of all the Strings which are Numbers G3,9
  - Check if all the Strings have at least 5 different characters G4,8
  - Check if the 2<sup>nd</sup> String has repeating set of three sequential characters “abc” (eg. abcabc, abc1abcaa) G2,7
  - Count the number of spaces in all the strings and sort the strings in increasing number of spaces G1,6
  - Check if any of the Strings is a real number, if yes obtain the closest integer to it. And print as an output the integer and its square root G0,6

# Programming using Java

## Java: Inheritance

# Objectives:

- Get an introduction to Inheritance
- Get a general idea of how a hierarchy of classes is put together

# Inheritance: Definition

- **inheritance:** a parent-child relationship between classes
- allows sharing of the behavior of the parent class into its child classes
  - one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance
- child class can add new behavior or override existing behavior from parent

# Inheritance terms

- **superclass, base class, parent class:** terms to describe the parent in the relationship, which shares its functionality
- **subclass, derived class, child class:** terms to describe the child in the relationship, which accepts functionality from its parent
- **extend, inherit, derive:** become a subclass of another class

# Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
  - `public class CheckingAccount  
 extends BankAccount {`
- Java forces a class to have exactly one parent ("single inheritance")
  - other languages (C++) allow multiple inheritance

# Inheritance in Java

- the objects of your class will now receive all of the state (fields) and behavior (methods) of the parent class
- constructors and static methods/fields are not inherited
- by default, a class's parent is `Object`

# Inheritance Example

```
class BankAccount {  
    private double myBal;  
    public BankAccount() { myBal = 0; }  
    public double getBalance() { return myBal; }  
}
```

```
class CheckingAccount extends BankAccount {  
    private double myInterest;  
    public CheckingAccount(double interest) { }  
    public double getInterest() { return  
        myInterest; }  
    public void applyInterest() { }  
}
```

- CheckingAccount objects have myBal and myInterest fields, and getBalance(), getInterest(), and applyInterest() methods

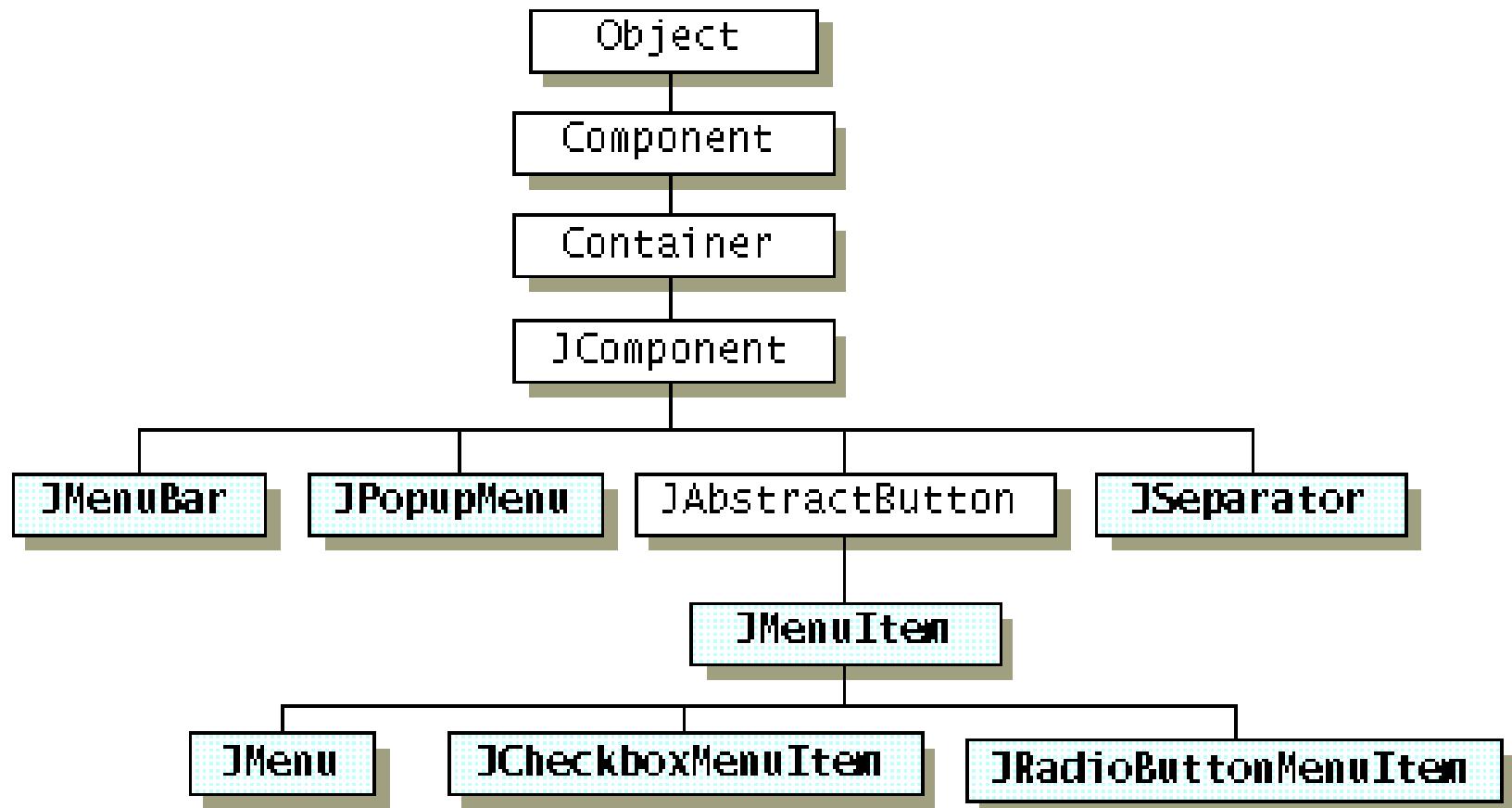
# Multiple layers of inheritance

- it is possible to extend a class that itself is a child class; inheritance chains like this can be arbitrarily deep

```
public class  
TransactionFeeCheckingAccount  
    extends CheckingAccount {  
    private static final double FEE =  
    2.00;  
  
    public void chargeFee() {  
        withdraw(FEE);  
    }  
}
```

# Inheritance Hierarchies

- Deeper layered chain of classes, many children extending many layers of parents



# "Has-a" Relationships

- "Has-a" relationship: when one object contains another as a field

```
public class BankAccountManager {  
    private List myAccounts;  
    // ...  
}
```

- a BankAccountManager object "has-a" List inside it, and therefore can use it

# "Is-a" relationships

- "Is-a" relationships represent sets of abilities; implemented through interfaces and inheritance

```
public class CheckingAccount  
    extends BankAccount {  
    // ...  
}
```

- a CheckingAccount object **"is-a"** BankAccount
  - therefore, it can do anything an BankAccount **can do**
  - it can be substituted wherever a BankAccount **is** needed
  - a **variable of type** BankAccount **may refer to a** CheckingAccount **object**

# Using the account classes

- CheckingAccount **inherits** BankAccount's methods

```
CheckingAccount c = new  
CheckingAccount(0.10);  
System.out.println(c.getBalance());  
c.applyInterest();  
 );
```

# Using the account classes

- a BankAccount variable can refer to a CheckingAccount object

```
BankAccount b2 = new CheckingAccount(0.06);  
System.out.println(b2.getBalance());
```

- an Object variable can point to either account type

```
Object o = new BankAccount();  
Object o2 = new CheckingAccount(0.09);
```

# Some code that won't compile

- CheckingAccount variable can't refer to BankAccount (not every BankAccount "is-a" CheckingAccount)

```
CheckingAccount c = new BankAccount();
```

- cannot call a CheckingAccount method on a variable of type BankAccount (can only use BankAccount behavior)

```
BankAccount b = new CheckingAccount(0.10);  
b.applyInterest();
```

- cannot use any account behavior on an Object variable
- ```
Object o = new CheckingAccount(0.06);  
System.out.println(o.getBalance());  
o.applyInterest();
```

# Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

# Overriding behavior

- Child class can replace the behavior of its parent's methods by redefining them

```
public class BankAccount {  
    private double myBalance; // ....  
    public String toString() {  
        return getID() + " $" + getBalance();  
    } }
```

```
public class FeeAccount extends BankAccount {  
  
    private static final double FEE = 2.00;  
    public String toString() { // overriding  
        return getID() + " $" + getBalance()  
            + " (Fee: $" + FEE + ")";  
    } }
```

# Overriding behavior example

```
BankAccount b = new BankAccount("Ed", 9.0);  
FeeAccount f = new FeeAccount("Jen", 9.0);  
  
System.out.println(b);  
System.out.println(f);
```

- **Output:**

Ed \$9.0

Jen \$9.0 (Fee: \$2.0)

# Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

# Access modifiers

- **public:** visible to all other classes  
**public** class BankAccount
- **private:** visible only to the current class, its methods, and every instance (object) of its class
  - a child class cannot refer to its parent's private variables and methods!**private** String myID;
- **protected:** visible to the current class, and all of its child classes  
**protected** int myWidth;
- **package (default access; no modifier):** visible to all classes in the current "package" (seen later)  
int myHeight;

# Access modifier problem

```
public class Parent {  
    private int field1;  
    protected int field2;  
    public int field3;  
    private void method1() {}  
    public void method2() {}  
    protected void setField1(int  
    value) {  
        field1 = value;  
    }  
}
```

# Access modifier problem

```
public class Child extends Parent {  
    public int field4;  
  
    public Child() { // Which are legal?  
        field4 = 0; // _____  
        field1++; // _____  
        field2++; // _____  
        field3++; // _____  
        method1(); // _____  
        method2(); // _____  
        setField1(field4); // _____  
    }  
}
```

# Some code that won't compile

```
public class Point2D {  
    private int x, y;  
    public Point2D(int x, int y) {  
        this.x = x;    this.y = y;  
    }  
}  
  
public class Point3D extends Point2D {  
    private int z;  
    public Point3D(int x, int y, int z) {  
        this.x = x;  this.y = y; // can't do  
this!  
        this.z = z;  
    }  
}
```

# The super Reference

- A child's constructor is **responsible** for calling the parent's constructor
- The **first line** of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

# super and constructors

- if the superclass has a constructor that requires any arguments (not `()`), you *must* put a constructor in the subclass and have it call the super-constructor (call to super-constructor must be the first statement)

# super and constructors

```
public class Point2D {  
    private int x, y;  
  
    public Point2D(int x, int y) {  
        this.x = x;    this.y = y;  
    }  
}  
  
public class Point3D extends Point2D {  
    private int z;  
    public Point3D(int x, int y, int z) {  
        super(x, y); // calls Point2D constructor  
        this.z = z;  
    }  
}
```

# super keyword

- used to refer to superclass (parent) of current class
- can be used to refer to parent class's methods, variables, constructors to call them
  - needed when there is a name conflict with current class
- useful when overriding and you want to keep the old behavior but add new behavior to it
- **syntax:**

```
super(args);           // call parent's constructor  
super.fieldName       // access parent's field  
super.methodName(args); // or method
```

# super example

```
public class BankAccount {  
    private double myBalance;  
    public BankAccount() { myBalance = 0; }  
    public double getBalance() { return  
        myBalance; }  
    public void withdraw(double amount) {  
        myBalance -= amount;  
    } }  
public class FeeAccount  
    extends BankAccount {  
    public void withdraw(double amount) {  
        super.withdraw(amount);  
        if (getBalance() < 100.00)  
            withdraw(2.00); // charge $2 fee  
    } }
```

- Can the withdraw(2.00) cause any problems?

# Constructors in extended classes

- A constructor of the extended class can invoke one of the superclass's constructors by using the *super* method.
- If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor
  - `super( )`is invoked automatically as the first statement of the extended class's constructor.
- Constructors are not methods and are NOT inherited.

# Three phases of an object's construction

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
  - invoke a superclass's constructor
  - initialize the fields by using their initializers and initialization blocks
  - execute the body of the constructor
- The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the Object class is reached

# To Illustrate the Construction Order. . .

```
class X {  
    protected int xOri = 1;  
    protected int whichOri;  
    public X() {  
        whichOri = xOri;  
    }  
}
```

```
class Y extends X {  
    protected int yOri = 2;  
    public Y() {  
        whichOri = yOri;  
    }  
}
```

**Y objectY = new Y();**

| <b>Step</b> | <b>what happens</b>          | <b>xOri</b> | <b>yOri</b> | <b>whichOri</b> |
|-------------|------------------------------|-------------|-------------|-----------------|
| 0           | fields set to default values | 0           | 0           | 0               |
| 1           | Y constructor invoked        | 0           | 0           | 0               |
| 2           | X constructor invoked        | 0           | 0           | 0               |
| 3           | Object constructor invoked   | 0           | 0           | 0               |
| 4           | X field initialization       | 1           | 0           | 0               |
| 5           | X constructor executed       | 1           | 0           | 1               |
| 6           | Y field initialization       | 1           | 2           | 1               |
| 7           | Y constructor executed       | 1           | 2           | 2               |

# Overloading and Overriding Methods

- *Overloading*: providing more than one method with the same name but different parameter list
  - overloading an inherited method means simply adding new method with the same name and different signature
- *Overriding*: replacing the superclass's implementation of a method with your own design.
  - both the parameter lists and the return types must be exactly the same
  - if an overriding method is invoked on an object of the subclass, then it's the subclass's version of this method that gets implemented
  - an overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed
  - the overriding method's throws clause can have fewer types listed than the method in the superclass, or more specific types

# Fields/Methods in Extended Classes

- An object of an extended class contains two sets of variables and methods
  1. fields/methods which are defined locally in the extended class
  2. fields/methods which are inherited from the superclass

?

# Accessibility and Overriding

- a method can be overridden only if it's accessible in the subclass
  - private methods in the superclass
    - cannot be overridden
    - if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated
  - package methods in the superclass
    - can be overridden if the subclass is in the same package as the superclass
  - protected, public methods
    - always will be

**Not as that simple as it seems!**

```
package P1;

public class Base {
    private void pri( ) { System.out.println("Base.pri()"); }
    void pac( ) { System.out.println("Base.pac()"); }
    protected void pro( ) { System.out.println("Base.pro()"); }
    public void pub( ) { System.out.println("Base.pub()"); }

    public final void show( ) {
        pri(); pac(); pro(); pub(); }
}
```

```
package P2;

import P1.Base;

public class Concrete1 extends Base {
    public void pri( ) { System.out.println("Concrete1.pri()"); }
    public void pac( ) { System.out.println("Concrete1.pac()"); }
    public void pro( ) { System.out.println("Concrete1.pro()"); }
    public void pub( ) { System.out.println("Concrete1.pub()"); }
}
```

**Concrete1 c1 = new Concrete1();  
c1.show( );**

**Output?**

Base.pri()  
Base.pac()  
Concrete1.pro()  
Concrete1.pub()

## Sample classes (cont.)

```
package P1;  
  
import P2.Concrete1;  
  
public class Concrete2 extends Concrete1 {  
    public void pri( ) { System.out.println("Concrete2.pri()"); }  
    public void pac( ) { System.out.println("Concrete2.pac()"); }  
    public void pro( ) { System.out.println("Concrete2.pro()"); }  
    public void pub( ) { System.out.println("Concrete2.pub()"); }  
}
```

---

**Concrete2 c2 = new Concrete2();  
c2.show( );**

### **Output?**

Base.pri()  
Concrete2.pac()  
Concrete2.pro()  
Concrete2.pub()

## Sample classes (cont.)

```
package P3;  
import P1.Concrete2;  
  
public class Concrete3 extends Concrete2 {  
    public void pri( ) { System.out.println("Concrete3.pri()"); }  
    public void pac( ) { System.out.println("Concrete3.pac()"); }  
    public void pro( ) { System.out.println("Concrete3.pro()"); }  
    public void pub( ) { System.out.println("Concrete3.pub()"); }  
}
```

```
Concrete3 c3 = new Concrete3();  
c3.show( );
```

---

### **Output?**

Base.pri()  
Concrete3.pac()  
Concrete3.pro()  
Concrete3.pub()

# Hiding fields

- Fields cannot be overridden, they can only be hidden
- If a field is declared in the subclass and it has the same name as one in the superclass, then the field belongs to the superclass cannot be accessed directly by its name any more

# Polymorphism

- An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it
- an object of an extended class can be used wherever the original class is used
- **Question:** given the fact that an object's actual class type may be different from its declared type, then when a method accesses an object's member which gets redefined in a subclass, then which member the method refers to (subclass's or superclass's)?
  - when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used
  - when you access a field, the declared type of the reference decides which implementation is used

## Output?

Extend.show: ExtendStr  
Extend.show: ExtendStr  
sup.str = SuperStr  
ext.str = ExtendStr

```
class SuperShow {  
    public String str = "SuperStr";  
  
    public void show( ) {  
        System.out.println("Super.show:" + str);  
    }  
}  
  
class ExtendShow extends SuperShow {  
    public String str = "ExtendedStr";  
  
    public void show( ) {  
        System.out.println("Extend.show:" + str);  
    }  
  
    public static void main (String[] args) {  
        ExtendShow ext = new ExtendShow( );  
        SuperShow sup = ext;  
        sup.show( ); //1  
        ext.show( ); //2      methods invoked through object reference  
        System.out.println("sup.str = " + sup.str); //3  
        System.out.println("ext.str = " + ext.str); //4      field access  
    }  
}
```

# protected members

- To allow subclass methods to access a superclass field, define it `protected`. But be cautious!
- Making methods `protected` makes more sense, if the subclasses can be trusted to use the method correctly, but other classes cannot.

# What protected really means

Precisely, a protected member is accessible

- Within the class itself
- within code in the same package
- it can also be accessed from a class through object references that are of at **least the same type as the class** – that is , references of the class's type or one of its subtypes

# What protected really means

```
public class Employee {  
    protected Date hireDay;  
    . . .  
}  
  
public class Manager extends Employee {  
    . . .  
    public void printHireDay (Manager p) {  
        System.out.println("mHireDay: " +  
                           (p.hireDay).toString());  
    }  
    // ok! The class is Manager, and the object reference type is also Manager  
  
    public void printHireDay (Employee p) {  
        System.out.println("eHireDay: " +  
                           (p.hireDay).toString());  
    }  
    // wrong! The class is Manager, but the object reference type is Employee  
    // which is a supertype of Manager  
    . . .  
}
```

# super example

```
public class BankAccount {  
    private double myBalance;  
    public BankAccount() { myBalance = 0; }  
    public double getBalance() { return myBalance; }  
    public void withdraw(double amount) {  
        myBalance -= amount; }  
    public String toString() {  
        return getID() + " $" + getBalance(); } }  
  
public class FeeAccount  
    extends BankAccount {  
    public void withdraw(double amount) {  
        super.withdraw(amount);  
        if (getBalance() < 100.00)  
            super.withdraw(2.00); // charge $2 fee } }
```

# Which method gets called?

```
BankAccount b = new FeeAccount ("Ed", 9.00);  
b.withdraw(5.00);  
System.out.println(b.getBalance());
```

- Will it call the withdraw method in BankAccount, leaving Ed with \$4?
- Will it call the withdraw method in FeeAccount, leaving Ed with \$2 (after his \$2 fee)?

# The answer: dynamic binding

- The version of `withdraw` from `FeeAccount` will be called
- The version of an object's method that gets executed is always determined by that object's type, not by the type of the variable
- The variable should only be looked at to determine whether the code would compile; after that, all behavior is determined by the object itself

# Static and Dynamic Binding

- **static binding:** methods and types that are hard-wired at compile time
  - static methods
  - referring to instance variables
  - the types of the reference variables you declare
- **dynamic binding:** methods and types that are determined and checked as the program is running
  - non-static (a.k.a virtual) methods that are called
  - types of objects that your variables refer to

# Polymorphism

- inheritance provides a way to achieve polymorphism in Java
- **polymorphism:** the ability to use identical syntax on different data types, causing possibly different underlying behavior to execute
  - example: If we have a variable of type BankAccount and call withdraw on it, it might execute the version that charges a fee, or the version from the checking account that tallies interest, or the regular version, depending on the type of the actual object.

# Type-casting and objects

- You cannot call a method on a reference unless the reference's type has that method

```
Object o = new BankAccount("Ed", 9.00);  
o.withdraw(5.00); // doesn't compile
```

- You can cast a reference to any subtype of its current type, and this will compile successfully

```
((BankAccount)o).withdraw(5.00);
```

# Converting Between Subclass and Superclass Types

---

- Occasionally you need to convert from a superclass reference to a subclass reference

```
BankAccount anAccount = (BankAccount) anObject;
```

- This cast is dangerous: if you are wrong, an exception is thrown

# The instanceof keyword

- Performs run-time type check on the object referred to by a reference variable
- Usage: *object-reference instanceof type* (result is a boolean expression)
  - if *type* is a class, evaluates to **true** if the variable refers to an object of *type* or any subclass of it.
  - if *type* is an interface, evaluates to true if the variable refers to an object that implements that interface.
  - if **object-reference is null, the result is false.**

- Example:

```
Object o = myList.get(2);  
if (o instanceof BankAccount)  
    ((BankAccount)o).deposit(10.0);
```

# Converting Between Subclass and Superclass Types

---

- Solution: use the `instanceof` operator
  - `instanceof`: tests whether an object belongs to a particular type
- 

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    . . .
}
```

# Down-casting and runtime

- It is illegal to cast a reference variable into an unrelated type (example: casting a String variable into a BankAccount)
- It is legal to cast a reference to the wrong subtype; this will compile but crash when the program runs
  - Will crash even if the type you cast it to has the method in question

```
((String)o).toUpperCase();           // crashes  
((FeeAccount)o).withdraw(5.00);    // crashes
```

# Some instanceof problems

```
Object o = new BankAccount(...);  
BankAccount c = new CheckingAccount(...);  
BankAccount n = new NumberedAccount(...);  
CheckingAccount c2 = null;
```

T/F

- ???
  - o instanceof Object
  - \_\_\_\_\_
  - o instanceof BankAccount
  - \_\_\_\_\_
  - o instanceof CheckingAccount
  - \_\_\_\_\_
  - c instanceof BankAccount
  - \_\_\_\_\_
  - c instanceof CheckingAccount

# A dynamic binding problem

```
class A {  
    public void method1() { System.out.println("A1"); }  
    public void method3() { System.out.println("A3"); }  
}
```

```
class B extends A {  
    public void method2() { System.out.println("B2"); }  
    public void method3() { System.out.println("B3"); }  
}
```

```
A var1 = new B();  
Object var2 = new A();
```

```
var1.method1(); _____
```

```
var1.method2(); _____
```

```
var2.method1(); _____
```

```
((A) var2).method3(); _____
```

*OUTPUT???*

# The Object & Class Classes

**Object Class : Top class in Java**

**equals() method**

**boolean equals(Object obj)**

**getClass() method**

**Class getClass()**

**toString() method**

**String toString()**

**Class Class**

**getName() method**

**String getName()**

**getSuperClass() method**

**Class getSuperClass()**

**forName() method**

**Static Class forName (String cIsName) throws ClassNotFoundException**

# The Object & Class Classes

```
class ClassDemo {  
  
    public static void main(String args[]) {  
  
        Integer obj = new Integer(8);  
        Class cls = obj.getClass();  
        System.out.println(cls);  
    }  
}
```

Result :  
Class  
java.lang.Integer

# Object Oriented Programming: UML Modeling with Classes

# UML diagrams

## *Use case diagrams*

- Describe user tasks and points of contact with the system

## *Class diagrams*

- describe classes and their relationships

## *Sequence diagrams*

- show the behaviour of systems as interactions between objects

## *State diagrams and activity diagrams*

- show how systems behave internally

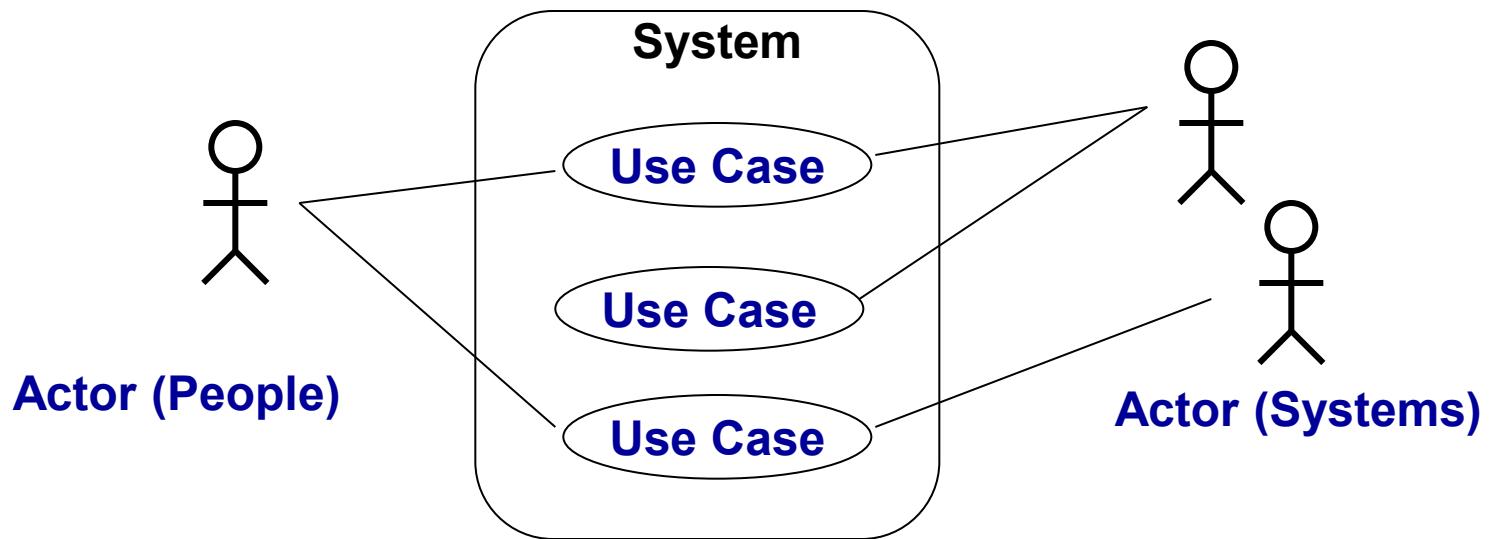
## *Component and deployment diagrams*

- “big picture” of how the components of a system are related

# Usecase diagram

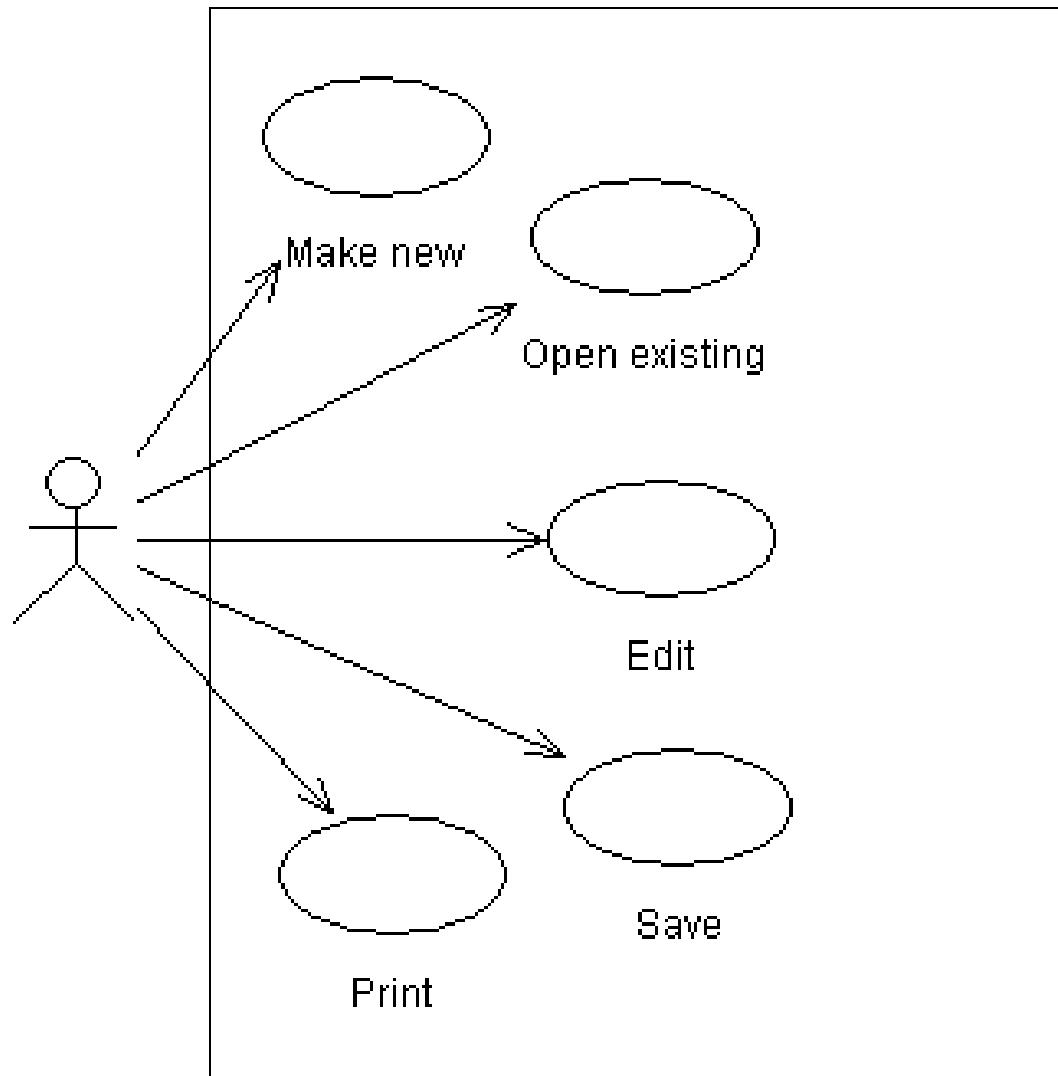
# Use Cases

- Two types of Actors: Users and System administrators



# Use case examples

(use cases for powerpoint.)

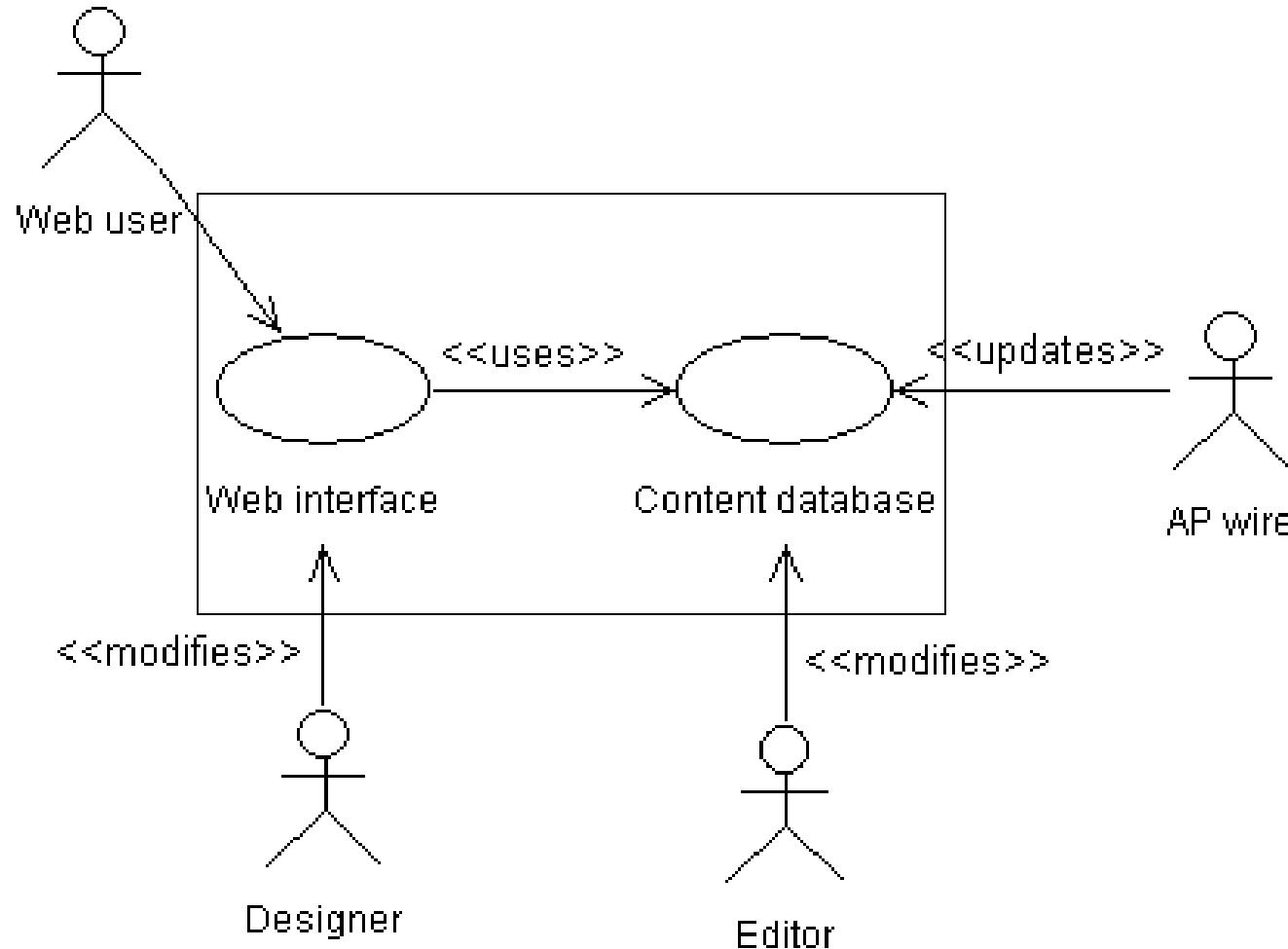


# About the last example...

- Gives a view of Powerpoint.
- focuses your attention to the key features
- HW: Make a detailed Use case for Powerpoint by clicking on the Menu items.
  - Can you find any items which are common to multiple Use cases?
  - Are there any Hierarchical Use cases?

# Use case examples

(Relationships in a news web site.)



# About the last example...

- The last is more complicated and realistic use case diagram. It captures several key use cases for the system.
- Note the multiple actors. In particular, 'AP wire' is an actor, with an important interaction with the system, but is not a person (or even a computer system, necessarily).
- The notes between <> marks are **stereotypes**
- HW: Make a detailed Use case for User login and a second database to store user profiles.

# Usecase diagram- ATM machine

- An automated teller machine (**ATM**) provides bank customers with access to financial transactions.
- *Customer* uses bank ATM to
  - *Check Balances*
  - *Deposit Funds,*
  - *Withdraw Cash*
  - *Transfer Funds*
- *ATM Technician* provides *Maintenance and Repairs.*
- *Bank* actor: customer transactions or to the ATM servicing.

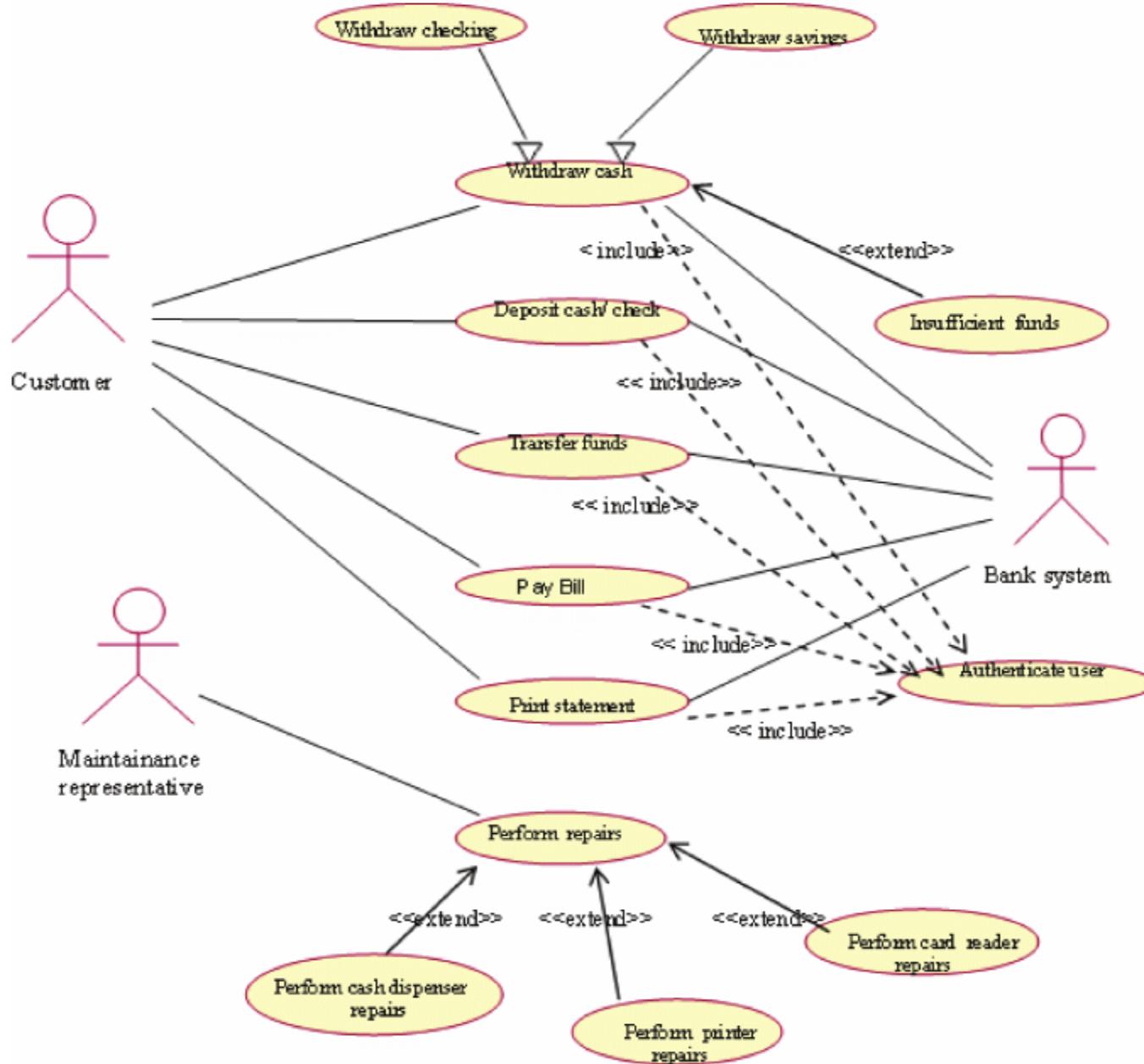
HW: Make a detailed Use case for the ATM based on the diagram in the next 2 slides.

Can you find any items which are common to multiple Use cases ( eg.  
Withdraw+Deposit= Transfer)

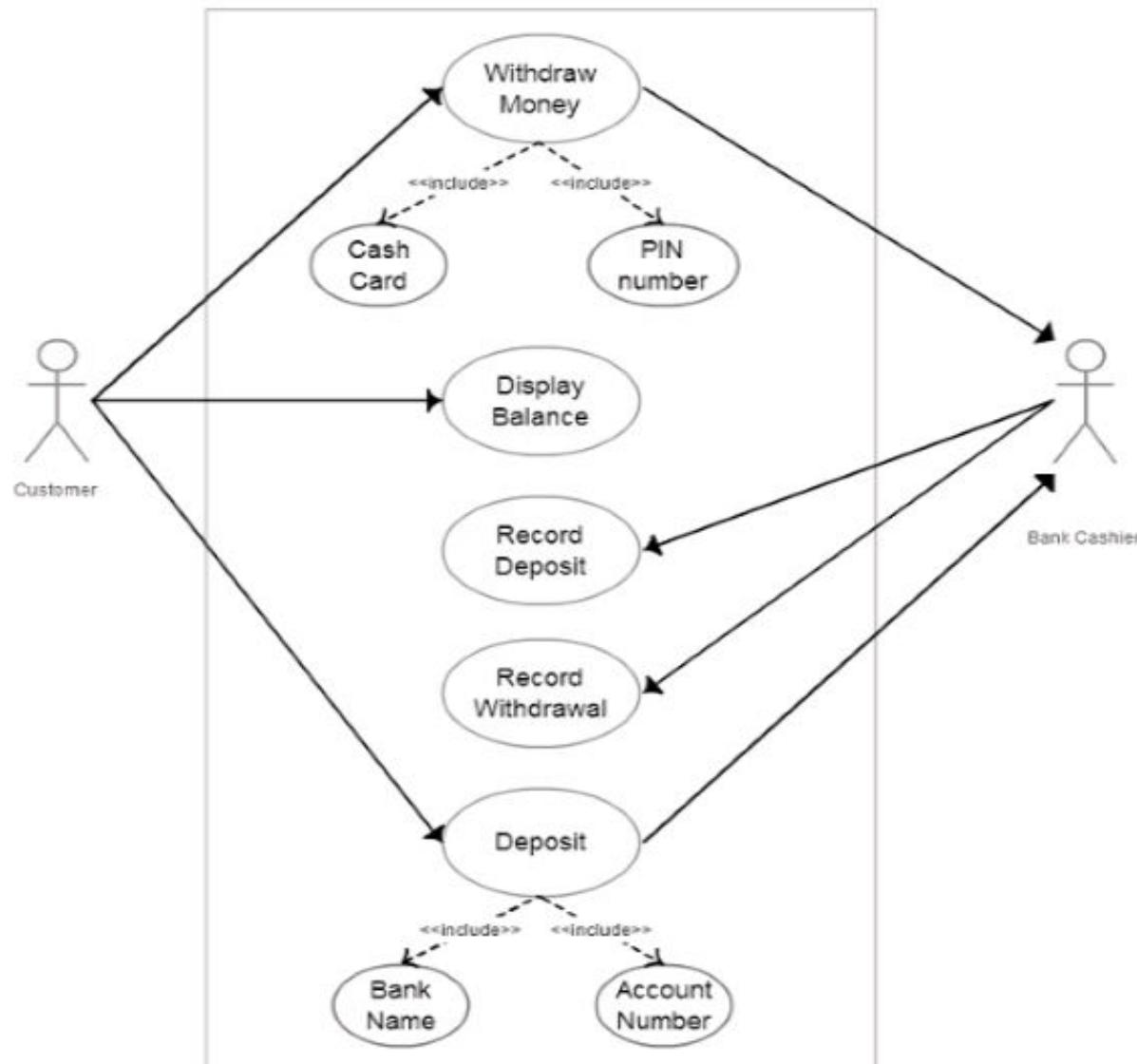
Are there any more Hierarchical Use cases?

(A few of you, **NOT ALL** can go to the ATM machine and get the details)

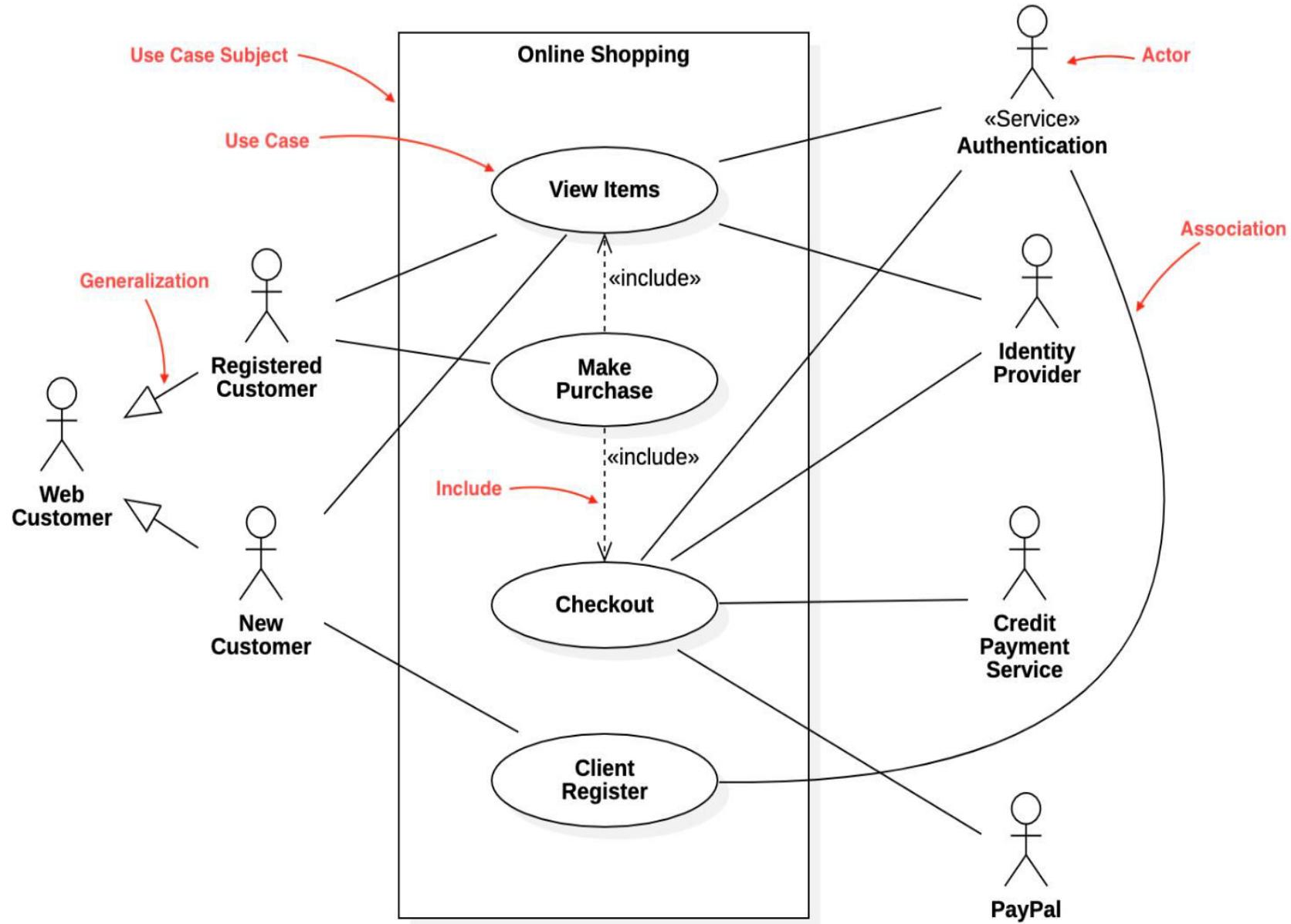
# Usecase diagram for an ATM machine



# Usecase diagram for an ATM machine



# Usecase diagram for an Online Purchase

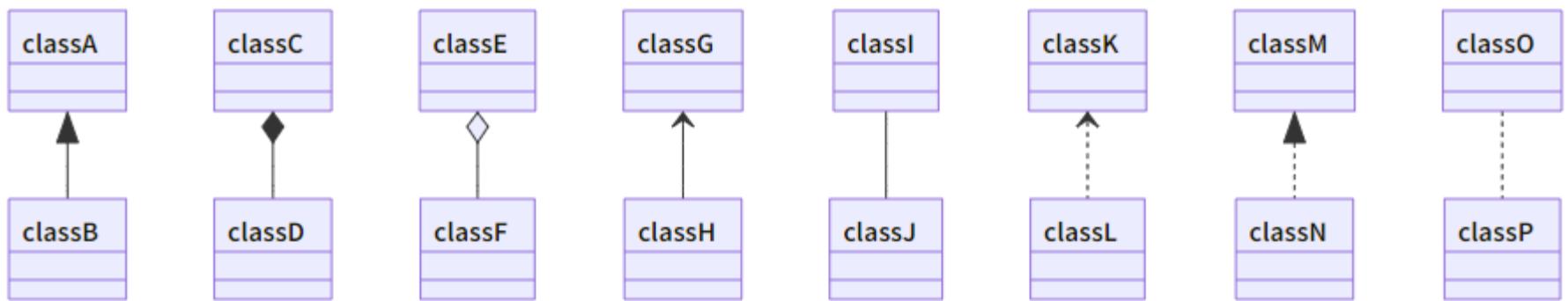


# Association with Stereotypes: Uses, Extends and Includes

- Uses or modifies or updates : (Side 7)
  - Direction indicates an entity doing the action on another entity eg: Designer <<modifies>> a Web interface
  - Open arrow is used

# Relations between entities

|                    |                      |                            |
|--------------------|----------------------|----------------------------|
| classA < -- classB | Super Class (A)      | Sub Class (B)              |
| classC *-- classD  |                      | Composition                |
| classE o-- classF  |                      | Aggregation                |
| classG <-- classH  | Directed             | Association                |
| classI -- classJ   |                      | Association                |
| classK <.. classL  | Directed association | between Interfaces         |
| classM < .. classN | M is implemented by  | class N                    |
| classO .. classP   | Association          | between Interfaces classes |



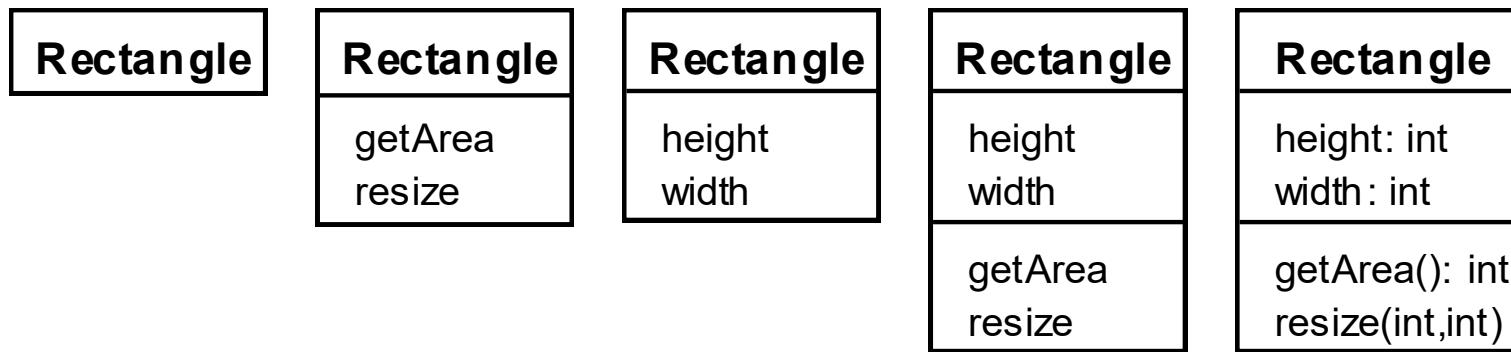
# Class Diagrams

# Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - the functions performed by classes and their instances
  - *Associations*
    - represent linkages between instances of classes
  - *Generalizations*
    - group classes into inheritance hierarchies

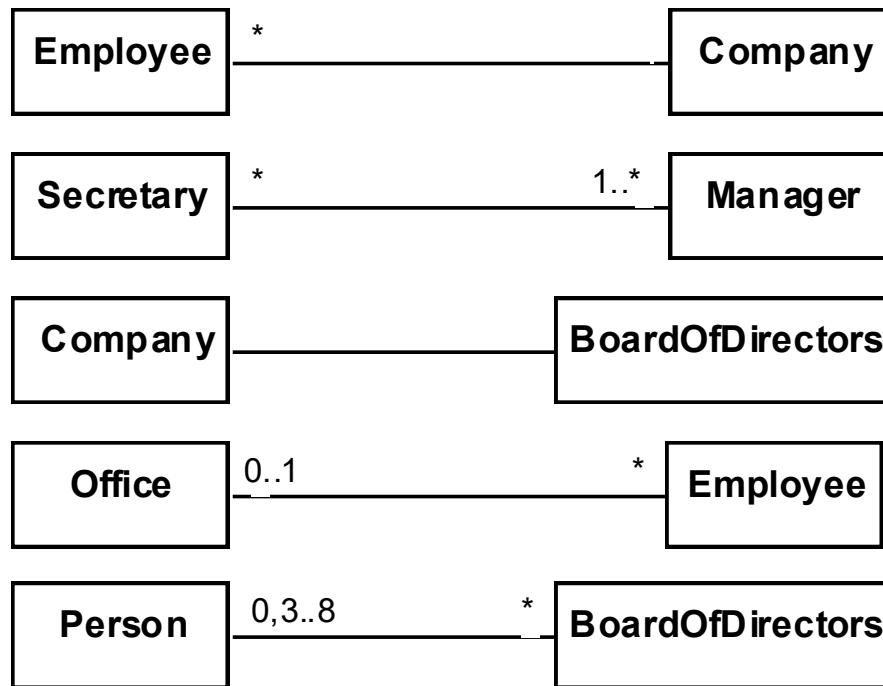
# Classes

- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The *UML signature* of an operation is:  
operationName(parameterName: parameterType ...): returnType



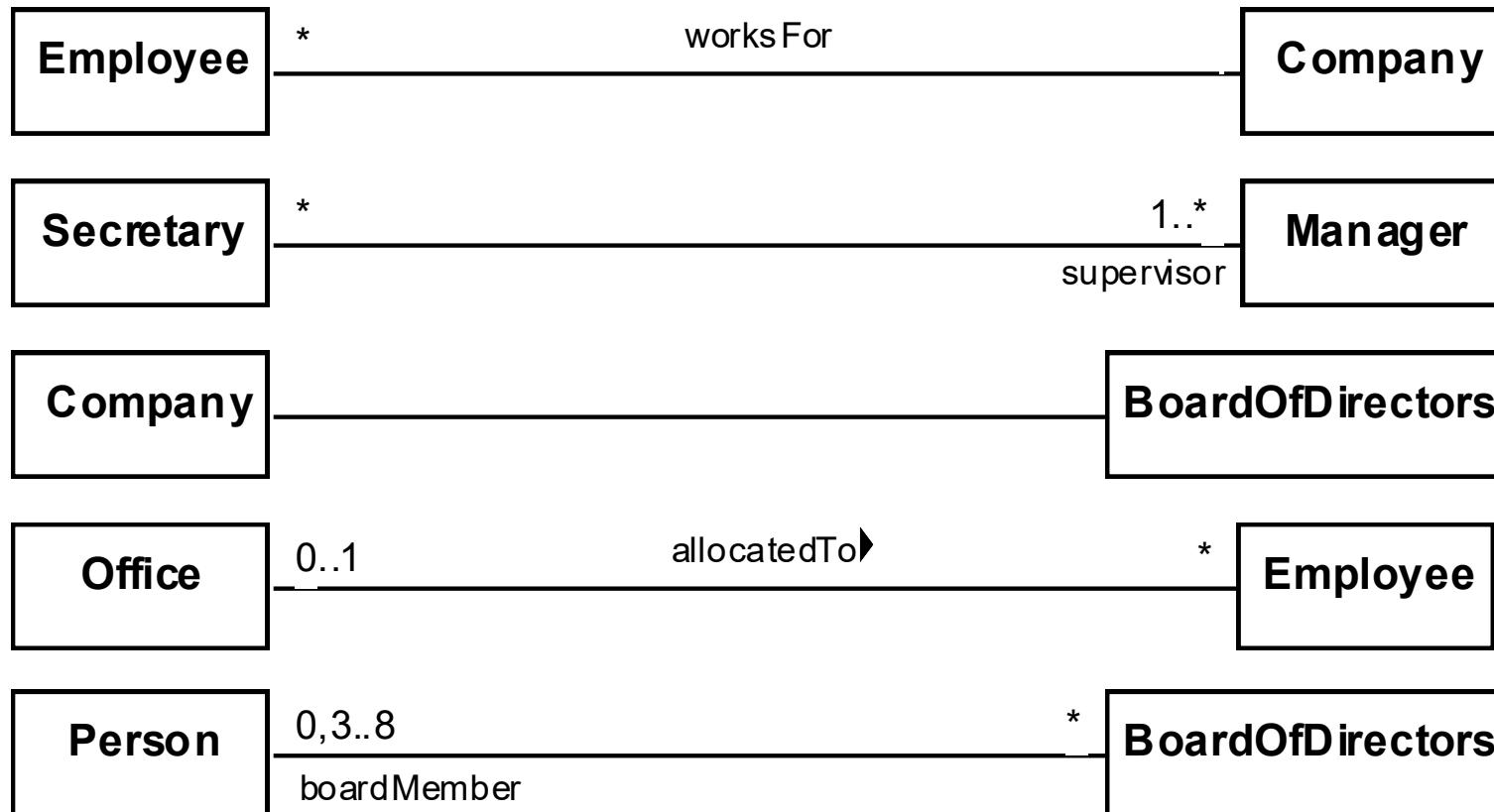
# Associations and Multiplicity

- An *association* is a line that relates two classes
- Symbols indicating *multiplicity* are shown at each end



# Labelling associations

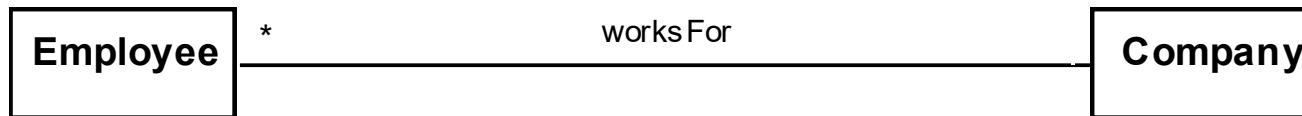
An association can be *labeled*, to clarify its nature



# Interpreting associations

## Many-to-one

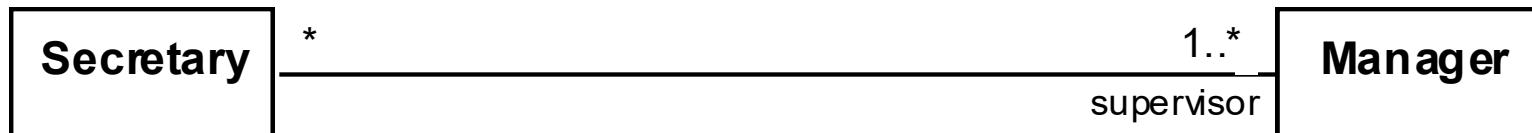
- A company has many employees,
- An employee can only work for one company.
  - No moonlighting!
- A company can have zero employees
  - E.g. a ‘shell’ company
- Every employee must work for some company



# Interpreting associations

## Many-to-many

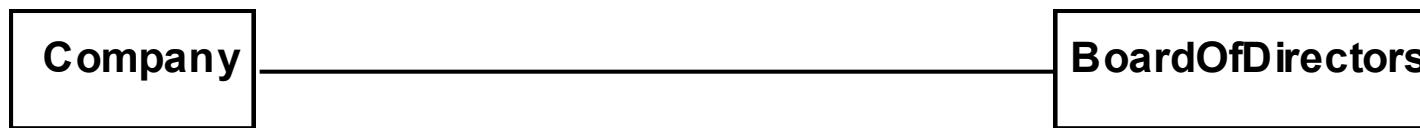
- A secretary can work for many managers
- A manager can have many secretaries
- Secretaries can work in pools
- Managers can have a group of secretaries
- Some managers might have zero secretaries.
- Is it possible for a secretary to have, perhaps temporarily, zero managers?



# Interpreting associations

## One-to-one

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company



# Another example

A booking is always for exactly one passenger

- no booking with zero passengers
- a booking could *never* involve more than one passenger.

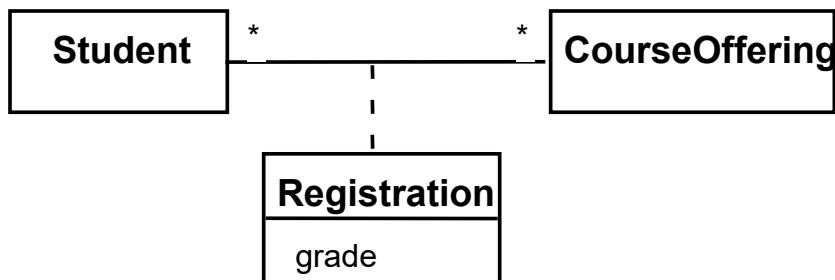
A Passenger can have any number of Bookings

- a passenger could have no bookings at all
- a passenger could have more than one booking



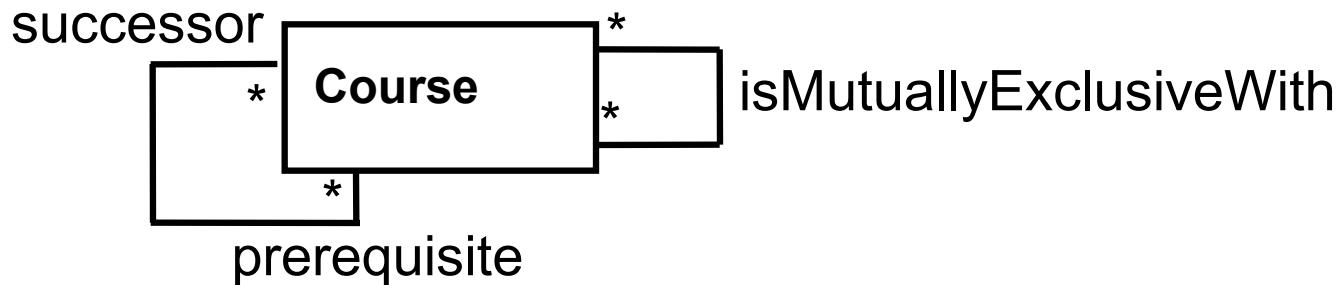
# Association classes

Sometimes, an attribute shared by two associated classes cannot be placed in either one. E.g., the following are equivalent:



# Reflexive associations

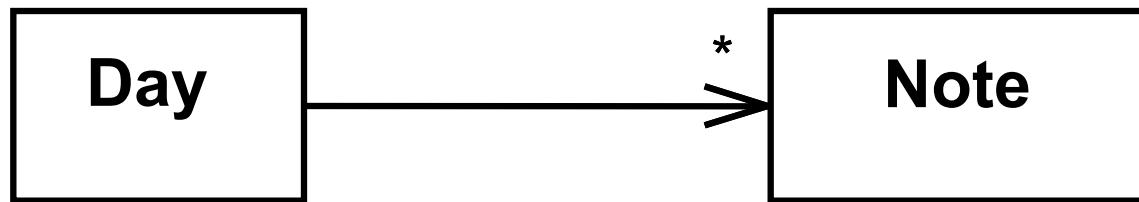
An association can connect a class to itself



# Directionality in associations

Associations are by default *bi-directional*

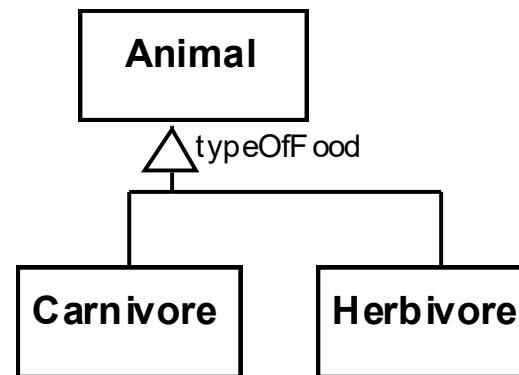
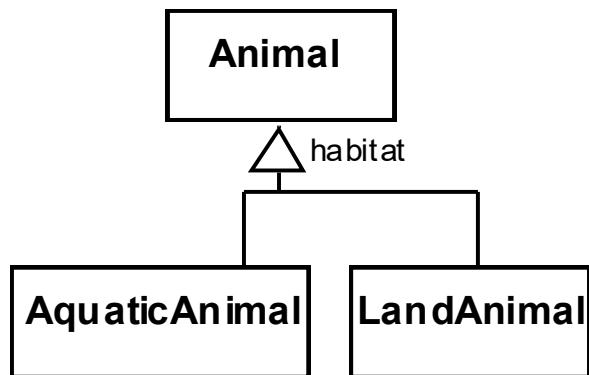
The direction can be limited by adding an arrow:



# Generalization

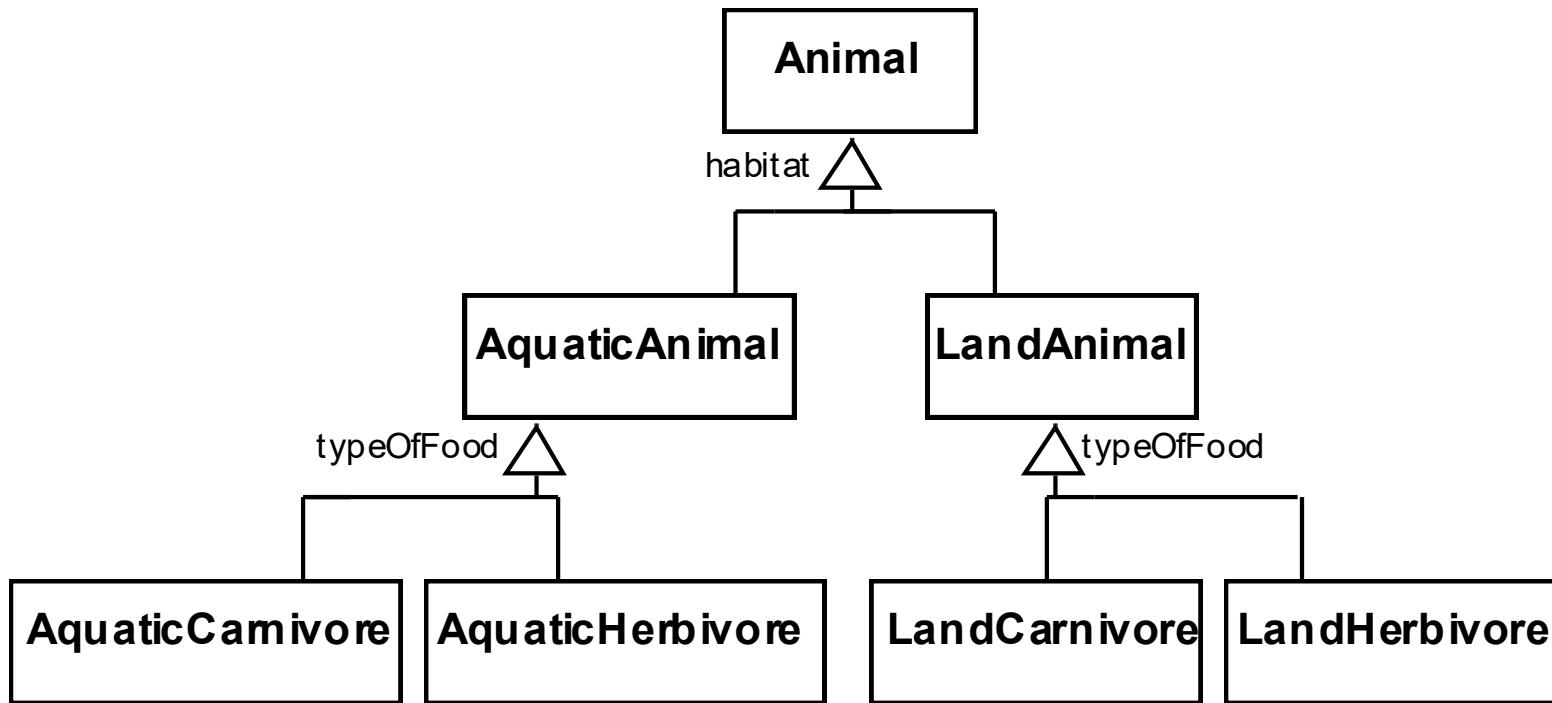
- Specializing a superclass into two or more subclasses

The *label* that describes the criterion for specialization



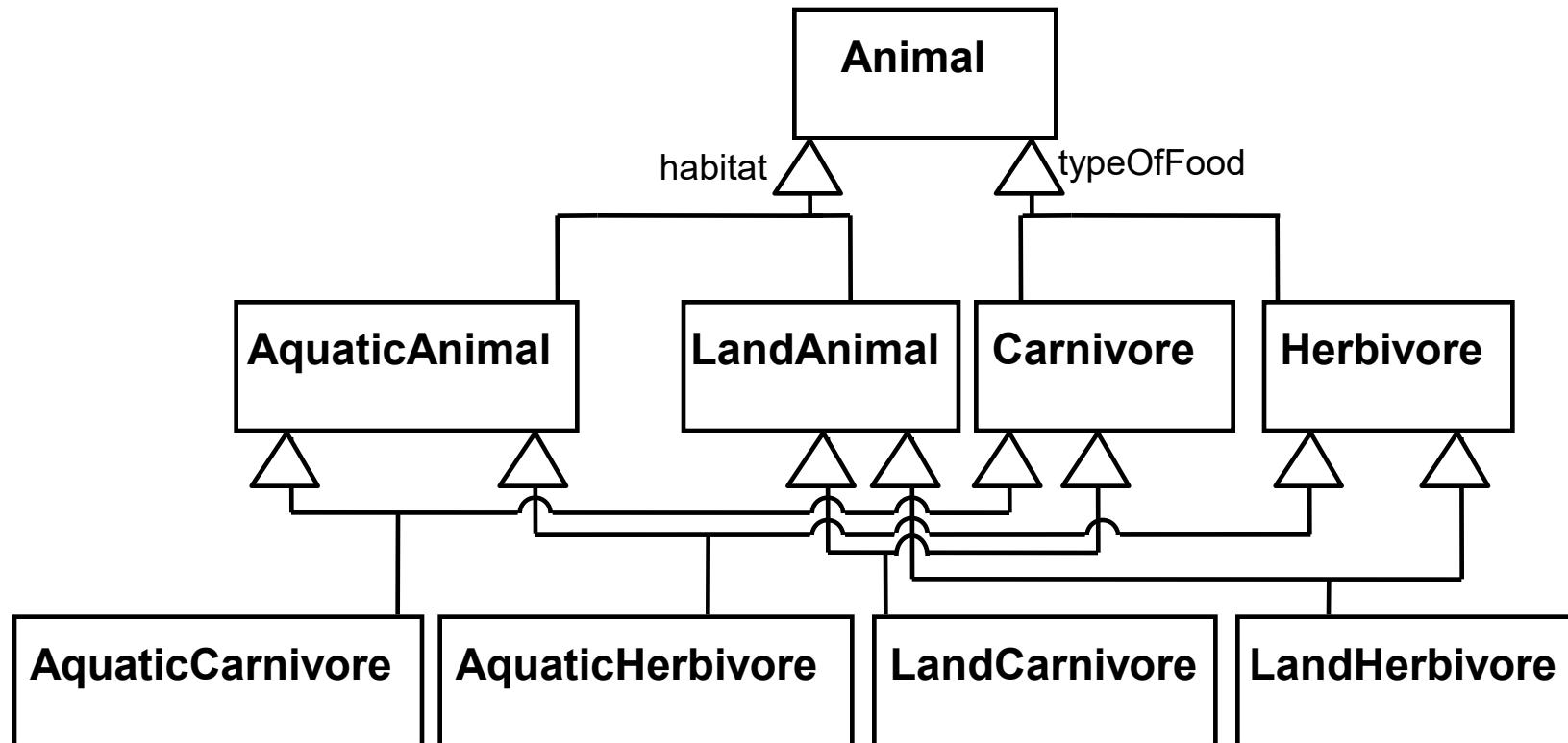
# Handling multiple discriminators

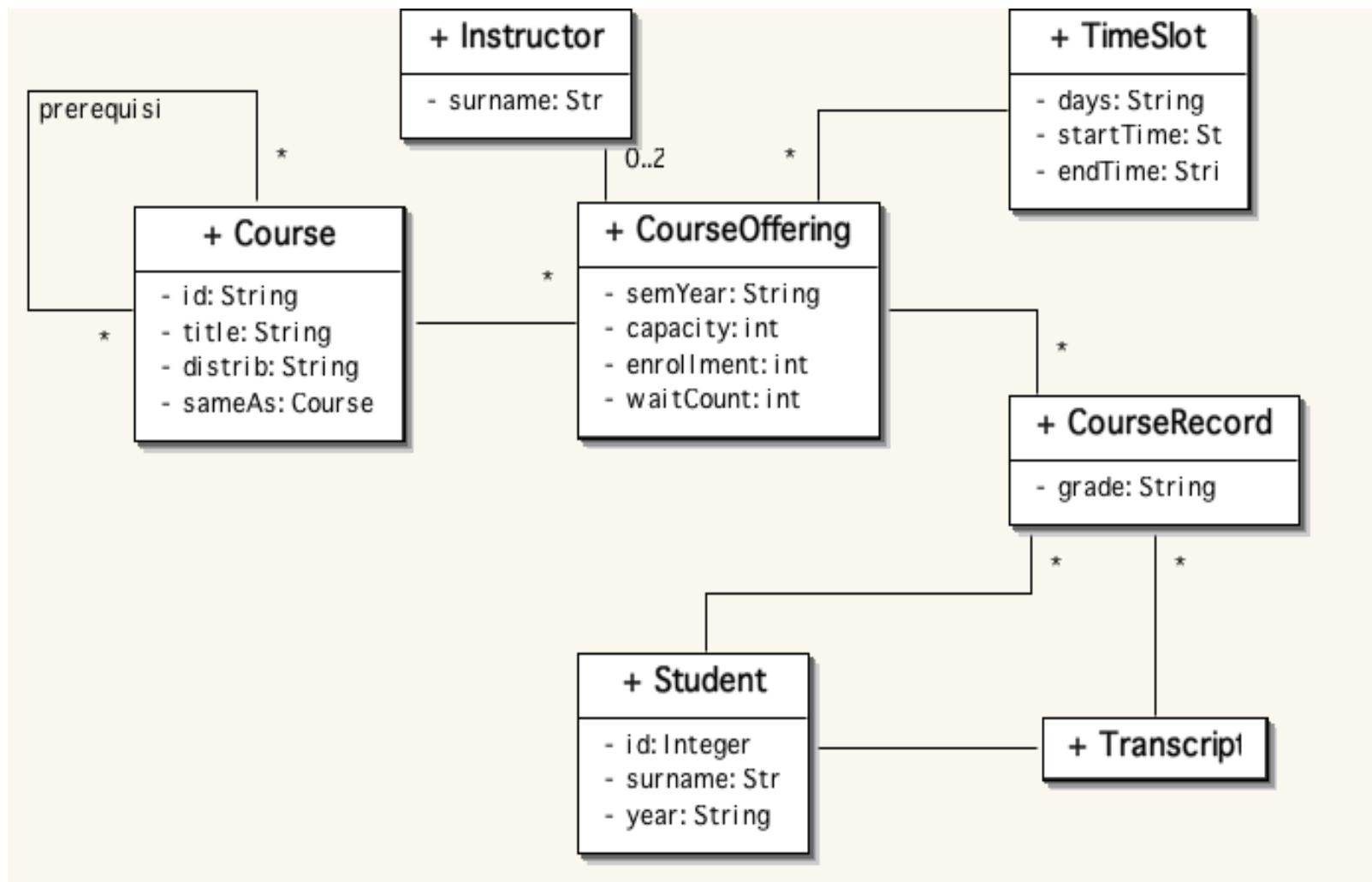
Creating higher-level generalization



# Handling multiple discriminators

## Using multiple inheritance





# Visibility of Class Members

|                  |   |                                                                           |
|------------------|---|---------------------------------------------------------------------------|
| <b>public</b>    | + | anywhere in the program and may be called by any object within the system |
| <b>private</b>   | - | the class that defines it                                                 |
| <b>protected</b> | # | (a) the class that defines it or<br>(b) a subclass of that class          |
| <b>package</b>   | ~ | instances of other classes within the same package                        |

# Association

- An association is a relationship between two separate classes. It joins two entirely separate entities.
- There are four different types of association:
  - Bi-directional
  - Uni-directional
  - Aggregation (includes composition aggregation)
  - Reflexive.
- Bi-directional and uni-directional associations are the most common ones.
- This can be specified using multiplicity (one to one, one to many, many to many, etc.).
- Implementation in Java: use of an instance field. The relationship can be bi-directional with each class holding a reference to the other.

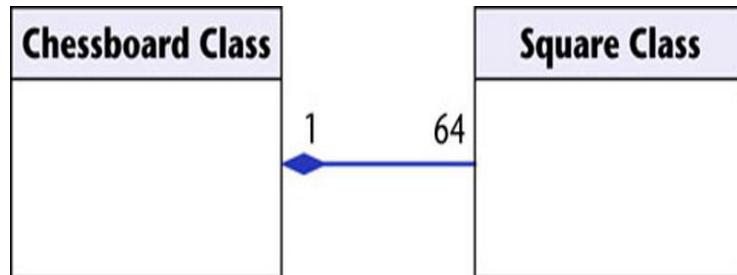
# Association



```
Class Student {  
    Course enrolls[40];  
}
```

```
Class Course {  
    Student has[];  
}
```

# Composition



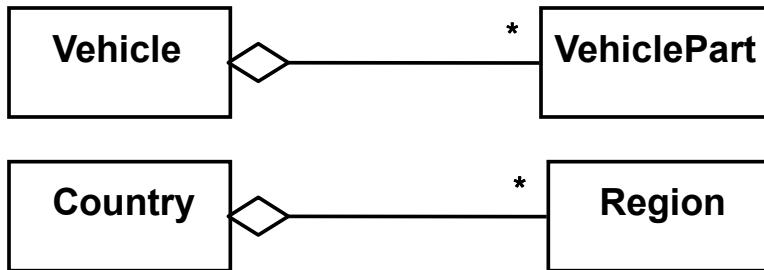
```
Class Chessboard {  
    Square[64];  
}
```

```
Class Square {  
    Color c;  
}
```

# Aggregation

Aggregations are associations that represent ‘part-whole’ relationships.

- The ‘whole’ side is often called the *assembly* or the *aggregate*
- This symbol is a shorthand notation association named `isPartOf`

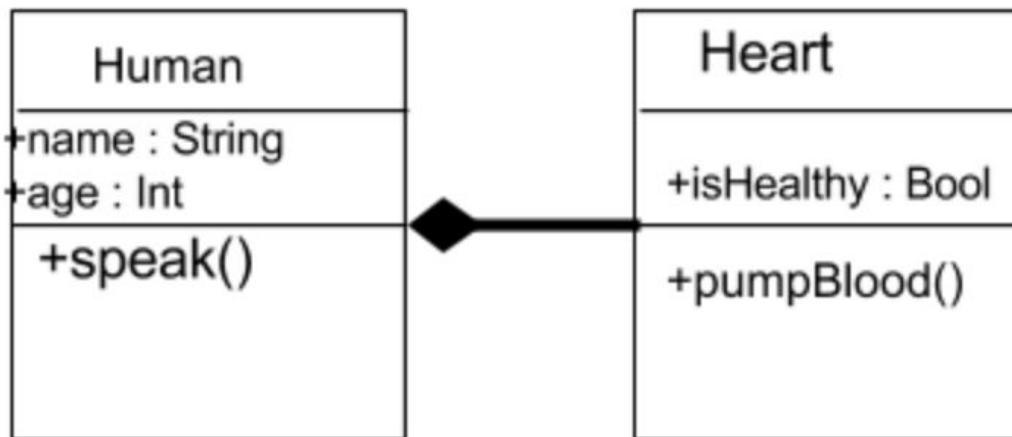
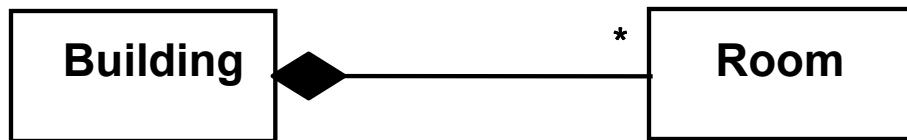


# When to use an aggregation

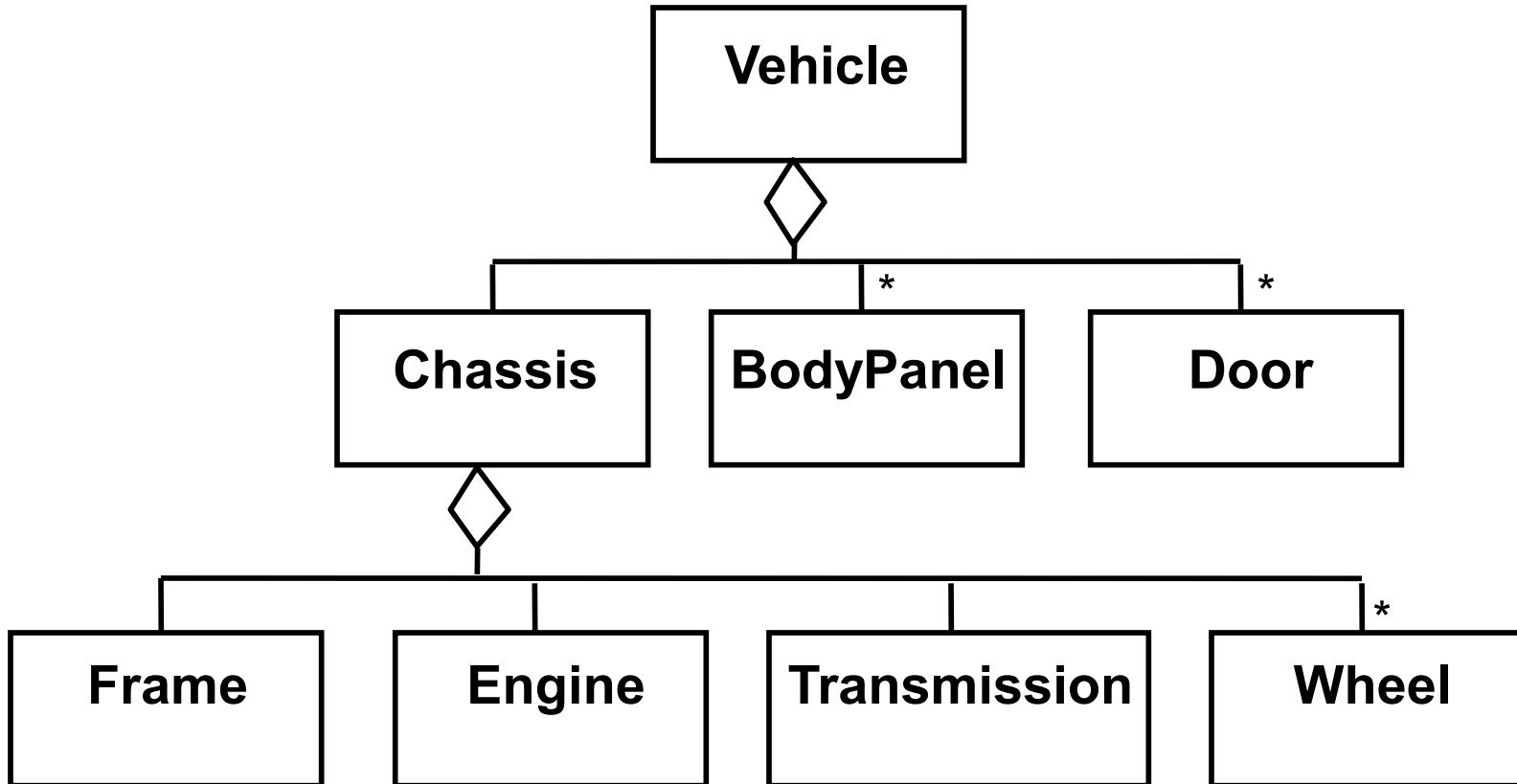
- An association is an aggregation if:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts

# Composition

- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, the parts are also destroyed. Highly dependent on each other.

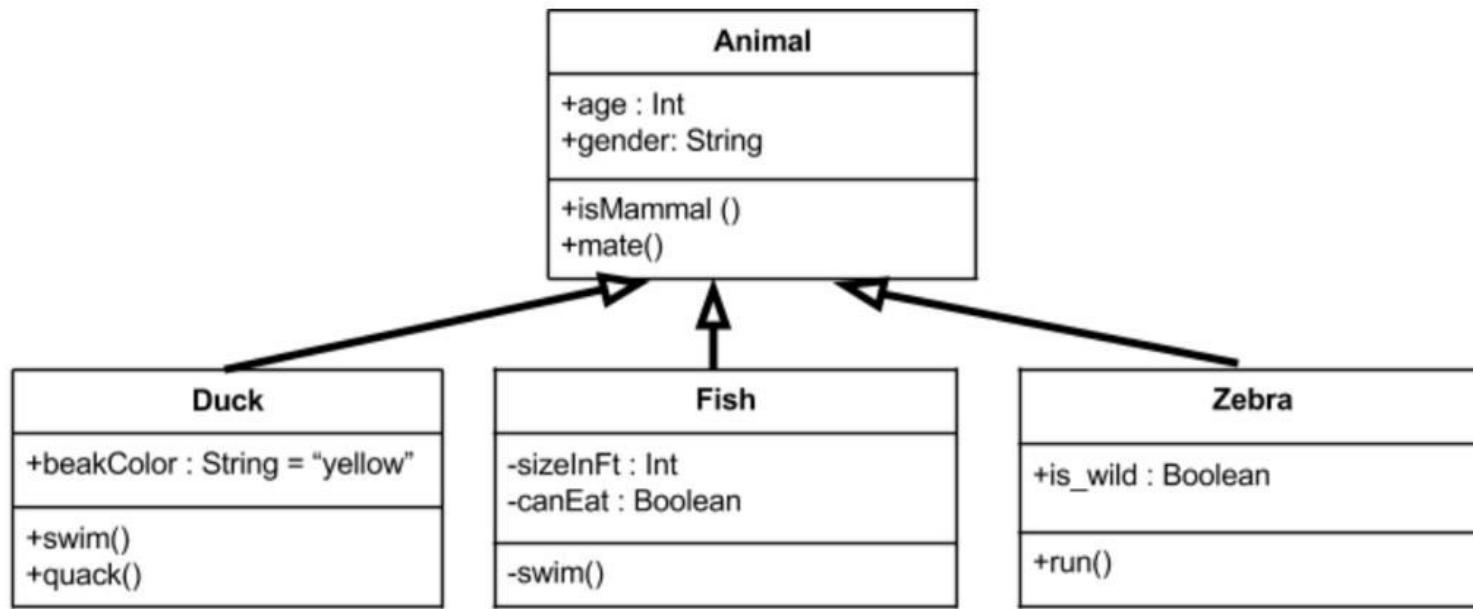


# Aggregation hierarchy



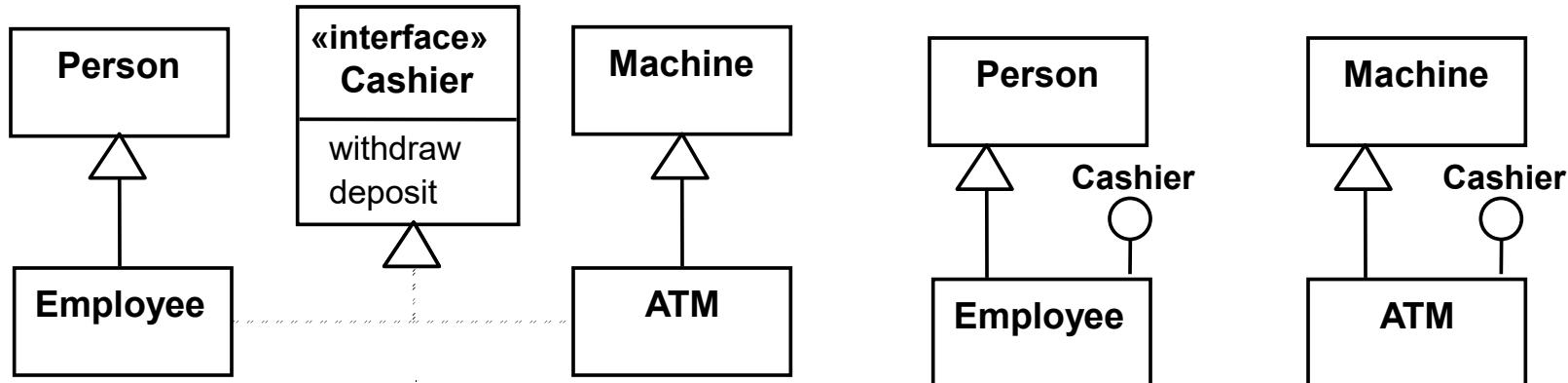
# Inheritance

- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).



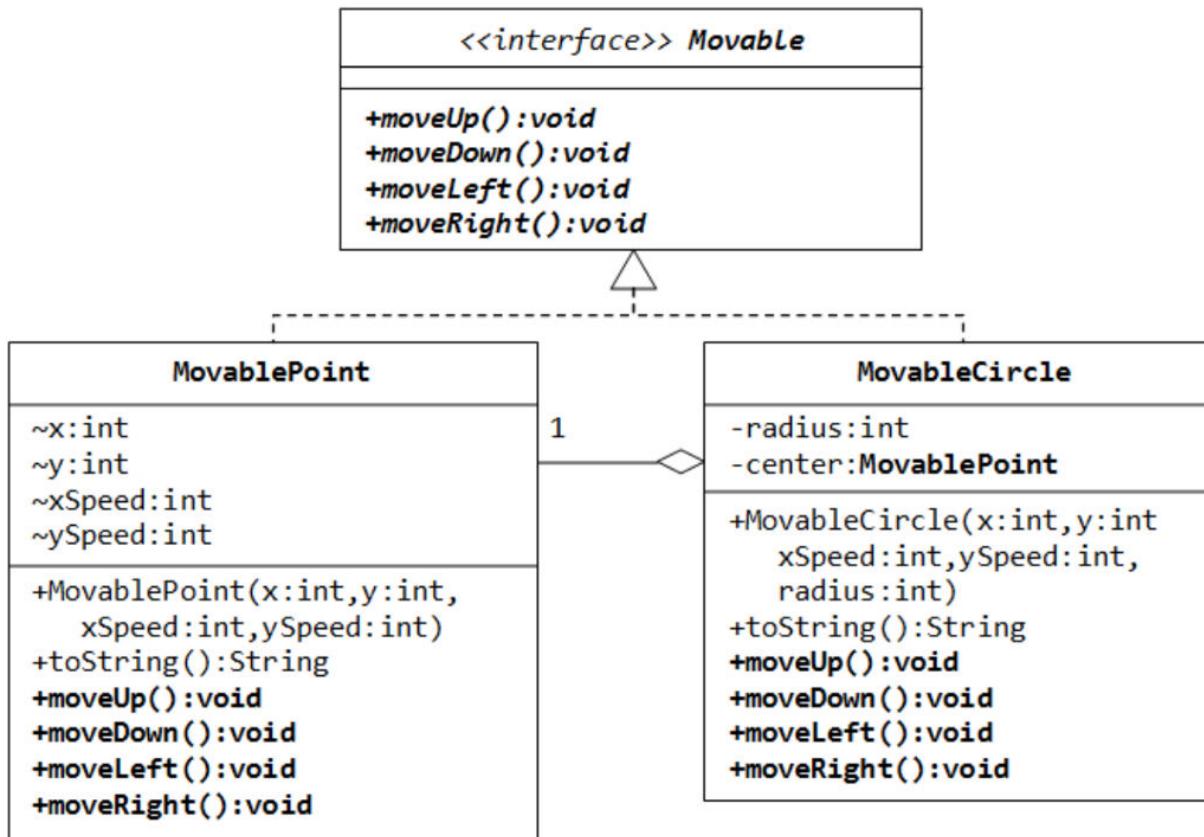
# Interfaces

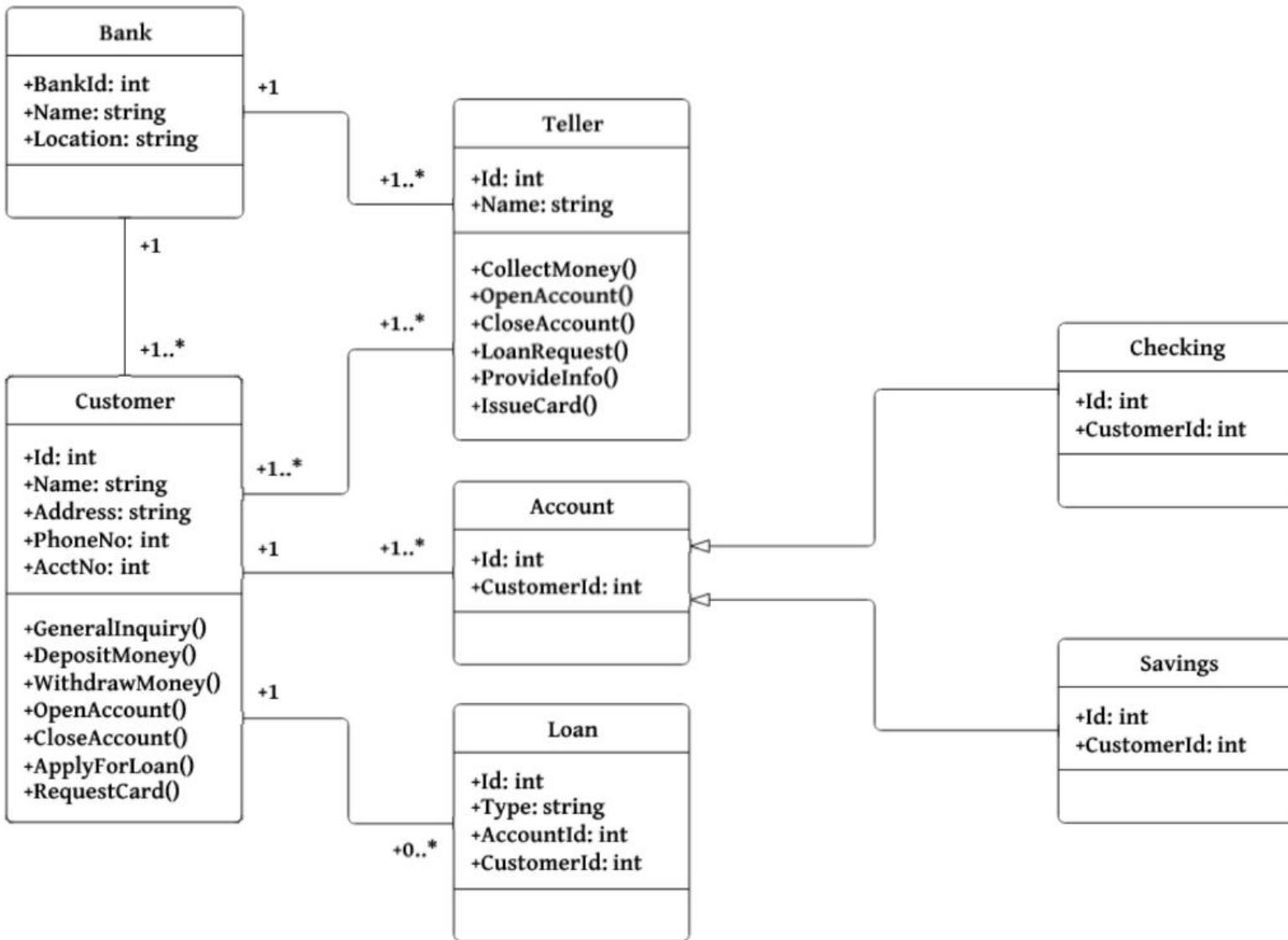
- An interface describes a *portion of the visible behaviour* of a class.
- An *interface* is similar to a class, except it lacks instance variables and method bodies

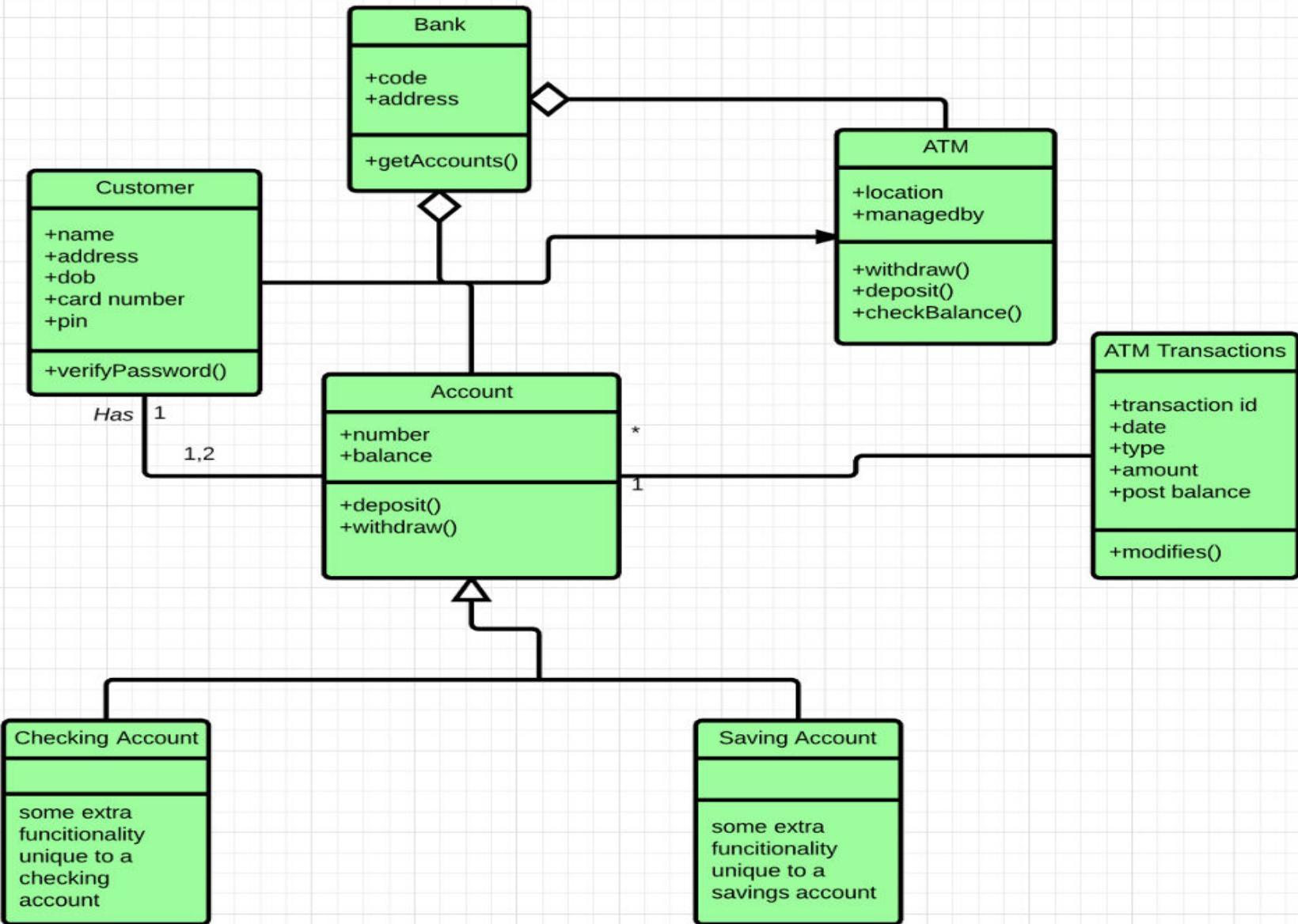


# Interface

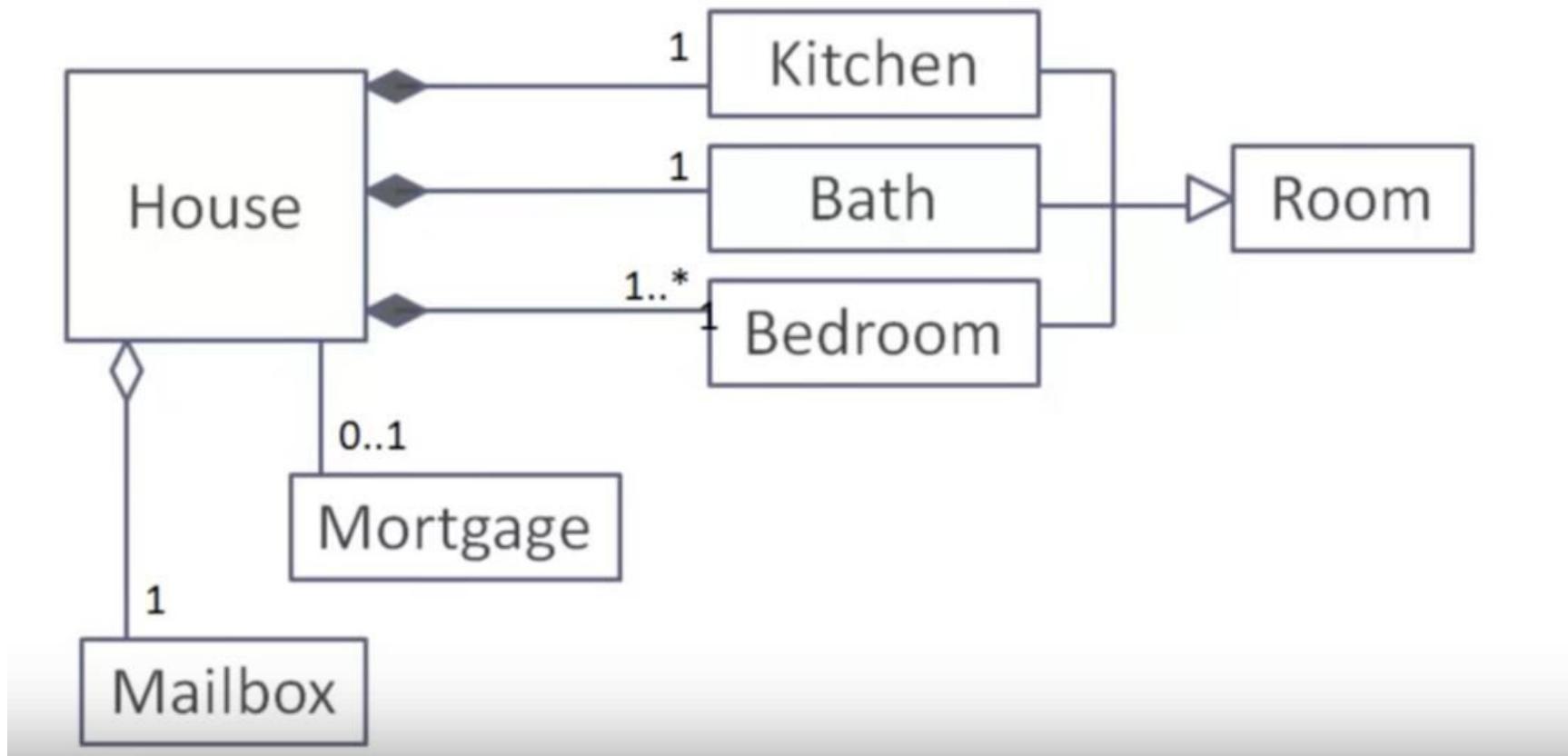
- A relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies







- A house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one).



# Modelling Interactions and Behaviour

# Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects of a software system
  - They help you to visualize how the system runs.
  - An interaction diagram is often built from a use case and a class diagram.
    - The objective is to show how a set of objects accomplish the required interactions with an actor.

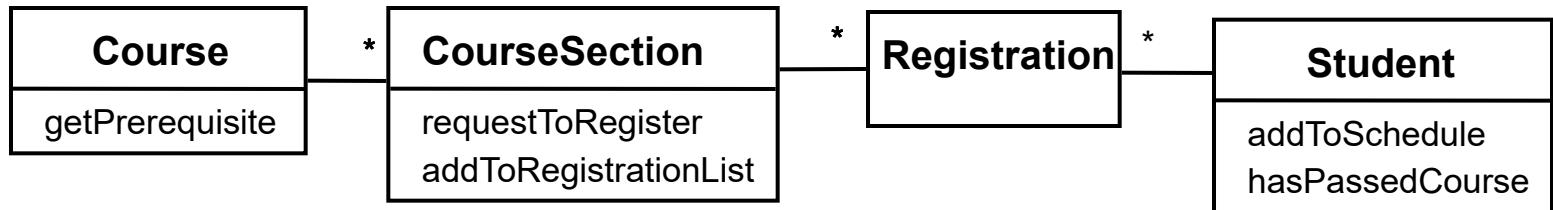
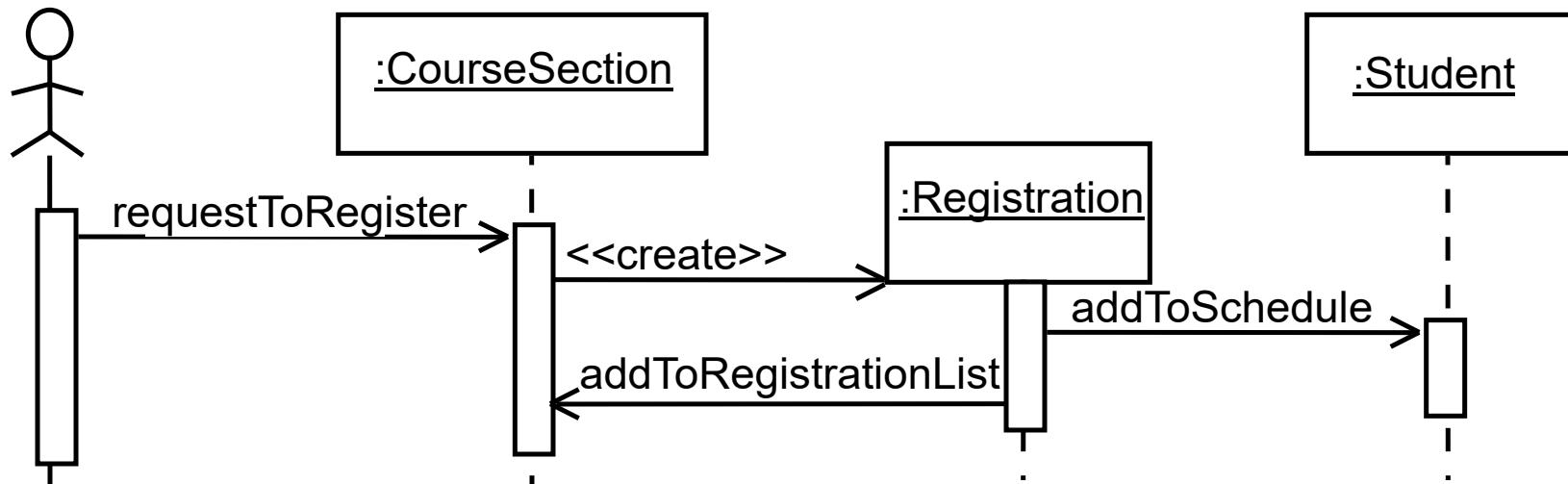
# Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called an *interaction*.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.

# Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object

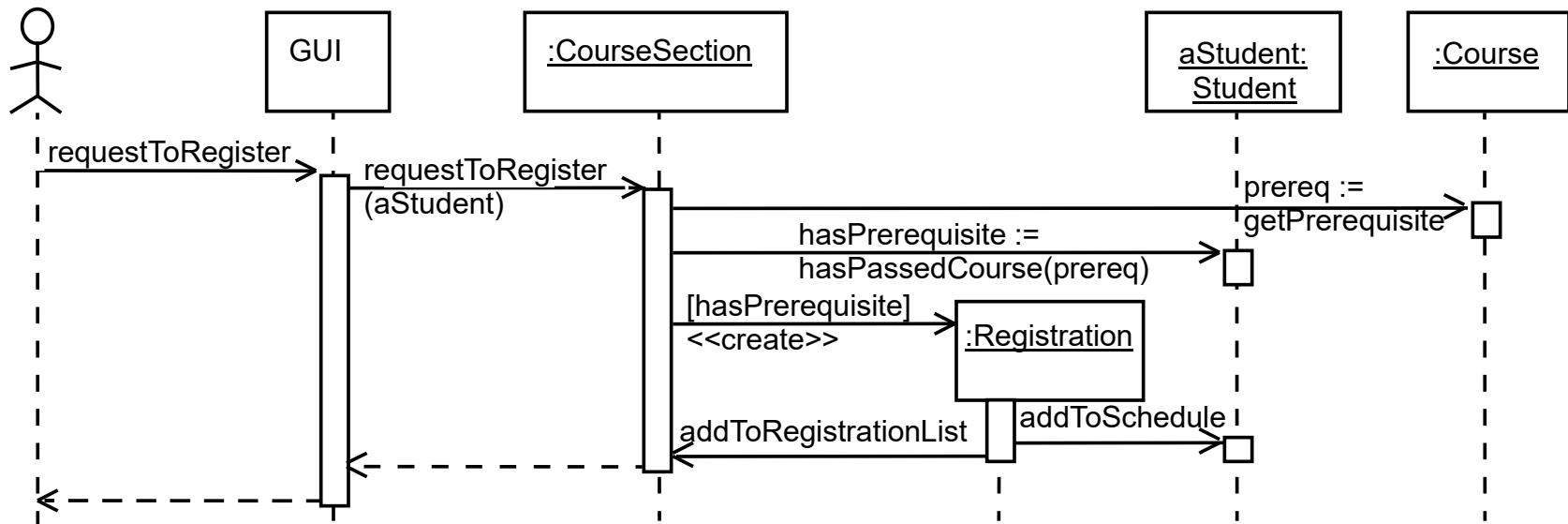
# Sequence diagrams – an example



# Sequence diagrams

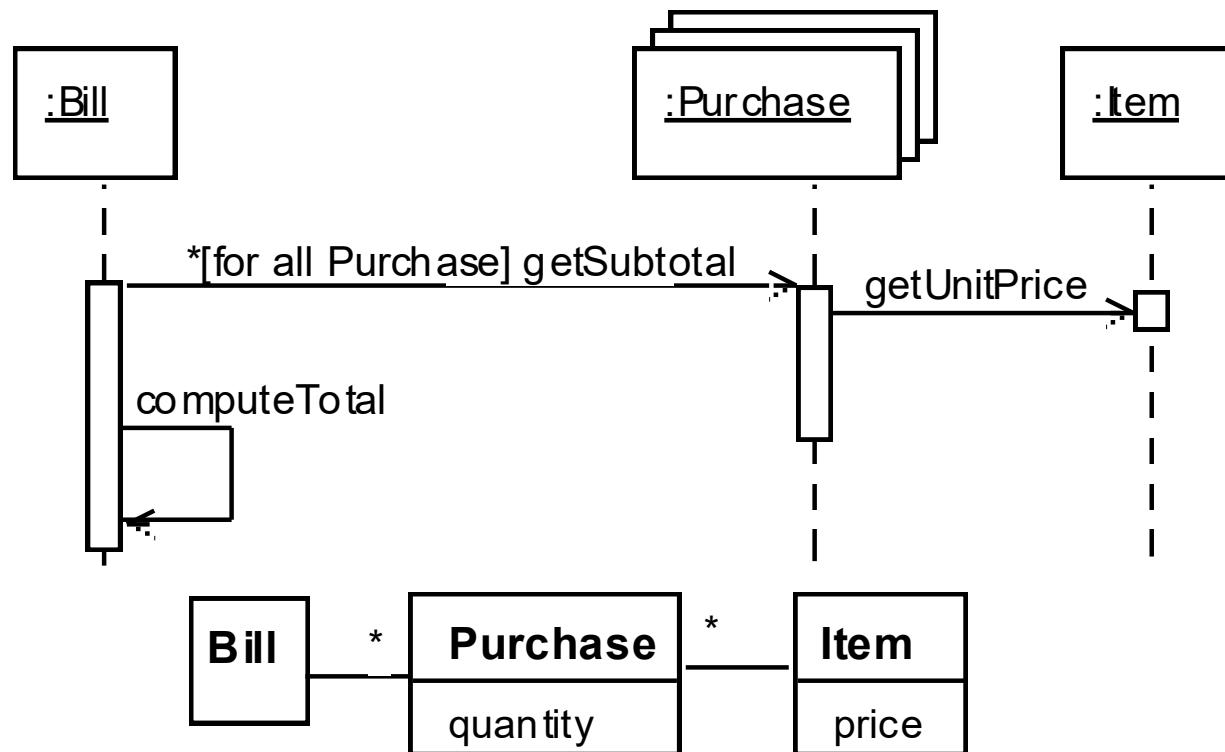
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.

# Sequence diagrams – same example, more details



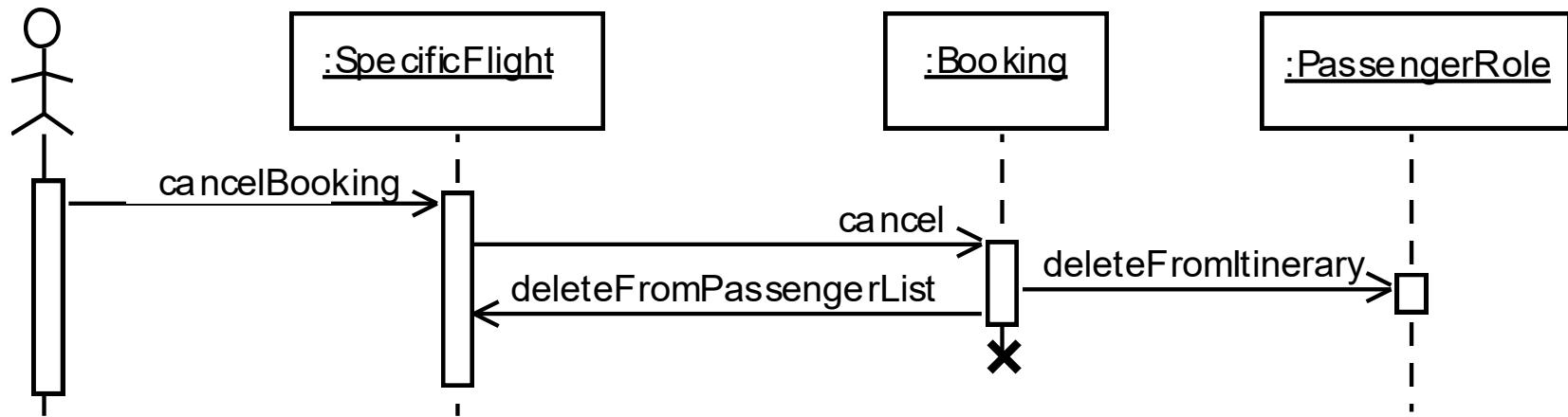
# Sequence diagrams

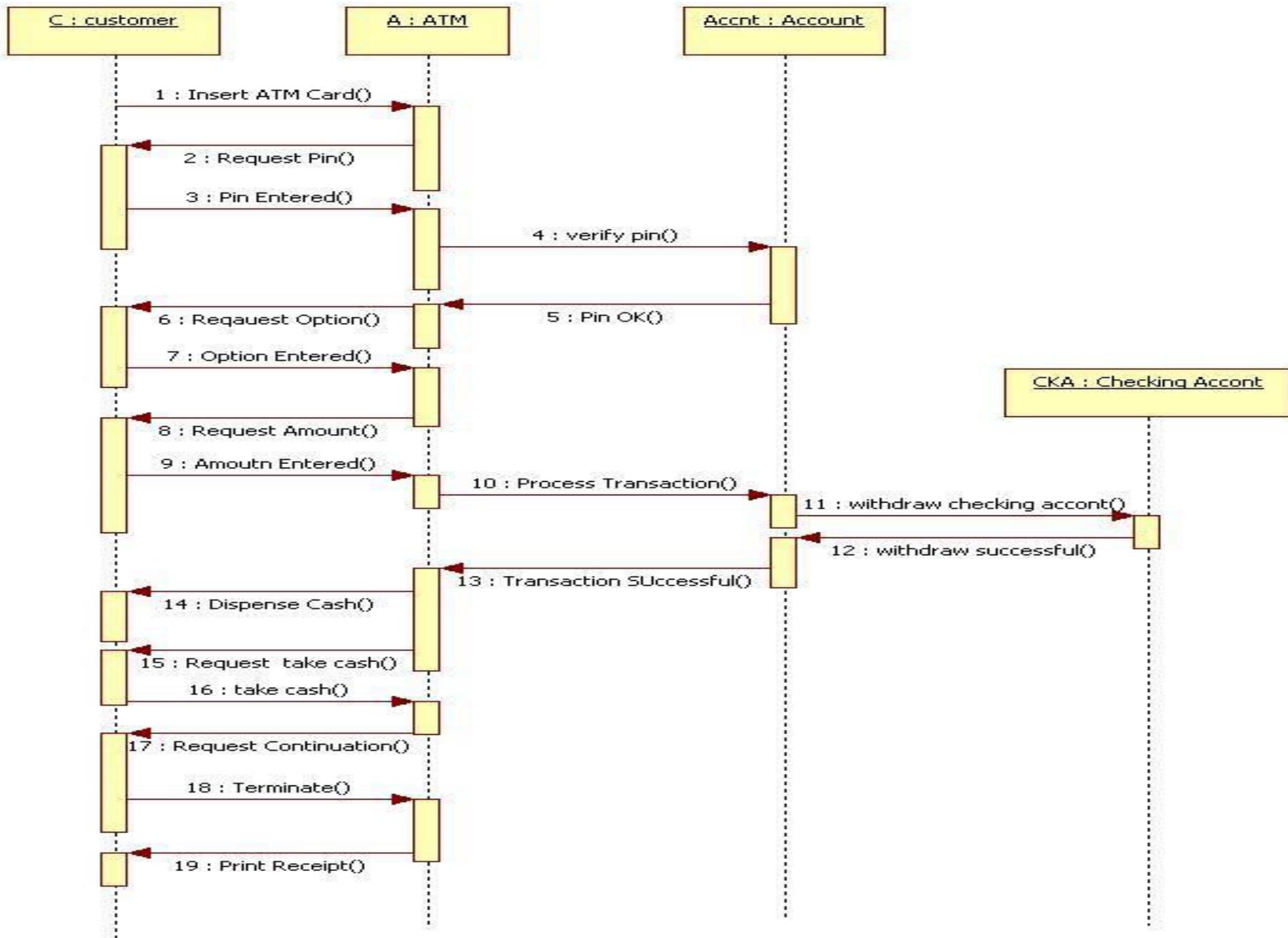
- An *iteration* over objects is indicated by an asterisk preceding the message name



# Object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline

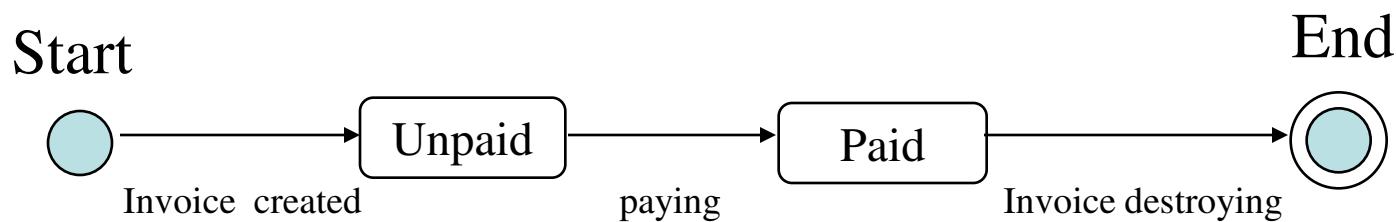




# Statechart diagram

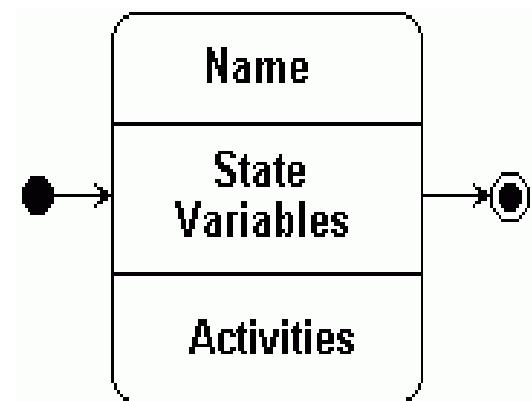
# State Diagrams

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



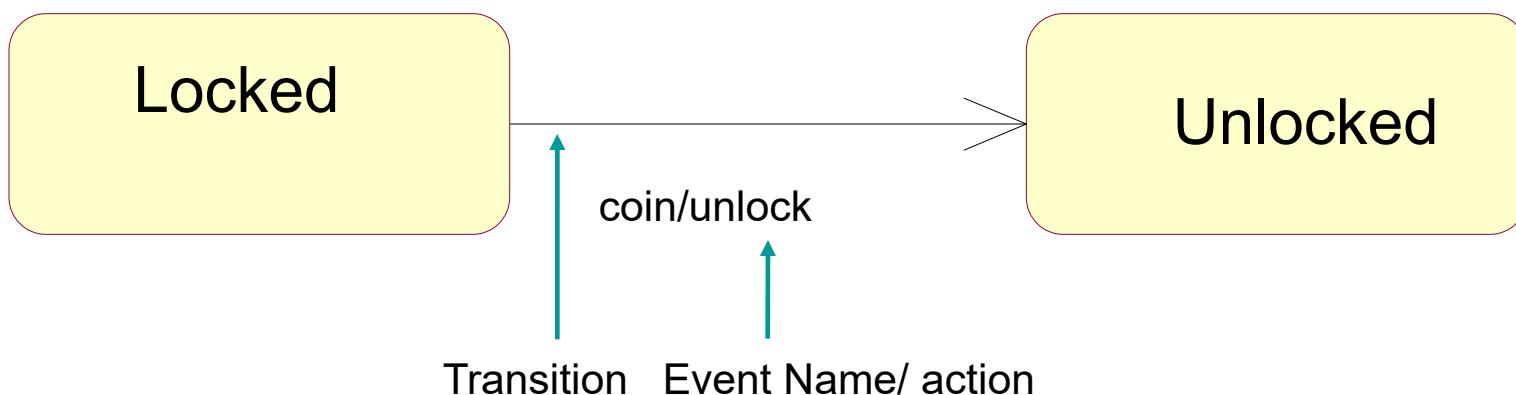
# Special States

- The initial state is the state entered when an object is created.
  - An initial state is mandatory.
  - Only one initial state is permitted.
  - The initial state is represented as a solid circle.
- A final state indicates the end of life
  - A final state is optional.
  - A final state is indicated by a bull's eye.
  - More than one final state may exist.



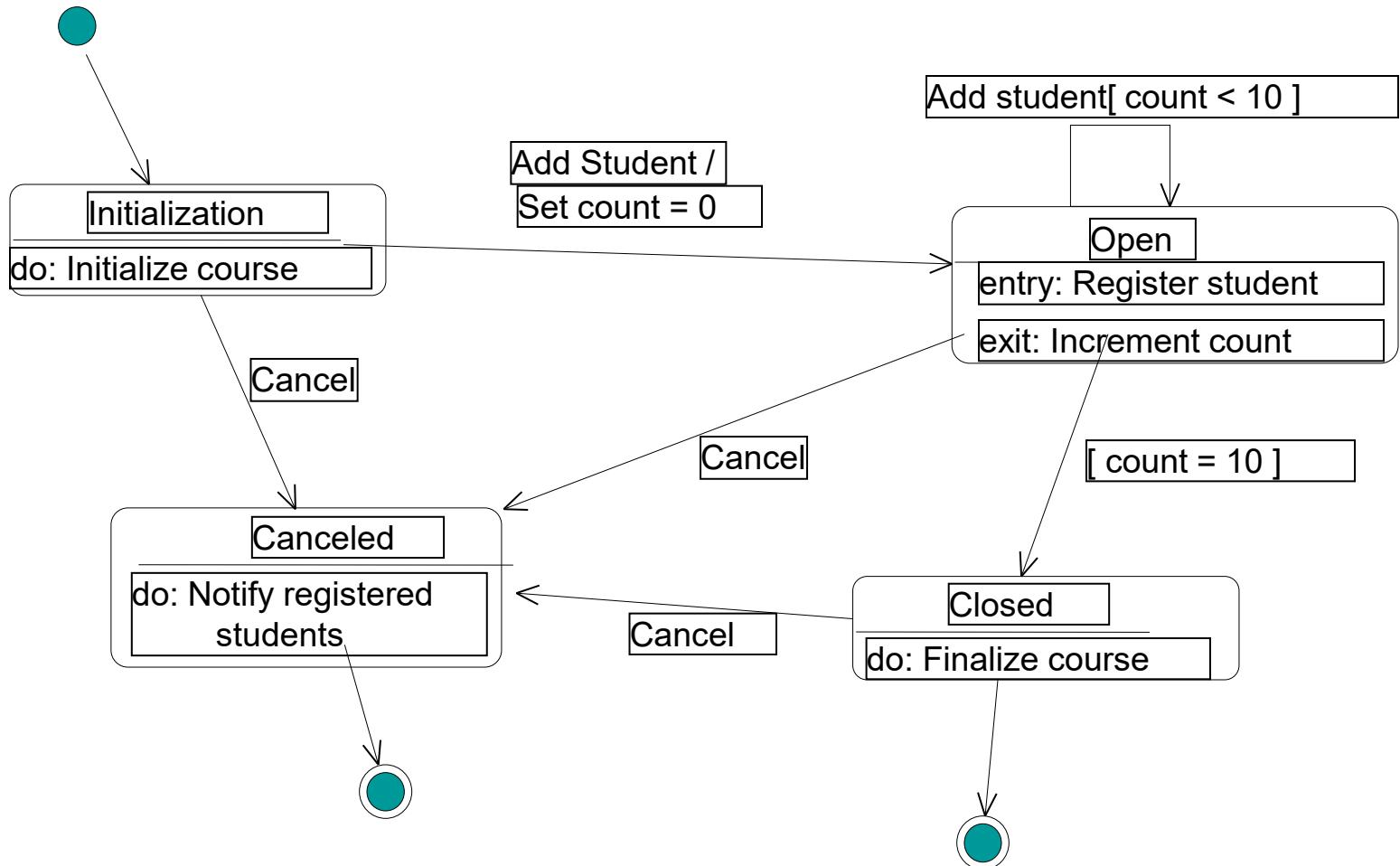
# Events, Actions & Transitions

- An event: stimulus that can trigger a state transition.
- A transition: is a change from an originating state to a successor state as a result of some stimulus.
  - The successor state could possibly be the originating state.
- A transition may take place in response to an event.
- Transitions can be labeled with event names.



# Statechart Diagram

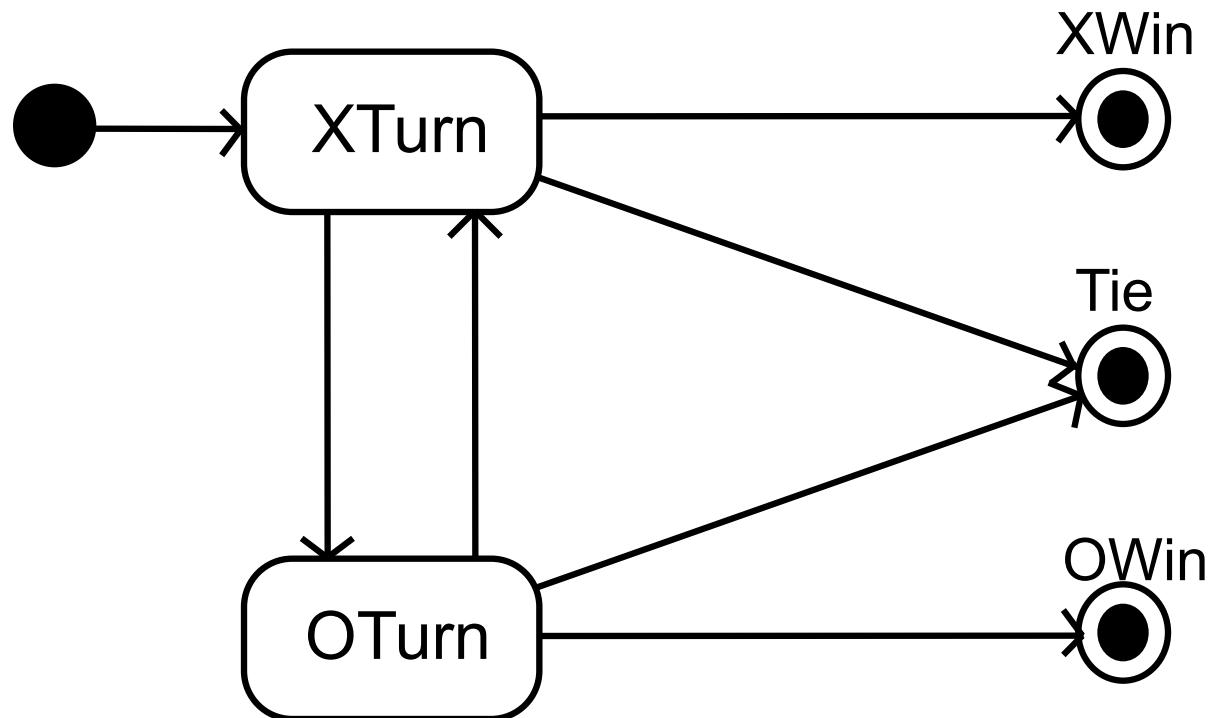
- A statechart diagram shows the lifecycle of a single class.



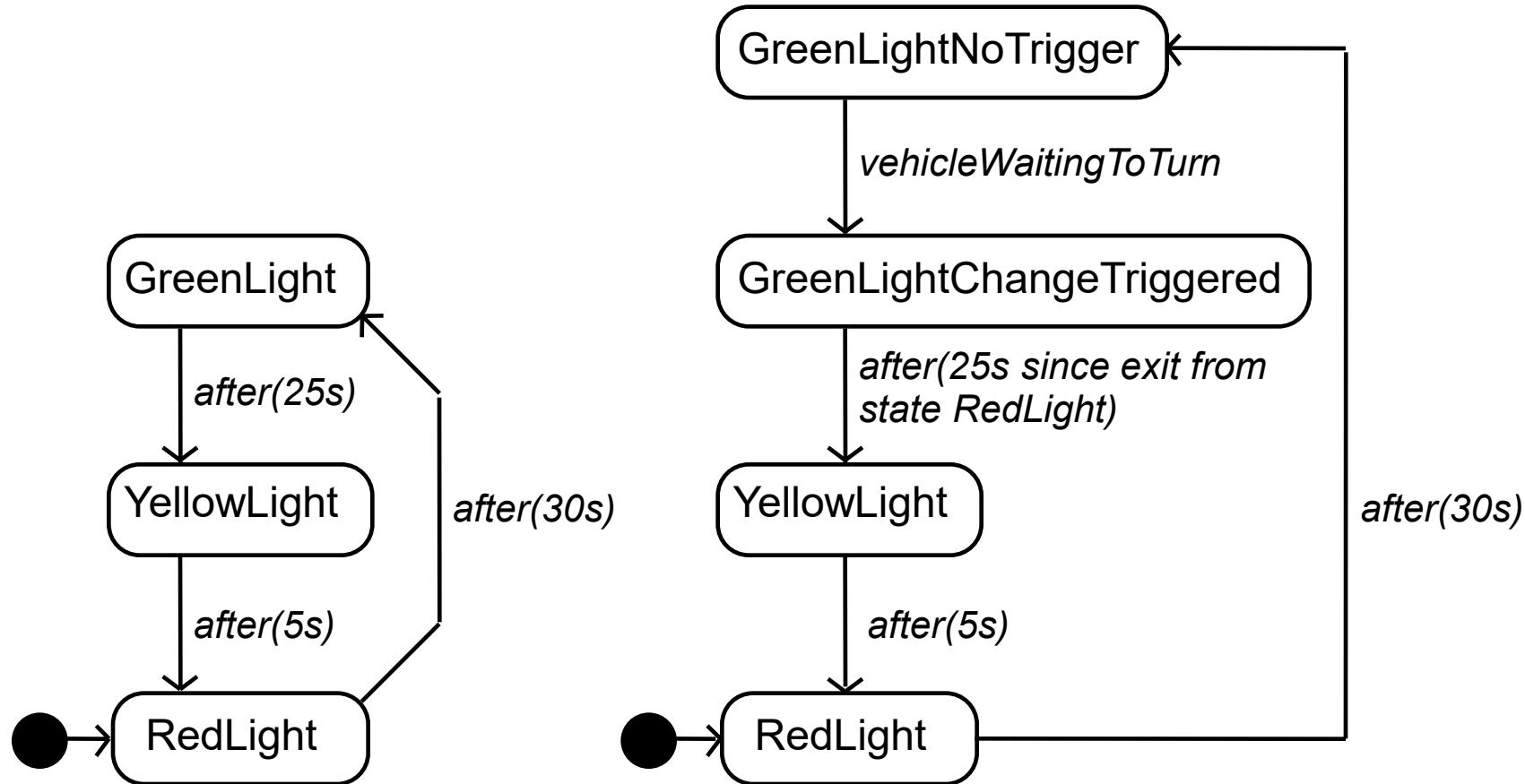
# State Diagrams

- A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.
  - At any given point in time, the system or object is in a certain state.
    - Being in a state means that it will behave in a *specific way* in response to any events that occur.
  - Some events will cause the system to change state.
    - In the new state, the system will behave in a different way to events.
  - A state diagram is a directed graph where the nodes are states and the arcs are transitions.

# State diagrams – an example



# State diagrams – an example of transitions with time-outs and conditions



# Abstract classes

- **abstract class:** a hybrid between an interface and a class
  - used to define a generic parent type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use new to create any objects of their type)

*What goes in an abstract class?*

- implement common state and behavior that will be inherited by subclasses (parent class role)
- declare generic behaviors that subclasses must implement (interface role)

# Abstract class syntax

- put abstract keyword on class header and on any generic (abstract) methods
  - A class can be declared abstract even though it has no abstract methods
  - Any class with abstract methods *must* be declared abstract, or it will not compile
- Variables of abstract types may be declared, but *objects* of abstract types cannot be constructed

# Abstract class example

```
public abstract class Shape implements
    Shape2D {
    private int myX, myY;

    public Shape(int x, int y) {
        myX = x; myY = y;
    }
    public int getX() { return myX; }
    public int getY() { return myY; }

    public abstract double getArea();
    public abstract double getPerimeter();
}
```

- Shape class cannot be instantiated
- all classes that extend Shape **must** implement getArea and getPerimeter or else must also be declared abstract

# Extending an abstract class

```
public class Rectangle extends Shape {  
    private int myWidth, myHeight;  
  
    public Rectangle(int x, int y, int w, int  
h) {  
        super(x, y);  
        myWidth = w;    myHeight = h;  
    }  
  
public double getArea() {  
    return myWidth * myHeight;  
}  
public double getPerimeter() {  
    return 2*myWidth + 2*myHeight;  
}  
}  
  
// ... example usage ...  
Shape rect = new Rectangle(1, 2, 10, 5);
```

# Interface / abstract class chart

TABLE 3.1

Comparison of Actual Classes, Abstract Classes, and Interfaces

| Property                                       | Actual Class | Abstract Class | Interface  |
|------------------------------------------------|--------------|----------------|------------|
| Instances (objects) of this can be created     | Yes          | No             | No         |
| This can define instance variables and methods | Yes          | Yes            | No         |
| This can define constants                      | Yes          | Yes            | Yes        |
| The number of these a class can extend         | 0 or 1       | 0 or 1         | 0          |
| The number of these a class can implement      | 0            | 0              | Any number |
| This can extend another class                  | Yes          | Yes            | No         |
| This can declare abstract methods              | No           | Yes            | Yes        |
| Variables of this type can be declared         | Yes          | Yes            | Yes        |

# Java Interfaces

# Interface

- is a way to describe what classes should do,
- without specifying how they should do it.
- NOT a class but a set of requirements
- class - implements an interface

# Concept

```
public interface Comparable  
{  
    int compareTo(Object otherObject);  
}
```

- Any class implementing the Comparable interface contains a compareTo method,
- and this method must take an Object parameter and return an integer

# Interface declarations

- The declaration consists of a keyword interface, its name, and the members
- Similar to classes, interfaces can have three types of members
  - constants (fields)
  - methods
  - nested classes and interfaces

# Interface member – constants

- An interface can define named constants, which are public, static and final (**these modifiers are omitted by convention**) automatically. Interfaces never contain instance fields.
- All the named constants MUST be initialized

An example interface

```
interface Verbose {  
    int SILENT = 0;  
    int TERSE = 1;  
    int NORMAL = 2;  
    int VERBOSE = 3;  
  
    void setVerbosity (int level);  
    int getVerbosity();  
}
```

# Interface member – methods

- They are implicitly abstract (omitted by convention). So every method declaration consists of the method header and a semicolon.
- They are implicitly public (omitted by convention). No other types of access modifiers are allowed.
- They can't be final, nor static

# Modifiers of interfaces itself

- An interface can have different modifiers as follows
  - public/package (default)
  - abstract
    - all interfaces are implicitly abstract
    - omitted by convention

# To implement interfaces in a class

- Two steps to make a class implement an interface
  - declare that the class intends to implement the given interface by using the `implements` keyword

```
class Employee implements Comparable  
{ . . . }
```

- supply definitions for **all** methods in the interface

```
public int compareTo(Object otherObject) {  
  
    Employee other = (Employee) otherObject;  
    if (salary < other.salary) return -1;  
    if (salary > other.salary) return 1;  
    return 0; }
```

**note:** in the Comparable interface declaration, the method `compareTo()` is `public` implicitly but this modifier is omitted. But in the `Employee` class design, you cannot omit the `public` modifier, otherwise, it will be assumed to have package accessibility

# To implement interfaces in a class

- If a class leaves any method of the interface undefined, the class becomes abstract class and must be declared abstract
- A single class can implement multiple interfaces. Just separate the interface names by comma

```
class Employee implements Comparable,  
Cloneable { . . . }
```

# Instantiation properties of interfaces

- Interfaces are not classes. You can never use the `new` operator to instantiate an interface.

```
public interface Comparable {  
    . . . }  
Comparable x = new Comparable( );
```

- You can still declare interface variables

```
Comparable x;
```

but they must refer to an object of a class that implements the interface

```
class Employee implements Comparable {  
    . . .  
}  
x = new Employee( );
```

# Extending interfaces

- Interfaces support **multiple** inheritance – an interface can extend more than one interface
- Superinterfaces and subinterfaces

## Example

```
public interface SerializableRunnable extends  
java.io.Serializable, Runnable {  
    . . .  
}
```

# Extending interfaces – about constants (1)

- An extended interface inherits all the constants from its superinterfaces
- Take care when the subinterface inherits more than one constants with the same name, or the subinterface and superinterface contain constants with the same name — always use sufficient enough information to refer to the target constants

- When an interface inherits two or more constants with the same name
  - In the subinterface, explicitly use the superinterface name to refer to the constant of that superinterface

```
interface A {  
    int val = 1;  
}  
  
interface B {  
    int val = 2;  
}  
  
interface C extends A, B {  
    System.out.println("A.val = "+ A.val);  
    System.out.println("B.val = "+ B.val);  
}
```

- If a superinterface and a subinterface contain two constants with the same name, then the one belonging to the superinterface is **hidden**
  1. in the subinterface
    - access the subinterface-version constants by directly using its name
    - access the superinterface-version constants by using the superinterface name followed by a dot and then the constant name

```
interface X {
    int val = 1; }

interface Y extends X{
    int val = 2;
    int sum = val + X.val; }
```

The diagram shows a box divided into two horizontal sections. The top section is labeled "Y's val" and the bottom section is labeled "X's val". Two arrows originate from the code: one arrow points from the declaration of "val" in interface Y to the "Y's val" section, and another arrow points from the declaration of "sum" in interface Y to the "X's val" section.

2. outside the subinterface and the superinterface
  - you can access both of the constants by explicitly giving the interface name.  
E.g. in previous example, use `Y.val` and `Y.sum` to access constants `val` and `sum` of interface `Y`, and use `X.val` to access constant `val` of interface `X`.

- When a superinterface and a subinterface contain two constants with the same name, and a class implements the subinterface
  - the class inherits the subinterface-version constants as its static fields. Their access follow the rule of class's static fields access.

```
E.g    class Z implements Y { }

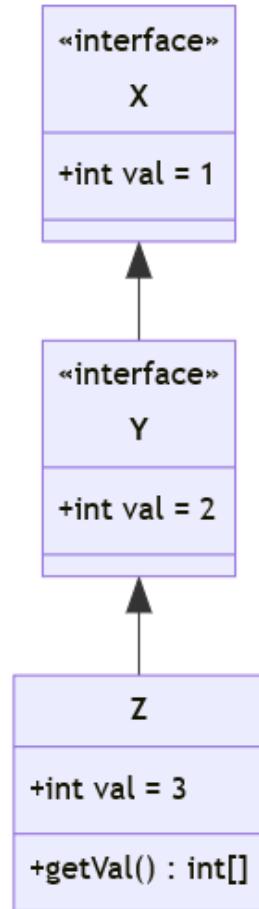
        //inside the class
System.out.println("Z.val:"+val);    //Z.val = 2
        //outside the class
System.out.println("Z.val:"+Z.val); //Z.val = 2
```

- object reference can be used to access the constants
  - subinterface-version constants are accessed by using the object reference followed by a dot followed by the constant name
  - superinterface-version constants are accessed by explicit casting

E.g.

```
Z v = new Z();
System.out.print( "v.val = " + v.val
                  +
                  ", ((Y)v).val = " + ((Y)v).val
                  +
                  ", ((X)v).val = " +
                  ((X)v).val );
```

**output:** v.val = 2, ((Y)v).val = 2, ((X)v).val = 1



Code the Interfaces X and Y. Code Class Z which has a method getVal which returns all three values of val.

# Extending interfaces – about methods

- If a declared method in a subinterface has the same signature as an inherited method and the same return type, then the new declaration *overrides* the inherited method in its superinterface. If the only difference is in the return type, then there will be a compile-time error
- An interface can inherit more than one methods with the same signature and return type. A class can implement different interfaces containing methods with the same signature and return type.
- Overriding in interfaces has **NO** question of ambiguity. The real behavior is ultimately decided by the implementation in the class implementing them. The real issue is whether a single implementation can honor all the contracts implied by that method in different interfaces
- Methods with same name but different parameter lists are overloaded

# Interface References

## Referencing an Interface

### Variable

*intfRef.varName*

*intfRef.mthName(args)*

```
interface A {  
    void display(String s);}
```

```
class C1 implements A {  
    public void display(String s) {  
        System.out.println("C1: " + s);  
    }}
```

```
class C2 implements A {  
    public void display(String s) {  
        System.out.println("C2: " + s);  
    }}
```

```
class C3 implements A {  
    public void display(String s) {  
        System.out.println("C3: " + s);  
    }}
```

```
class InterfaceReferenceVariable {  
    public static void main(String args[]) {  
        A a;  
        a = new C1();  
        a.display("String 1");  
        a = new C2();  
        a.display("String 2");  
        a = new C3();  
        a.display("String 3");  
    }  
}
```

Result :

C1: String 1

C2: String 2

C3: String 3

## Marker interfaces

- A marker (tagging) interface has neither methods nor constants, its only purpose is to allow the use of `instanceof` in a type inquiry.

# Interfaces and abstract classes

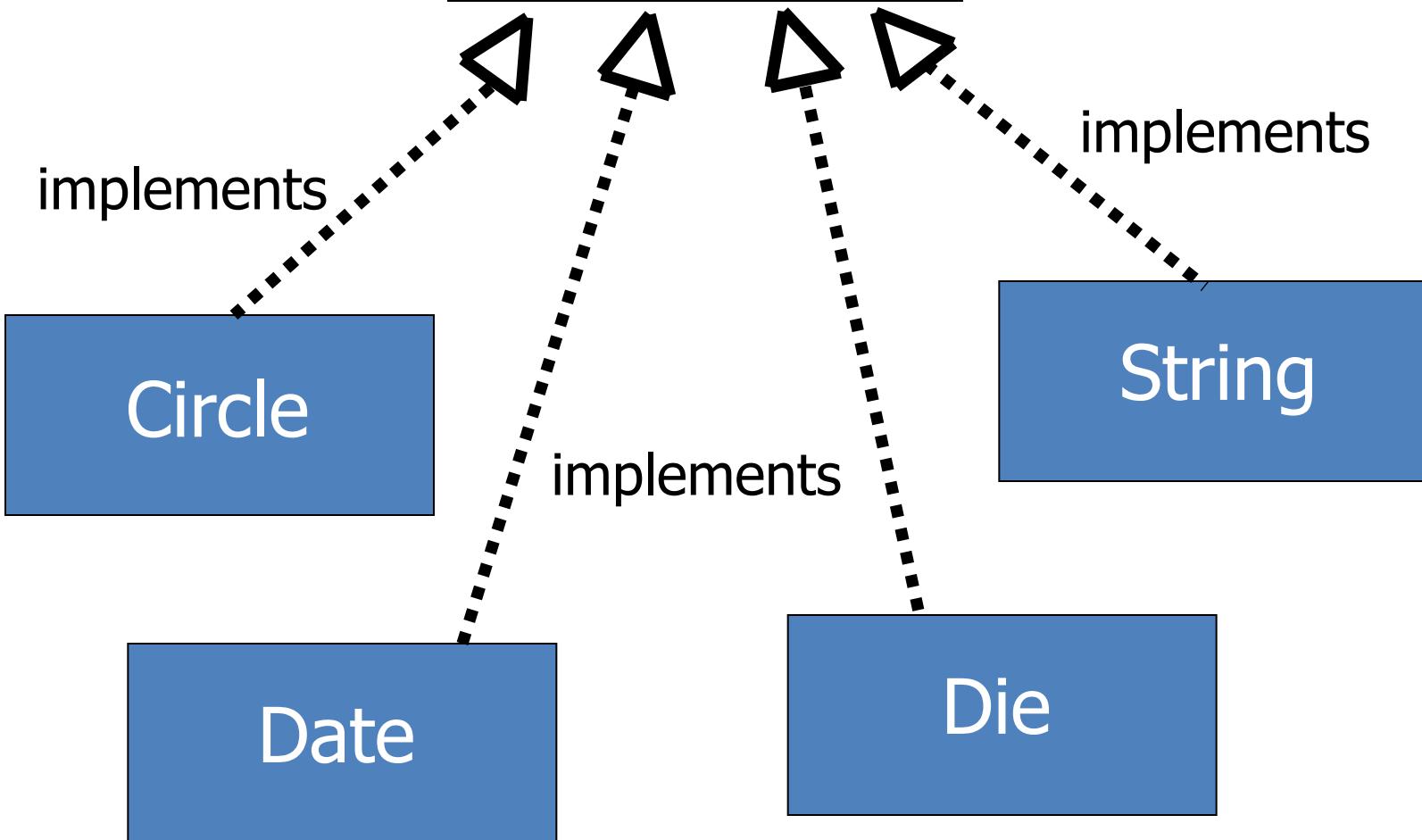
- Why bother introducing two concepts: abstract class and interface?

```
abstract class Comparable {
    public abstract int compareTo (Object otherObject);
}
class Employee extends Comparable {
    public int compareTo (Object otherObject) { . . . }
}

public interface Comparable {
    int compareTo (Object otherObject);
}
class Employee implements Comparable {
    public int compareTo (Object otherObject) { . . . }
}
```

- A class can only extend a single abstract class, but it can implement as many interfaces as it wants
- An abstract class can have a partial implementation, protected parts, static methods and so on, while interfaces are limited to public constants and public methods with no implementation

<<interface>>  
Comparable



# Comparable interface

```
public interface Comparable {  
    public int compareTo( Object o );  
}
```

for x, y objects of the same class

x.compareTo(y) < 0 means “x < y”

x.compareTo(y) > 0 means “x > y”

otherwise, “x is neither < nor > y”

- recommended that compareTo() is consistent with equals()
- if o cannot be cast to the same class as x, then generates a ClassCastException

```
public class Student implements Comparable
{ private String name;
  public int compareTo( Object o )
  {
    Student other = (Student) o;
    return ((this.name).compareTo(other.name));
  }
}
```

**use < & > to compare primitives**

**invoke compareTo() method to compare objects**

**remember to have “implements Comparable” on  
the class header**

# Mixing inheritance, interfaces

- It is legal for a class to extend a parent and to implement any number of interfaces

```
public class BankAccount {  
    // ...  
}
```

```
public class NumberedAccount  
    extends BankAccount implements Comparable {  
    private int myNumber;  
  
    public int compareTo(Object o) {  
        return myNumber -  
            ((BankAccount)o).myNumber;  
    }  
}
```

# A problem with interfaces

```
public interface Shape2D {  
    int getX();  
    int getY();  
    double getArea();  
    double getPerimeter();  
}
```

- Every shape will implement `getX` and `getY` the same, but each shape probably implements `getArea` and `getPerimeter` differently

# A bad solution

```
public class Shape implements Shape2D {  
    private int myX, myY;  
  
    public Shape(int x, int y) {  
        myX = x; myY = y;  
    }  
    public int getX() { return myX; }  
    public int getY() { return myY; }  
  
    // subclasses should override these,  
    // please  
    public double getArea() { return 0; }  
    public double getPerimeter() { return 0; }  
}
```

- BAD: the Shape class can be instantiated
- BAD: a subclass might forget to override the methods

# Understanding Packages

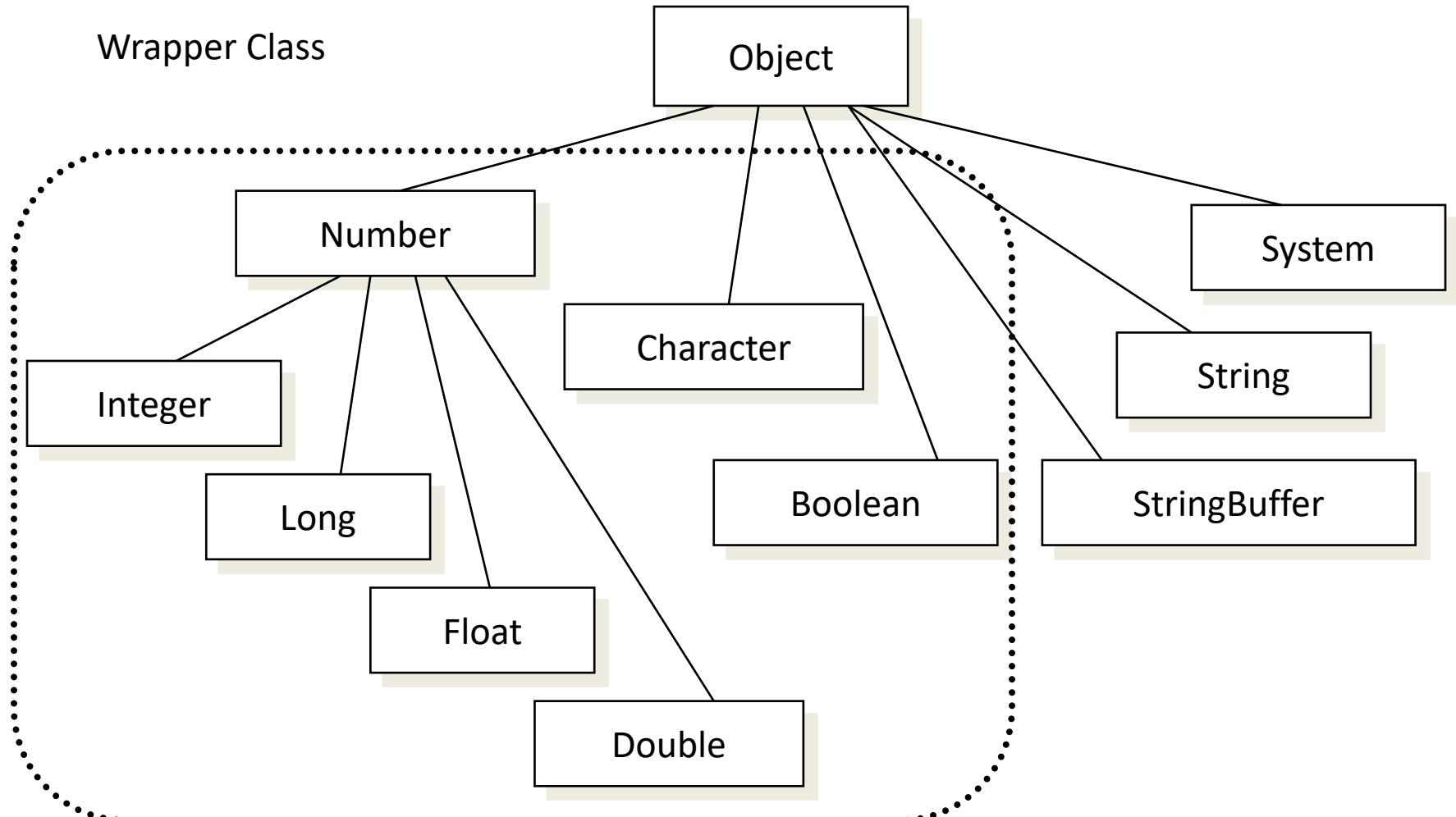
# Packages

- A Package is a unique named collection of classes
- The purpose of grouping classes is to make it easy to add any or all of the classes
- Names of the classes within the package should be unique, but, same name can be repeated across the packages – as a class name is qualified with the package name

# Package in Java

- A package is a namespace that organizes a set of related classes and interfaces.
- A java package is a group of related classes similar to a class library.
- Keyword ‘import’
- All of the standard classes are contained with in the packages.
- If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.
- *java.lang* is special

# Package in Java



# Selected packages

- `java.lang` -- `String`, wrapper classes, `Math`
- `java.util` – `Calendar`, `Date`, `Vector`
- `java.applet` – `Applet`
- `java.text` – `DateFormat`
- `java.awt` – `Graphics`, `Button`, `Label..`
- `javax.swing` – `JButton`
- `java.io` – `InputStream`, `OutputStream`

# Naming Conventions

- Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.  
Companies use their reversed Internet domain name to begin their package name
  - For example, com.example.orion for a package named orion created by a programmer at example.com.
- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name
  - (for example, com.company.region.package).
- Packages in the Java language itself begin with java. or javax.

# Naming conventions

- In some cases, the internet domain name may not be a valid package name.
  - This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int".
  - In this event, the suggested convention is to add an underscore.
  - Ex: `clipart-open.org` → `org.clipart_open`

# Using Package Members

- The types that comprise a package are known as the *package members*. To use a public package member from outside its package, you must do one of the following:
  - Refer to the member by its fully qualified name
    - *graphics.Rectangle myRect = new graphics.Rectangle();*
  - Import the package member
    - *import graphics.Rectangle;*  
*Rectangle myRectangle = new Rectangle();*
  - Import the member's entire package
    - *import java.util.\*;*
    - *import graphics.\*;*
    - *import graphics.A\*; //does not work*

# Packaging your classes

- Add a package statement as the first statement in the source file containing the class definition
- It must always be the first statement
- Ex: **package geometry;**

**keyword**

**necessary**

```
//Sphere.java
package geometry;
public class Sphere
{
    //details of the class definition
}
```

# Packaging your classes

- Directory : geometry
- File: *Sphere.java*
- *Compiling: Ensure that you are out of the directory*  
`javac /geometry/Sphere.java`

*To execute*

`java geometry.Sphere`

*For a large number of files in a package you can use*  
`javac /geometry/*.java`

All interdependencies will be figured out by the compiler

# Packages and directory structure

- A package is intimately related to the directory structure in which it is stored
- A class with a *Classname* must be stored in a file called *Classname.java*
- Similarly, all the files for classes within a package, *packageName* must be included in a directory with the name, *packageName*.
- A package need not have a single name. it can be a sequence of names separated by a period

*package geometry.shapes3D;*

*package geometry.shapes2D;*

- shapes3D and shapes2D are sub directories of geometry directory.

# Setting classpath

- From the command prompt,

```
set CLASSPATH = .;c:\MySource; c:\MyPackages
```

- Unless a class path is set, or set incorrectly, java will not be able to find the classes in any new packages you might create.

# Adding classes from a Package to your Program

- Classes with ‘public’ are accessible to your program by using ‘*import*’ statement.
  - *import geometry.shapes3D.\*;* //includes all the classes from this package
  - ‘\*’ selects all the classes in the package
  - You can refer any public class in the package
  - If you have to add a specific class, specify explicitly like,

*import geometry.shapes3D.Sphere;*

- ‘\*’ can be only used to select all the classes in a package. You cannot use *geometry.\** to select all the packages in the directory *geometry*.

# Encapsulation of classes into a package

- Add a class into a package — two steps:
  1. put the name of the package at the top of your source file

```
package com.hostname.corejava;  
public class Employee {  
    . . .  
}
```

2. put the files in a package into a subdirectory which matches the full package name

→ stored in the file “Employee.java” which is stored under “somePath/com(hostname)/corejava/”

# Setting the class path

E.g.

- current classpath:

/home/user/classdir.:./home/user/archives/archive.jar

- to search for the class file of the  
com.horstmann.corejava.Employee class

first searches in the system class files that are stored in  
archives in the jre/lib and jre/lib/ext directories.

If it can't find the class there it will search in the order:

- 1) /home/user/classdir/com/horstmann/corejava/Employee.class
- 2) ./com/horstmann/corejava/Employee.class
- 3) com/horstmann/corejava/Employee.class inside  
/home/user/archives/archive.jar

- if found the interpreter stops searching process

# Naming conventions

- **Package names:** start with lowercase letter
  - E.g. java.util, java.net, java.io ...
- **Class names:** start with uppercase letter
  - E.g. File, Math ...
  - avoid name conflicts with packages
  - avoid name conflicts with standard keywords in java system
- **Variable, field and method names:** start with lowercase letter
  - E.g. x, out, abs ...
- **Constant names:** all uppercase letters
  - E.g. PI ...
- **Multi-word names:** capitalize the first letter of each word after the first one
  - E.g. HelloWorldApp, getName ...
- **Exception class names:** (1) start with uppercase letter (2) end with “Exception” with normal exception and “Error” with fatal exception
  - E.g. OutOfMemoryError, FileNotFoundException

# Nested Classes

# Nested Classes and Interfaces

- ❖ Classes and interfaces can be declared inside other classes and interfaces, either as members or within blocks of code.

# Some Definitions

- ¶ **Nested** = a class or interface definition is somewhere inside another one
- ¶ **Top-level class or interface** = an instance of a type **does not** need to be instantiated with a reference to any enclosing instance
- ¶ **Inner class** = an instance of a class **does** need to be instantiated with an enclosing instance
- ¶ **Inner Member class** = defined inside another class, but not inside any methods.
- ¶ **Inner Local class** = defined inside a method
- ¶ **Named Inner Local class** = has a class name
- ¶ **Anonymous Inner Local class** = does not have a class name

# Nested Classes

```
class TheEnclosingClass{  
    ...  
    class ANestedClass {  
        ...  
    }  
}
```

- ❖ **Definition:** A **nested class** is a class that is a member of another class.
- ❖ **Reason for making nested classes:**
  - ❖ the nested class **makes sense only in the context of its enclosing class**
  - ❖ the nested class **needs the enclosing class to have the right functionality.**

# Nested Static Classes

- ¶ A nested class that is declared static **is attached to the enclosing class** and not to objects of the enclosing class. Instance fields and methods can not be directly accessed.

# Example: *Linked List*

```
public class LinkedList {  
    private Node first;  
    ....  
    public static class Node{  
        public Node next;  
        public Object data;  
    }  
    ....  
}
```

# Static Nested Classes/Interfaces — Overview

- ❖ A nested class/interface which is declared as static acts just like any non-nested class/interface, except that its name and accessibility are defined by its enclosing type.
- ❖ Static nested types are members of their enclosing type
  - ❖ They can access all other members of the enclosing type including the private ones.
  - ❖ Inside a class, the static nested classes/interfaces can have private, package, protected or public access; while inside an interface, all the static nested classes/interfaces are implicitly public.
  - ❖ They serve as a structuring and scoping mechanism for logically related types

# Static Inner Classes

- ❖ Since a static inner class has no connection to an object of the outer class, within an inner class method
  - ❖ Instance variables of the outer class cannot be referenced
  - ❖ Nonstatic methods of the outer class cannot be invoked
- ❖ To invoke a static method or to name a static variable of a static inner class within the outer class, preface each with the name of the inner class and a dot

# Static Nested Classes/Interfaces (cont.)

- ❖ Static nested classes
  - ❖ If a class is nested in an interface, it's always static (omitted by convention)
  - ❖ It can extend any other class, implement any interface and itself be extended by any other class to which it's accessible
  - ❖ Static nested classes serve as a mechanism for defining logically related types within a context where that type makes sense.

# Static Nested Classes/Interfaces (cont.)

## ❖ Nested interfaces

- ❖ Nested interfaces are always static (omitted by convention) since they don't provide implementation

# Static Nested Classes/Interfaces (cont.)

```
public class BankAccount {  
    private long number;      //account number  
    private long balance;    //current balance  
  
    public static class Permissions {  
        public boolean canDeposit, canWithdraw,  
canClose;  
    }  
    // . . .  
}
```

- ❖ Code outside the `BankAccount` class must use `BankAccount.Permissions` to refer to this class

```
BankAccount.Permissions perm = new  
BankAccount.Permissions();
```

## Home Work: Modify the LinkedList and Node classes on Slide 6

1. Give code for a CreateLL class which will have main() to take input from the user- No of LL to be created, number of nodes for each
2. LinkedList should have a int variable numNodes which will keep a count of ALL nodes created

# Inner Classes

- ❖ A nested class that is not static is called an inner class.
- ❖ An inner class is associated with an object of its enclosing class and it has direct and unlimited access to that object's instance variables and methods.
- ❖ A nested class can be declared at the top level inside a class or inside any block of code.

# Inner classes

```
class Outer {  
    int n;  
  
    class Inner {  
        int ten = 10;  
        void setNToTen( ) { n = ten; }  
    }  
  
    void setN ( ) {  
        new Inner( ).setNToTen( );  
    }  
}
```

# Inner Class

## ❖ Name Reference

- ❖ inside OuterClass : Use InnerClass Simple name
- ❖ outside OuterClass : OuterClass.InnerClass

```
public static void main(String[] args) {  
    OuterClass outObj = new OuterClass();  
    OuterClass.InnerClass inObj = outObj.new InnerClass();  
}
```

## ❖ Access Modifier

- ❖ public, private, protected

**Inner class cannot have static variables**

# Nesting Inner Classes

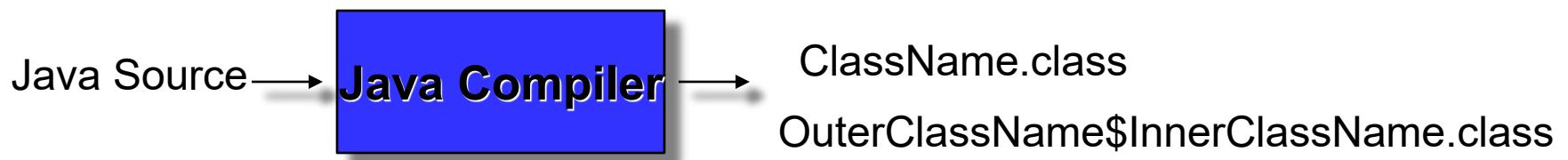
- ❖ It is legal to nest inner classes within inner classes
  - ❖ The rules are the same as before, but the names get longer
  - ❖ Given class **A**, which has public inner class **B**, which has public inner class **C**, then the following is valid:

```
A aObject = new A();
```

```
A.B bObject = aObject.new B();
```

```
A.B.C cObject = bObject.new C();
```

# Name of Inner Class



```
class Outer {  
    class Inner1 {  
        class Inner2 { // ... }  
        // ...  
    }  
    // ...  
}
```

→ **Outer.class**  
**Outer\$Inner1.class**  
**Outer\$Inner1\$Inner2.class**

# Simple inner class example

```
class Outer{
    private int x1;

    Outer(int x1){
        this.x1 = x1;
    }

    public void foo(){ System.out.println("fooing");}

    public class Inner{
        private int x1 = 0;
        void foo(){
            System.out.println("Outer value of x1: " + Outer.this.x1);
            System.out.println("Inner value of x1: " + this.x1);
        }
    }
}
```

# Simple example, cont -- driver

- Rules for instantiation

```
public class TestDrive{  
  
    public static void main(String[] args){  
        Outer outer = new Outer();  
        Inner inner = outer.new Inner(); //must call new through  
                                         //outer object handle  
        inner.foo();  
  
        // note that this can only be done if inner is visible  
        // according to the regular scoping rules  
    }  
}
```

# Non-static Classes — Inner classes

- Inner classes are associated with instances of its enclosing class.

```
public class BankAccount {  
    private long number;      // account number  
    private long balance;    // current balance  
    private Action lastAct; //last action performed  
  
    public class Action {  
        private String act;  
        private long amount;  
  
        Action(String act, long amount) {  
            this.act = act;  
            this.amount = amount;  
        }  
    }  
}
```

# Non-static Classes — Inner classes

```
public void deposit(long amount) {  
    balance += amount;  
    lastAct = new Action("deposit", amount);  
}  
  
public void withdraw(long amount) {  
    balance -= amount;  
    lastAct = new Action("withdraw", amount);  
}  
// . . .  
}
```

# Inner classes (cont.)

- When an inner class object is created, it MUST be associated with an object of its enclosing class.
- Usually, inner class objects are created inside instance methods of the enclosing class. When this occurs, the current enclosing object `this` is associated with the inner object by default.

```
lastAct = this.new Action("deposit", amount);
```

- When `deposit` creates an `Action` object, a reference to the enclosing `BankAccount` object is automatically stored in the `Action` object.

# Inner classes (cont.)

& method for transfer is added

```
public void transfer(BankAccount other, long  
amount) {  
  
    other.withdraw(amount);  
    deposit(amount);  
  
    lastAct = this.new Action("transfer", amount);  
  
  
  
    other.lastAct = other.new Action("transfer",  
        amount); // will this work?  
}
```

# Inner classes (cont.)

- ❖ The enclosing class can also access the private members of its inner class, but only via explicit reference to an inner class object.
- ❖ An object of the enclosing class need not have any inner class objects associated with it, or it could have many.
- ❖ An inner class acts as a top-level class except that it can't have static members (except for final static fields).
- ❖ Inner classes can also be extended.

# Inheritance, Scoping and Hiding

- ❖ All members declared within the enclosing class are said to be in scope inside the inner class.
- ❖ An inner class's own fields and methods can hide those of the enclosing object. Two possible ways:
  - 1). A member with same name is declared in the inner class
    - ❖ Any direct use of the name refers to the version inside the inner class

```
class Host {  
    int x;  
    class Helper {  
        void increment()  
        {int x=0; x++;}  
    } } X
```

- ❖ Access to the enclosing object's members needs be preceded by this explicitly

# Inheritance, Scoping and Hiding

2). A member with the same name is inherited by the inner class

- ¶ The direct use of the name is not allowed

```
class Host {  
    int x;  
    class Helper extends Unknown {  
        //Unknown class has a field x  
        void increment () {x++; }  
    }  
}
```

- ¶ Use `enclosingClassName.this.name` to refer to the version in the outer class
- ¶ or `super.name` to refer to the version in the super class
- ¶ Use `this.name` in the inner class

# Specifying which scope you want:

```
public class ScopeConflict {  
    String s = "outer";  
  
    class Inner extends SuperClass {  
  
        String s = "inner";  
  
        void foo(){  
            System.out.println(this.s);  
            System.out.println(super.s);  
            System.out.println(ScopeConflict.this.s);  
        }  
    }  
}  
  
class SuperClass {  
    String s = "super";  
}
```

output:

**inner** (from this.s)

**super** (from super.s)

**outer** (from ScopeConflict.this.s)

# Inheritance, Scoping and Hiding (cont.)

- ¶ A method within an inner class which has the same name as an enclosing **method hides all overloaded forms** of the enclosing method, even if the inner class itself does not declare those overloaded forms.

```
class Outer {  
    void print() {}  
    void print(int value) {}  
    class Inner {  
        void print() {}  
        void show() {  
            print();  
            Outer.this.print();  
            print(1); // no Inner.print(int)  
        }  
    }  
}
```

# Local Inner Classes

- ❖ You can define inner classes in code blocks. They are called *local inner classes*.
- ❖ You can actually declare an inner class inside of a method, just like you could declare a local variable.
- ❖ Local classes do not get an access modifier – they are automatically restricted to the method they are defined in
- ❖ Can only refer to final members of the enclosing class
  - ❖ They are NOT members of the class which contains the code but are local to that block, as a local variable.
  - ❖ They are completely inaccessible outside of the block.
  - ❖ Only one modifier is allowed—final—which makes them unextendable

# Anonymous Inner classes

- When using a local inner class, if you only want to make one instance of it, you don't even need to give it a name
- This is known as an anonymous inner class
- These are convenient for event programming
- However, the syntax is extremely cryptic.

# Anonymous Inner classes

- ¶ You have to look very carefully to see a difference between construction of a new object, and construction of a new inner class extending a class.

```
//A person object from Person class
```

```
Person queen=new Person("Mary"); //Person Object
```

---

```
//An object of an inner class extending Person interface
```

```
Person count = new Person() {
```

```
    //class code here
```

```
};
```

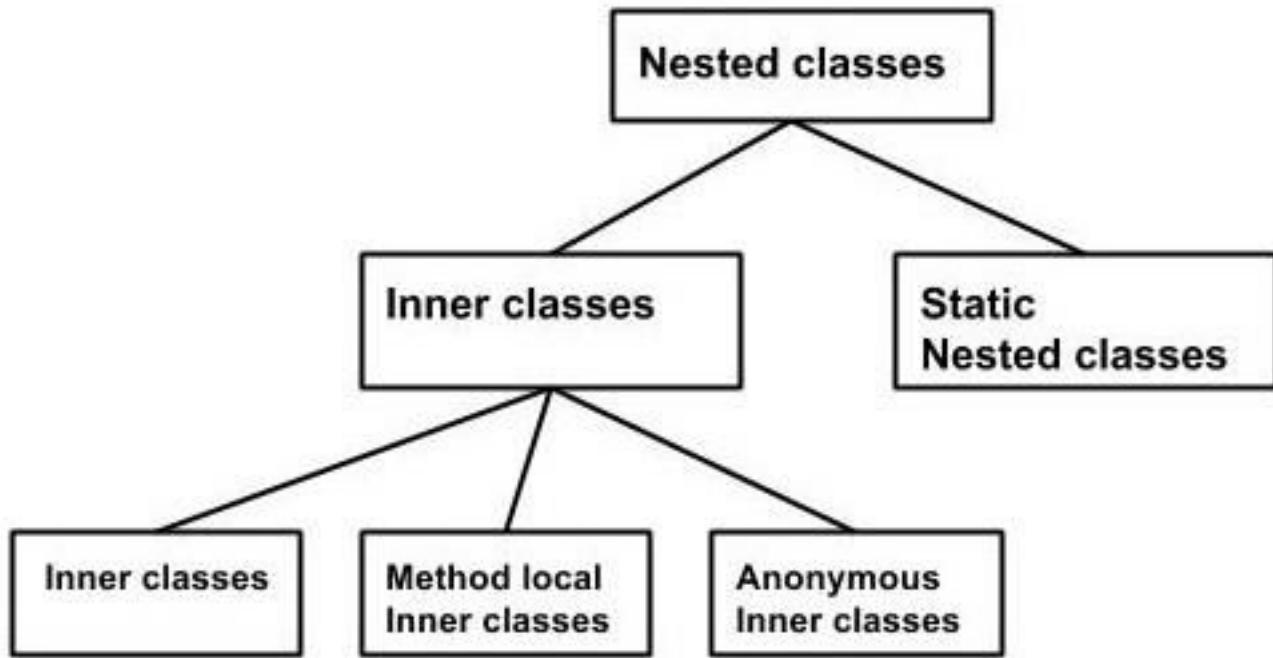
# Anonymous Inner classes

- ❖ Anonymous Inner classes cannot have constructors, since constructors have to have the same name as the class, and these classes have no names.
- ❖ As you can see, the syntax for these is confusing – both for people writing and reading the code. Use this with care, if at all.

# Anonymous Inner classes

```
public void start(final double rate)
{
    ActionListener adder = new ActionListener()
    {
        public void actionPerformed(ActionEvent evt)
        {
            double interest = balance * rate / 100;
            balance += interest;
        }
    };
}
```

- ❖ This is saying, construct a new object of a class that implements the ActionListener interface, where the one required method (actionPerformed) is defined inside the brackets.



# Abstract Data Types

# Abstract Data Types

- An **abstract data type** is a mathematical set of data, along with operations defined on that kind of data
- Examples:
  - ◆ **int**: it is the set of integers (up to a certain magnitude), with operations +, -, /, \*, %
  - ◆ **double**: it's the set of decimal numbers (up to a certain magnitude), with operations +, -, /, \*

# Data Structures

- A data structure is a user-defined abstract data type
- Examples:
  - ◆ **Complex numbers:** with operations  $+$ ,  $-$ ,  $/$ ,  $*$ , *magnitude*, *angle*, etc.
  - ◆ **Stack:** with operations *push*, *pop*, *peek*, *isempty*
  - ◆ **Queue:** *enqueue*, *dequeue*, *isempty* ...
  - ◆ **Binary Search Tree:** *insert*, *delete*, *search*.
  - ◆ **Heap:** *insert*, *min*, *delete-min*.

# Data Structure Design

- Specification
  - ◆ A set of data
  - ◆ Specifications for a number of operations to be performed on the data
- Design
  - ◆ A lay-out organization of the data
  - ◆ Algorithms for the operations
- Goals of Design: **fast** operations

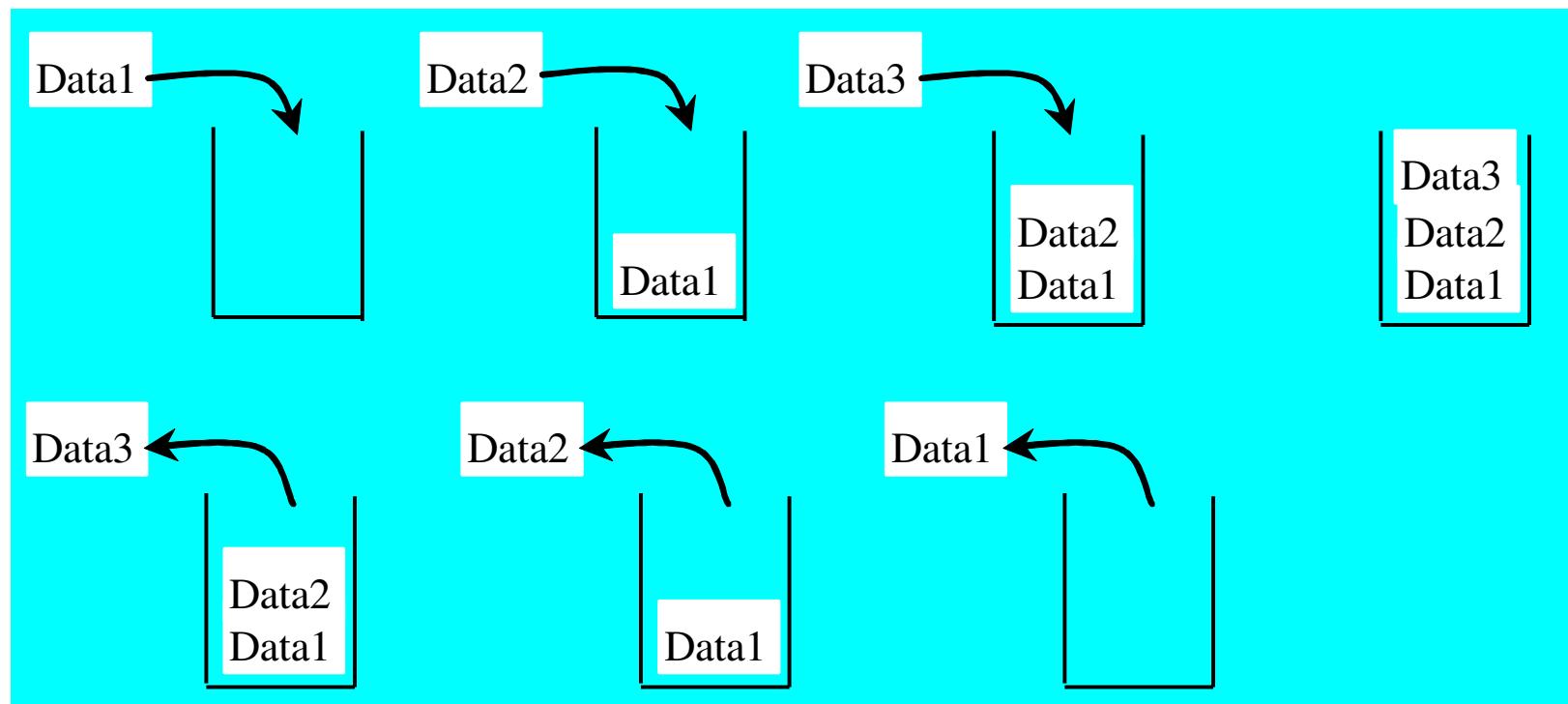
# Implementation of a Data Structure

- Representation of the data using built-in data types of the programming language (such as int, double, char, strings, arrays, structs, classes, pointers, etc.)
- Language implementation (code) of the algorithms for the operations
- In OOP languages both the data representation and the operations are aggregated together into what is called **objects**
- The data type of such objects are called **classes**.
- Classes are blue prints, objects are instances.

# Stack, Queue and List

# Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.

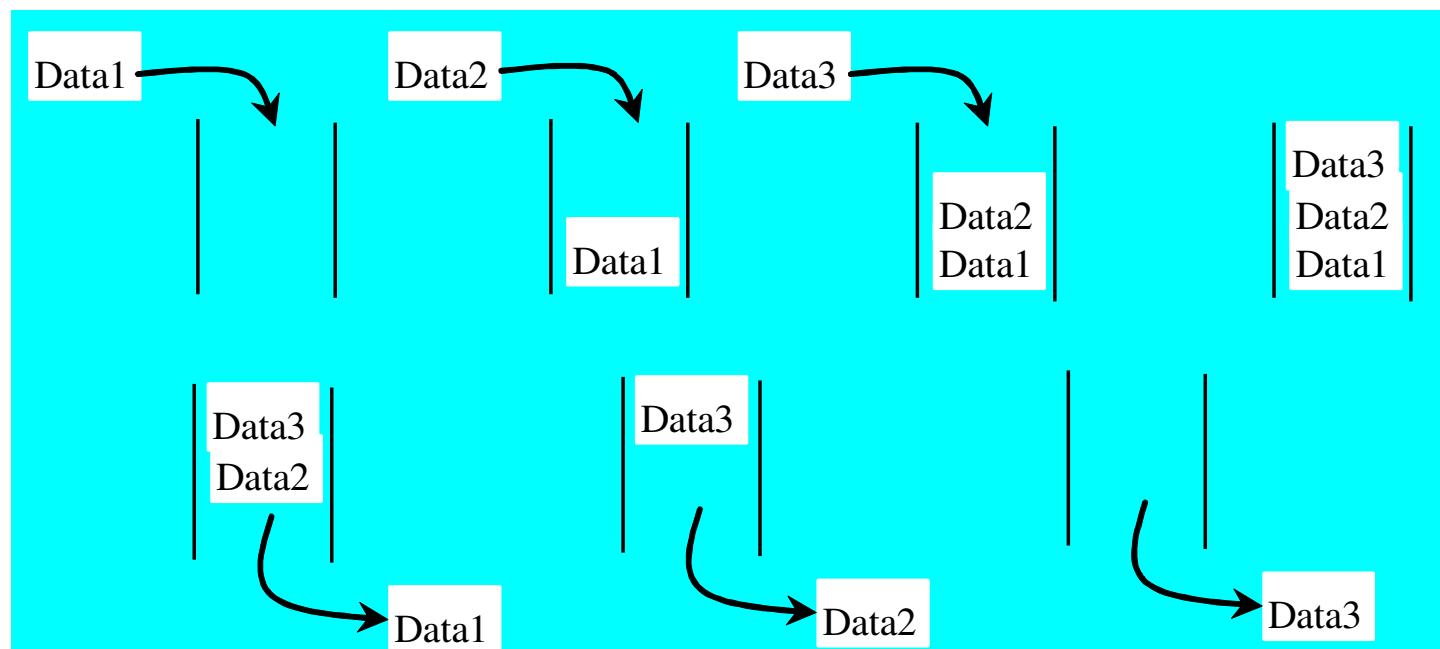


# Stack

- A stack is maintained Last-In-First-Out  
(not unlike a stack of plates in a cafeteria)
- Standard operations
  - ◆ **isEmpty ()** : returns **true** or **false**
  - ◆ **top ()** : returns copy of value at top of stack (without removing it)
  - ◆ **push (v)** : adds a value v at the top of the stack
  - ◆ **pop ()** : removes and returns value at top

# Queues

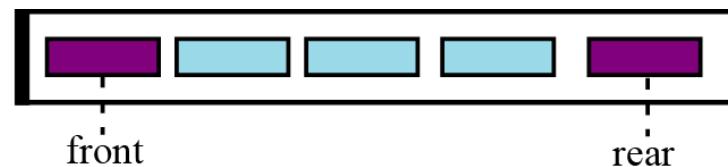
A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.



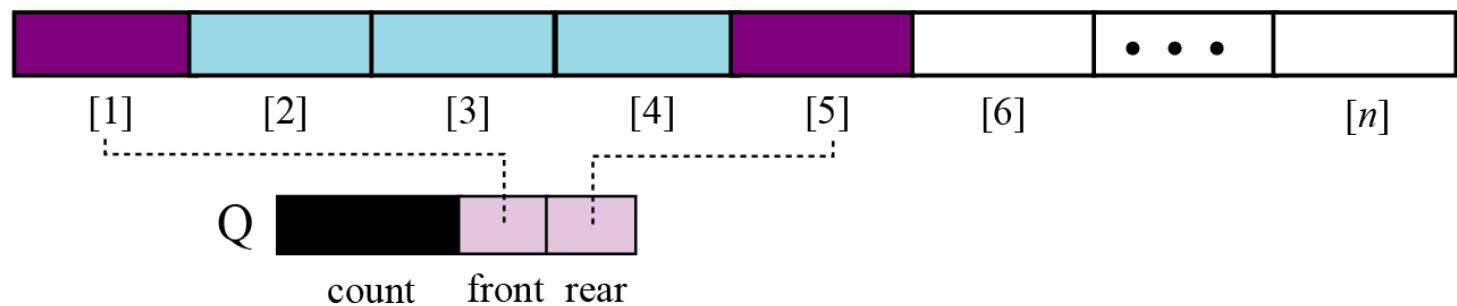
# Queues

- Queue Manipulation Operations
  - ◆ **isEmpty ()** : returns **true** or **false**
  - ◆ **first ()** : returns copy of value at front
  - ◆ **add (v)** : adds a new value at rear of queue **Enqueue**
  - ◆ **remove ()** : removes, returns value at front **Dequeue**

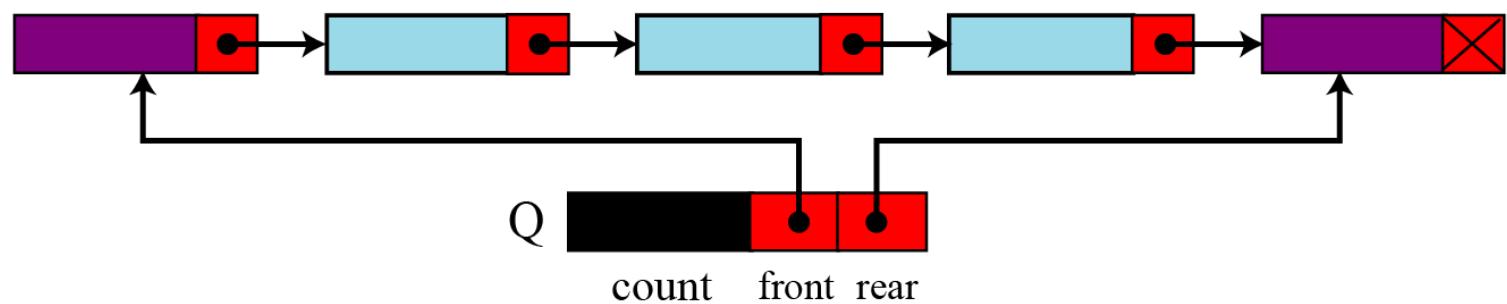
a. ADT



b. Array implementation



c. Linked list implementation



Queue implementation

# Implementing Stacks and Queues

- Using an ArrayList to implement Stack
- Use a Linked list to implement Queue

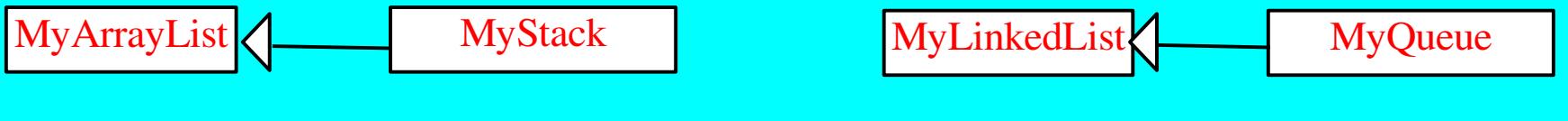
Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list.

Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list.

# Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

- ◆ Using inheritance: You can declare the stack class by extending the array list class, and the queue class by extending the linked list class.



- Using composition: You can declare an array list as a data field in the stack class, and a linked list as a data field in the queue class.



# Composition is Better

Both designs are fine, but using composition is better because it enables you to declare a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

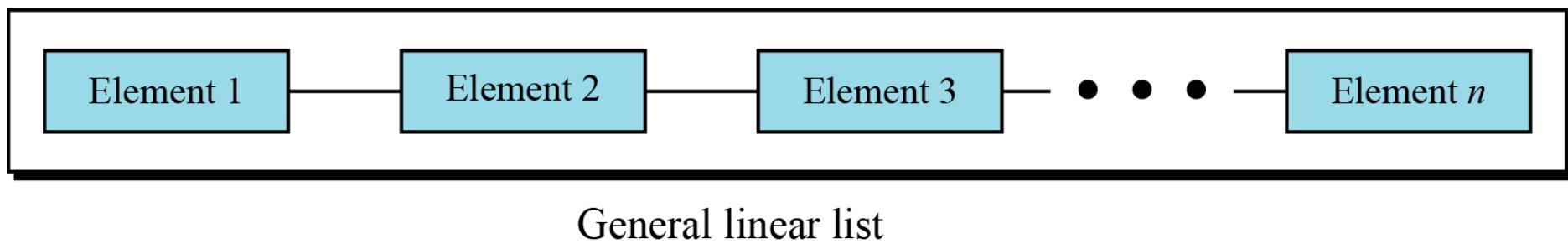
# MyStack and MyQueue

| MyStack                  |                                                              |
|--------------------------|--------------------------------------------------------------|
| -list: MyArrayList       |                                                              |
| +isEmpty(): boolean      | Returns true if this stack is empty.                         |
| +getSize(): int          | Returns the number of elements in this stack.                |
| +peek(): Object          | Returns the top element in this stack.                       |
| +pop(): Object           | Returns and removes the top element in this stack.           |
| +push(o: Object): Object | Adds a new element to the top of this stack.                 |
| +search(o: Object): int  | Returns the position of the specified element in this stack. |

| MyQueue                         |                                                 |
|---------------------------------|-------------------------------------------------|
| -list: MyLinkedList             |                                                 |
| +enqueue(element: Object): void | Adds an element to this queue.                  |
| +dequeue(): Object              | Removes an element from this queue.             |
| +getSize(): int                 | Returns the number of elements from this queue. |

# GENERAL LINEAR LISTS

- Stacks and queues defined in the two previous sections are **restricted linear lists**.
- A **general linear list** is a list in which operations, such as insertion and deletion, can be done anywhere in the list—at the beginning, in the middle or at the end. Figure shows a general linear list.



# Operations on general linear lists

Six common operations: *list*, *insert*, *delete*, *retrieve*, *traverse* and *empty*.

## The *list* operation

The list operation creates an empty list. The following shows the format:

**list (listName)**

# List features

- **ORDERING:** maintains order elements were added (new elements are added to the end by default)
- **DUPPLICATES:** yes (allowed)
- **OPERATIONS:** add element to end of list, insert element at given index, clear all elements, search for element, get element at given index, remove element at given index, get size
  - some of these operations are inefficient! (seen later)
- list manages its own size; user of the list does not need to worry about overfilling it

# Java's **List** interface

- Java also has an interface `java.util.List` to represent a list of objects:  
(a partial list)

`public void add(int index, Object o)`

Inserts the specified element at the specified position in this list.

`public Object get(int index)`

Returns the element at the specified position in this list.

`public int indexOf(Object o)`

Returns the index in this list of the first occurrence of the specified element, or `-1` if the list does not contain it.

# List interface, cont'd.

public int lastIndexOf (Object o)

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain it.

public Object remove (int index)

Removes the object at the specified position in this list.

public Object set (int index, Object o)

Replaces the element at the specified position in this list with the specified element.

- Notice that the methods added to Collection by List all deal with indexes; a list has indexes while a general collection may not

# Some list questions

- all of the list operations on the previous slide could be performed using an array instead!
- open question: What are some reasons why we might want to use a list class, rather than an array, to store our data?
- thought question: How might a List be implemented, under the hood?
- why do all the List methods use type Object?  
21

# List Iterations

# A particularly slow idiom

```
// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ":" + element);
}
```

- This code executes an  $O(n)$  operation (`get`) every time through a loop that runs  $n$  times!
  - Its runtime is  $O(n^2)$ , which is much worse than  $O(n)$
  - this code will take prohibitively long to run for large data sizes

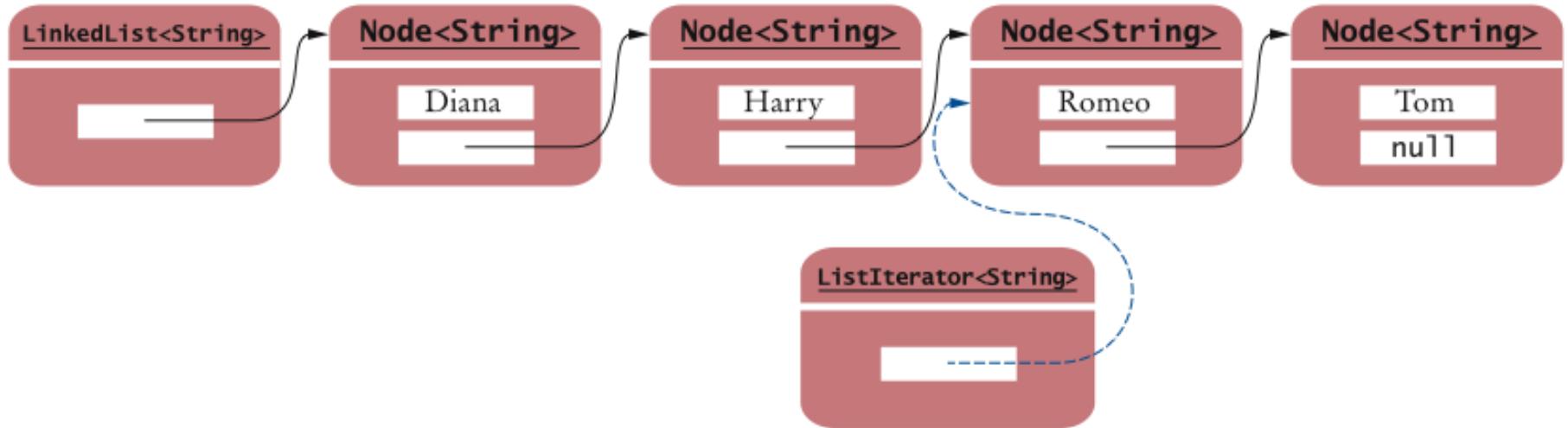
# The problem of position

- The code on the previous slide is wasteful because it throws away the position each time
  - every call to `get` has to re-traverse the list!
- it would be much better if we could somehow keep the list in place at each index as we looped through it
- Java uses special objects to represent a position of a collection as it's being examined...
  - these objects are called "iterators"

# List Iterator

- **ListIterator** type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

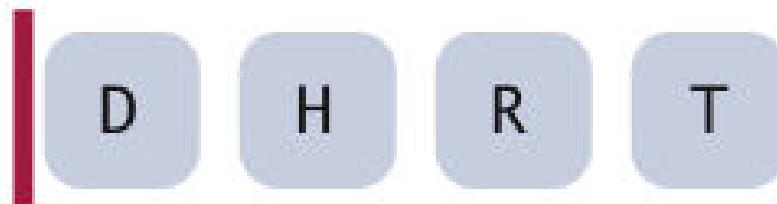
# A List Iterator



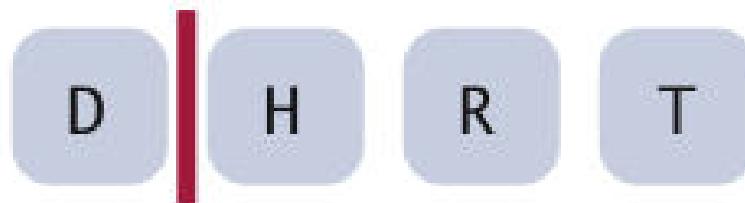
**Figure 2** A List Iterator

# A Conceptual View of the List Iterator

Initial ListIterator position



After calling `next`



After inserting J



**Figure 3** A Conceptual View of the List Iterator

# List Iterator

- Think of an iterator as pointing between two elements
  - *Analogy: Like the cursor in a word processor points between two characters*
- The listIterator method of the LinkedList class gets a list iterator

```
LinkedList<String> employeeNames = ...;  
ListIterator<String> iterator =  
employeeNames.listIterator();
```

# List Iterator

- Initially, the iterator points before the first element
- The next method moves the iterator:

**iterator.next();**

- next throws a **NoSuchElementException** if you are already past the end of the list
- hasNext returns true if there is a next element:

**if (iterator.hasNext())  
    iterator.next();**

# List Iterator

- The next method returns the element that the iterator is passing:

```
while iterator.hasNext()  
{  
    String name = iterator.next();  
    //Do something with name  
}
```

# List Iterator

- Shorthand for loop:

```
for (String name : employeeNames)  
{  
    // Do something with name  
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements

# List Iterator

- LinkedList is a *doubly linked list*
  - *Class stores two links:*
    - *One to the next element, and*
    - *One to the previous element*
- To move the list position backwards, use:
  - *hasPrevious*
  - *previous*

# Adding and Removing from a LinkedList

- The add method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element:*  
**iterator.add('Juliet');**

# Adding and Removing from a LinkedList

- The remove method
  - *Removes and*
  - *Returns the object that was returned by the last call to next or previous*

```
//Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
    String name = iterator.next();
    if (name fulfills condition)
        iterator.remove(); } 
```

# Adding and Removing from a LinkedList

- Be careful when calling remove:
  - *It can be called **only once** after calling next or previous:*

```
iterator.next();
iterator.next();
iterator.remove();
iterator.remove();
// Error: You cannot call remove twice.
```
  - *You cannot call it immediately after a call to add:*

```
iter.add("Fred");
iter.remove(); // Error: Can only call remove after
              // calling next or previous
```
  - *If you call it improperly, it throws an IllegalStateException*

# Methods of the ListIterator Interface

Table 2 Methods of the ListIterator Interface

|                                                                           |                                                                                                                                                                                            |
|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String s = iter.next();</code>                                      | Assume that <code>iter</code> points to the beginning of the list [Sally] before calling <code>next</code> . After the call, <code>s</code> is "Sally" and the iterator points to the end. |
| <code>iter.hasNext()</code>                                               | Returns <code>false</code> because the iterator is at the end of the collection.                                                                                                           |
| <code>if (iter.hasPrevious()) {<br/>    s = iter.previous();<br/>}</code> | <code>hasPrevious</code> returns <code>true</code> because the iterator is not at the beginning of the list.                                                                               |
| <code>iter.add("Diana");</code>                                           | Adds an element before the iterator position. The list is now [Diana, Sally].                                                                                                              |
| <code>iter.next();<br/>iter.remove();</code>                              | <code>remove</code> removes the last element returned by <code>next</code> or <code>previous</code> . The list is again [Diana].                                                           |

# Sample Program

- ListTester is a sample program that
  - *Inserts strings into a list*
  - *Iterates through the list, adding and removing elements*
  - *Prints the list*

# ListTester.java

```
1 import java.util.LinkedList;
2 import java.util.ListIterator;
3
4 /**
5     A program that tests the LinkedList class
6 */
7 public class ListTester
8 {
9     public static void main(String[] args)
10    {
11         LinkedList<String> staff = new LinkedList<String>();
12         staff.addLast("Diana");
13         staff.addLast("Harry");
14         staff.addLast("Romeo");
15         staff.addLast("Tom");
16
17         // | in the comments indicates the iterator position
18
19         ListIterator<String> iterator = staff.listIterator(); // |DHRT
20         iterator.next(); // D|HRT
21         iterator.next(); // DH|RRT
22 }
```

*Continued*

## ListTester.java (cont.)

```
23      // Add more elements after second element
24
25      iterator.add("Juliet"); // DHJ|RT
26      iterator.add("Nina"); // DHJN|RT
27
28      iterator.next(); // DHJNR|T
29
30      // Remove last traversed element
31
32      iterator.remove(); // DHJN|T
33
34      // Print all elements
35
36      for (String name : staff)
37          System.out.print(name + " ");
38      System.out.println();
39      System.out.println("Expected: Diana Harry Juliet Nina Tom");
40  }
41 }
```

*Continued*

# ListTester.java (cont.)

## Program Run:

Diana Harry Juliet Nina Tom

Expected: Diana Harry Juliet Nina Tom

## Self Check

Do linked lists take more storage space than arrays of the same size?

**Answer:** Yes, for two reasons. You need to store the node references, and each node is a separate object. (There is a fixed overhead to store each object in the virtual machine.)

## **Self Check**

Why don't we need iterators with arrays?

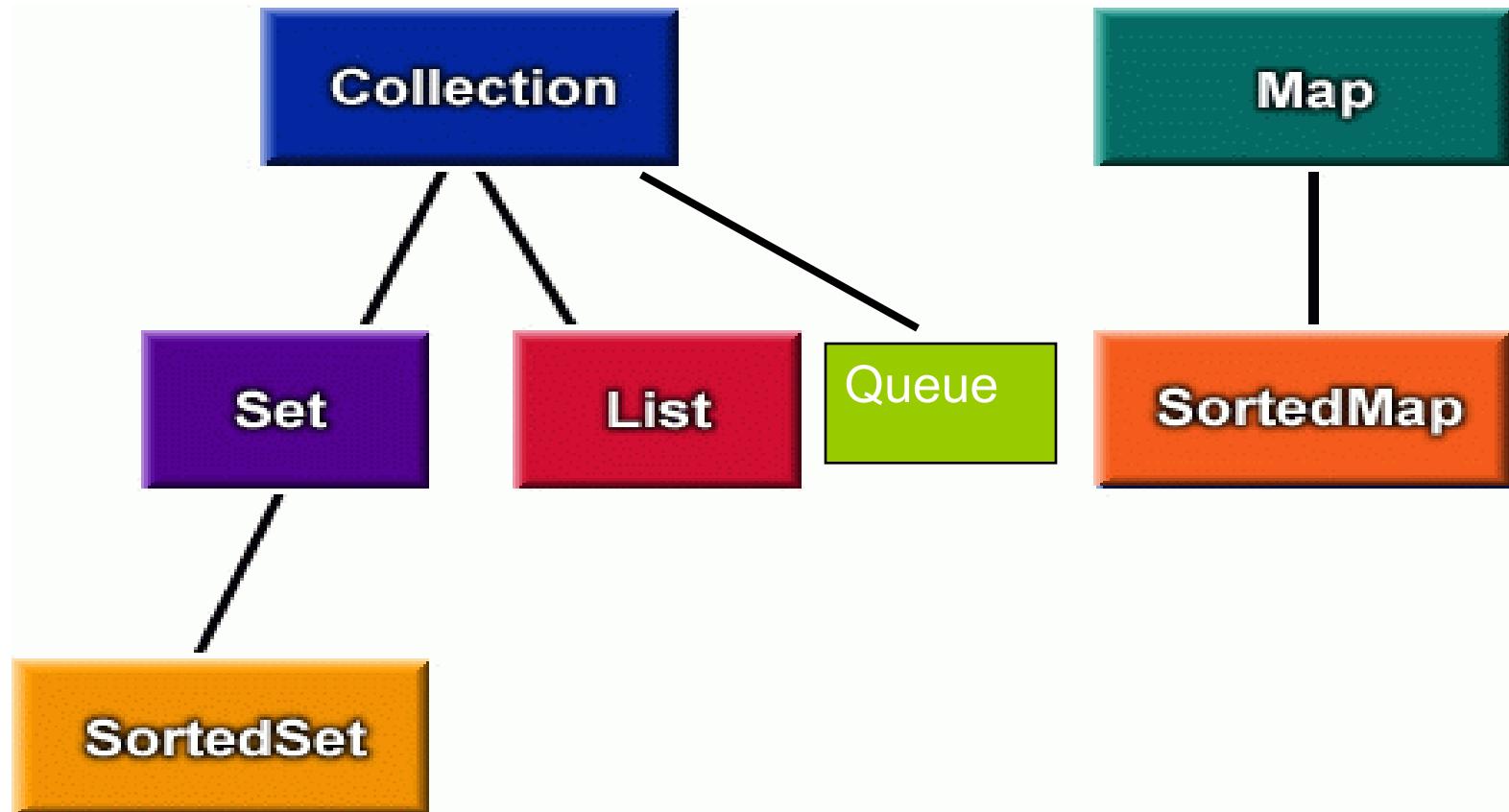
**Answer:** An integer index can be used to access any array location.

# Java Collection Framework

# Collection Framework

- ◆ A *collection framework* is a unified architecture for representing and manipulating collections. It has:
  - **Interfaces:** abstract data types representing collections
  - **Implementations:** concrete implementations of the collection interfaces
  - **Algorithms:** methods that perform useful computations, such as searching and sorting
    - These algorithms are said to be *polymorphic*: the same method can be used on different implementations

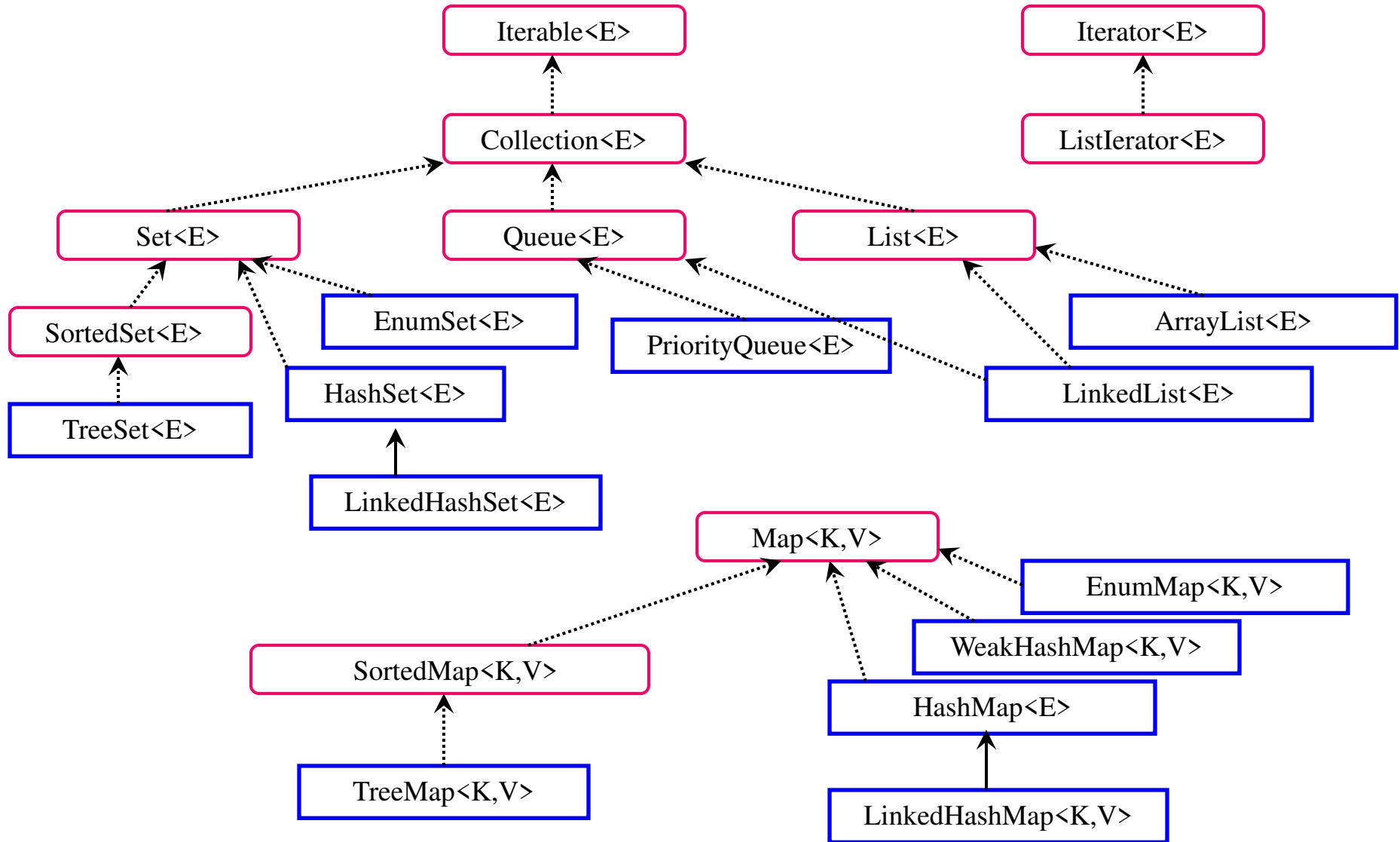
# Collection interfaces



# Collection Interface continued

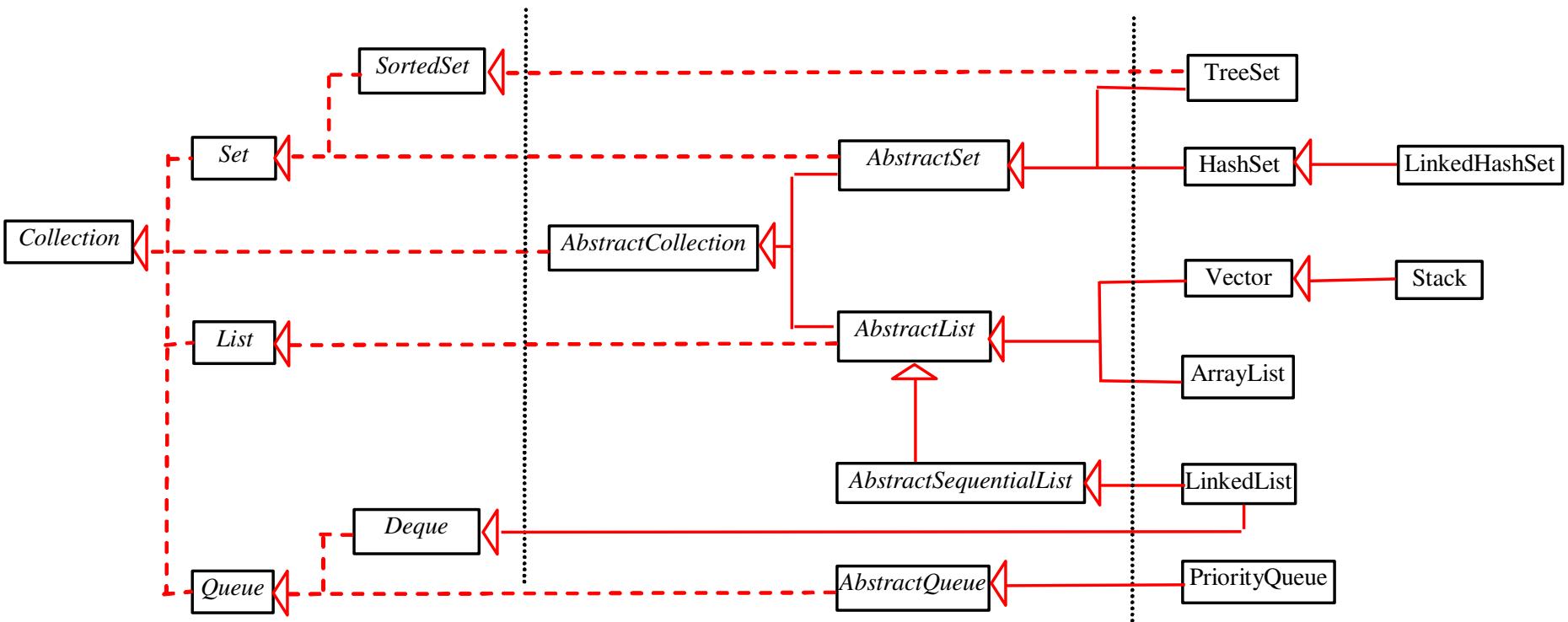
- **Set →**
  - ◆ The familiar set abstraction.
  - ◆ No duplicates; May or may not be ordered.
- **List →**
  - ◆ Ordered collection, also known as a sequence.
  - ◆ Duplicates permitted; Allows positional access
- **Map →**
  - ◆ A mapping from keys to values.
  - ◆ Each key can map to at most one value (function).
  - ◆ The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.
- **Queue →**
  - ◆ Ordered collection. FIFO (First In First Out)

# Type Trees for Collections

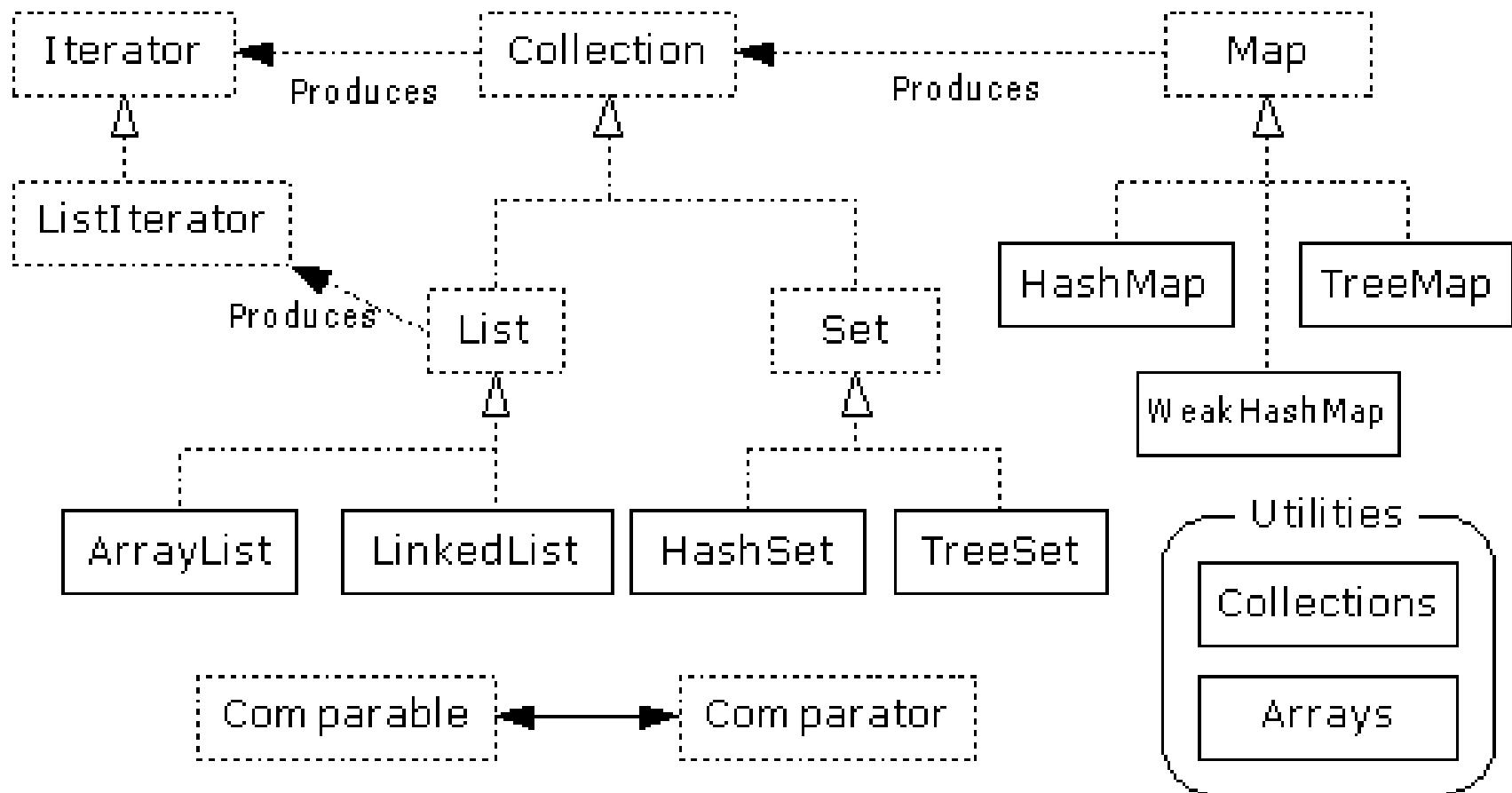


# Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



# Collections Framework Diagram



# Collection Interface

- Defines fundamental methods
  - ◆ **int size();**
  - ◆ **boolean isEmpty();**
  - ◆ **boolean contains(Object element);**
  - ◆ **boolean add(Object element); // Optional**
  - ◆ **boolean remove(Object element); // Optional**
  - ◆ **Iterator iterator();**
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

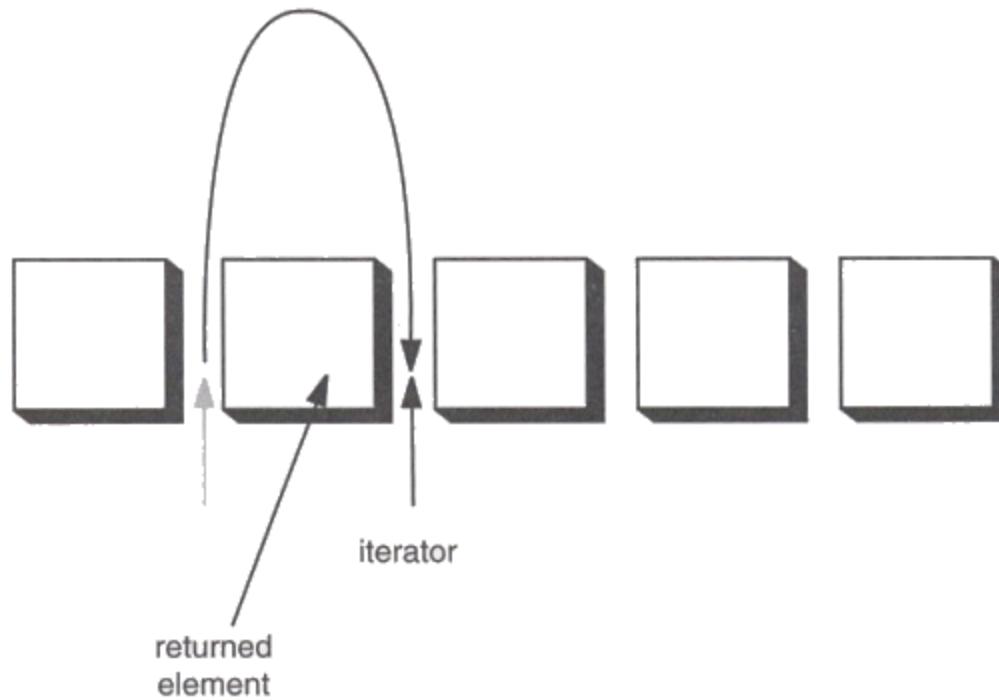
# Interface Collection

- |                  |                        |
|------------------|------------------------|
| • add(o)         | Add a new element      |
| • addAll(c)      | Add a collection       |
| • clear()        | Remove all elements    |
| • contains(o)    | Membership checking.   |
| • containsAll(c) | Inclusion checking     |
| • isEmpty()      | Whether it is empty    |
| • iterator()     | Return an iterator     |
| • remove(o)      | Remove an element      |
| • removeAll(c)   | Remove a collection    |
| • retainAll(c)   | Keep the elements      |
| • size()         | The number of elements |

# Iterator Interface

- Defines three fundamental methods
  - ◆ **Object next()**
  - ◆ **boolean hasNext()**
  - ◆ **void remove()**
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - ◆ Then you can use it or remove it

# Iterator Position

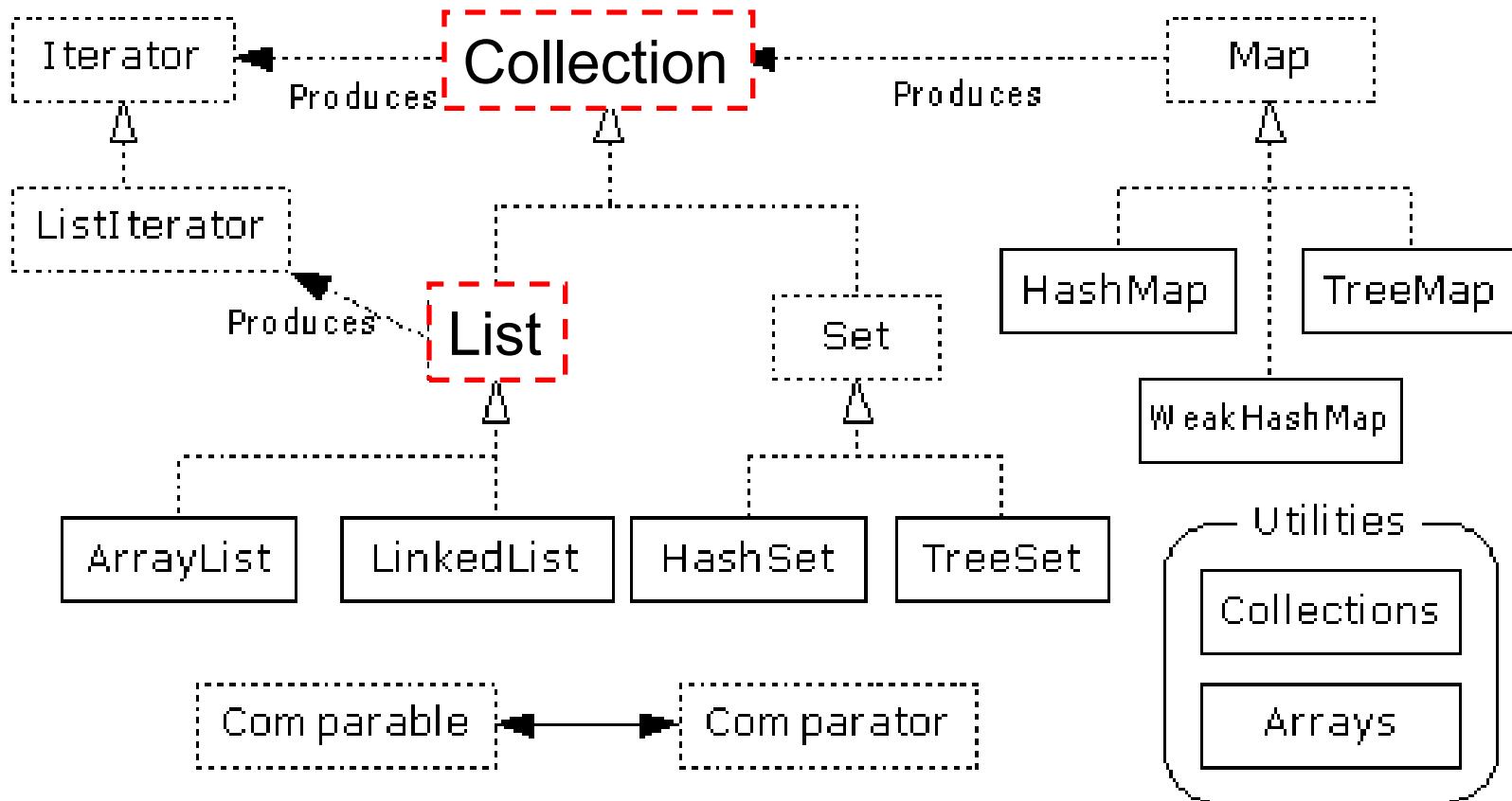


**Figure 2–3: Advancing an iterator**

# Example - SimpleCollection

```
public class SimpleCollection {  
    public static void main(String[] args) {  
        Collection c;  
        c = new ArrayList();  
        System.out.println(c.getClass().getName());  
        for (int i=1; i <= 10; i++) {  
            c.add(i + " * " + i + " = "+i*i);  
        }  
        Iterator iter = c.iterator();  
        while (iter.hasNext())  
            System.out.println(iter.next());  
    }  
}
```

# List Interface Context



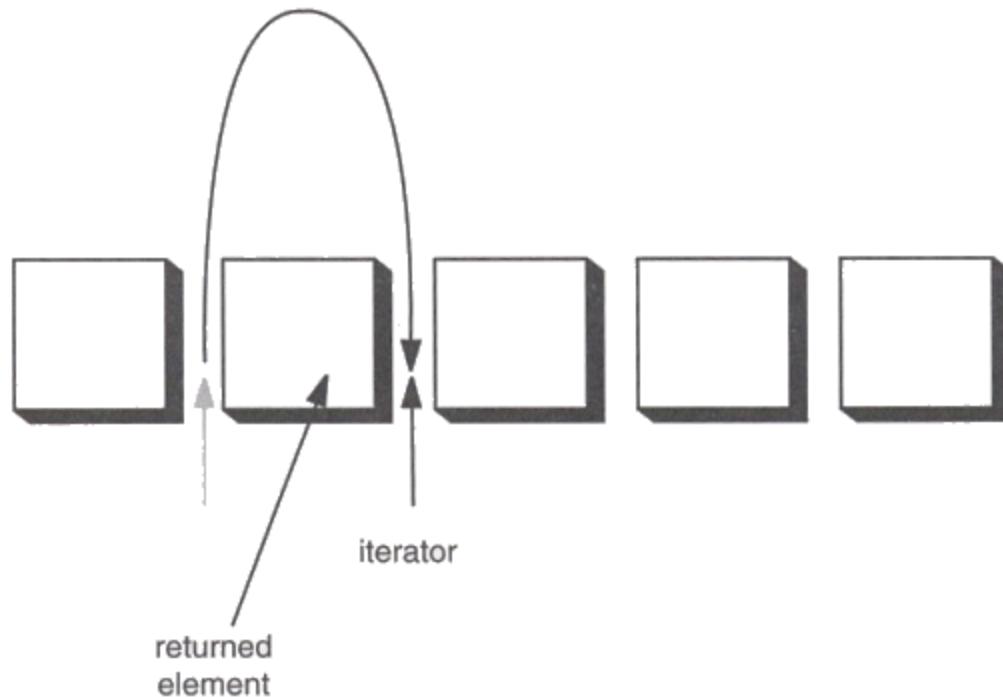
# List Interface

- The List interface adds the notion of *order* to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow *duplicate* elements
- Provides a **ListIterator** to step through the elements in the list.

# ListIterator Interface

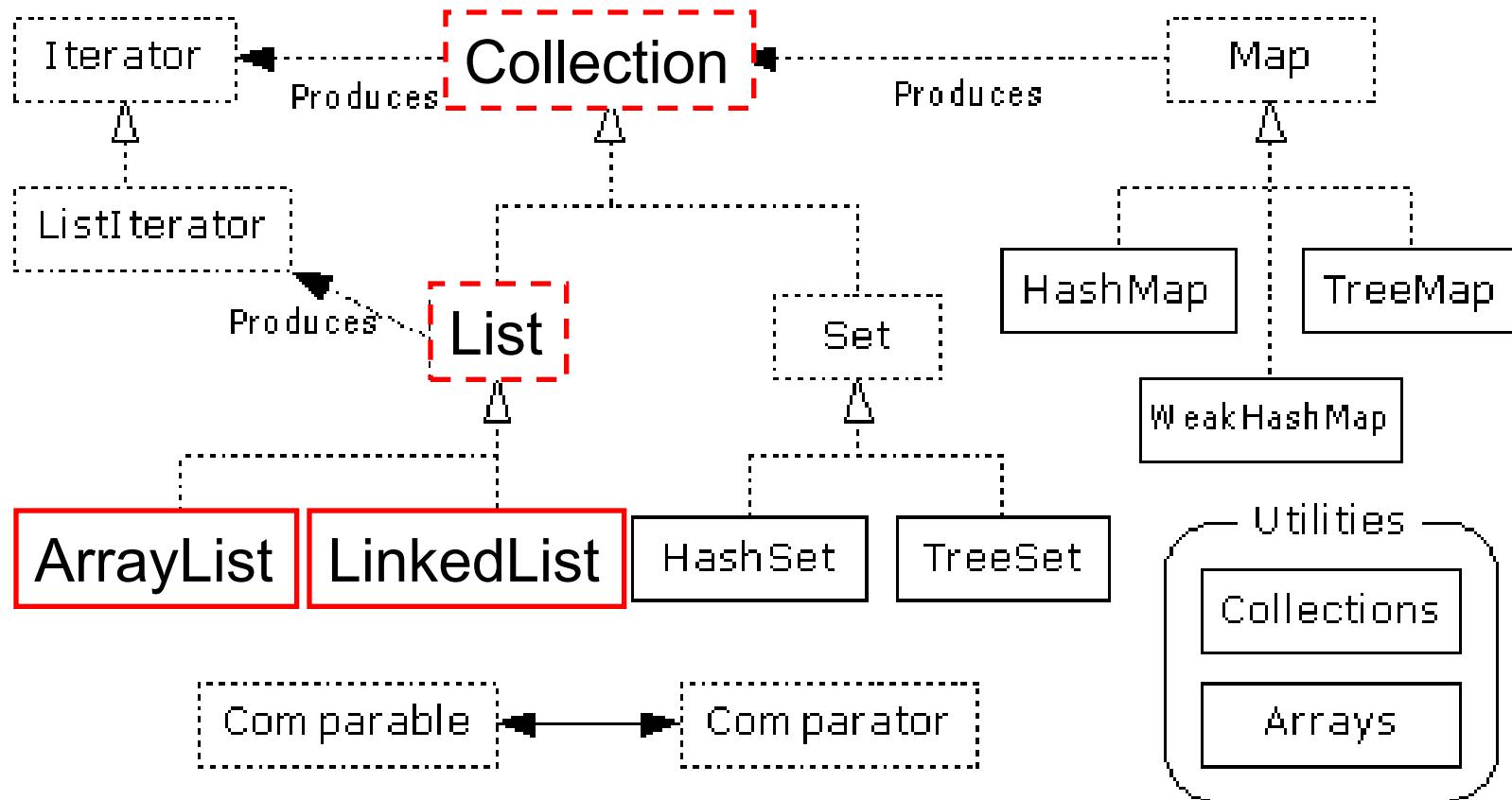
- Extends the Iterator interface
- Defines three fundamental methods
  - ◆ **void add(Object o)** - before current position
  - ◆ **boolean hasPrevious()**
  - ◆ **Object previous()**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

# Iterator Position - `next()`, `previous()`



**Figure 2–3: Advancing an iterator**

# ArrayList and LinkedList Context



# List Implementations

- **ArrayList**
  - ◆ low cost random access
  - ◆ high cost insert and delete
  - ◆ array that resizes if need be
- **LinkedList**
  - ◆ sequential access
  - ◆ low cost insert and delete
  - ◆ high cost random access

# ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {  
    super();  
    if (initialCapacity < 0)  
        throw new IllegalArgumentException(  
            "Illegal Capacity: "+initialCapacity);  
    this.elementData = new Object[initialCapacity];  
}
```

# 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)**
- Indexed add and remove are provided, but can be costly if used frequently
  - ◆ **void add(int index, Object element)**
  - ◆ **Object remove(int index)**
- May want to resize in one shot if adding many elements
  - ◆ **void ensureCapacity(int minCapacity)**

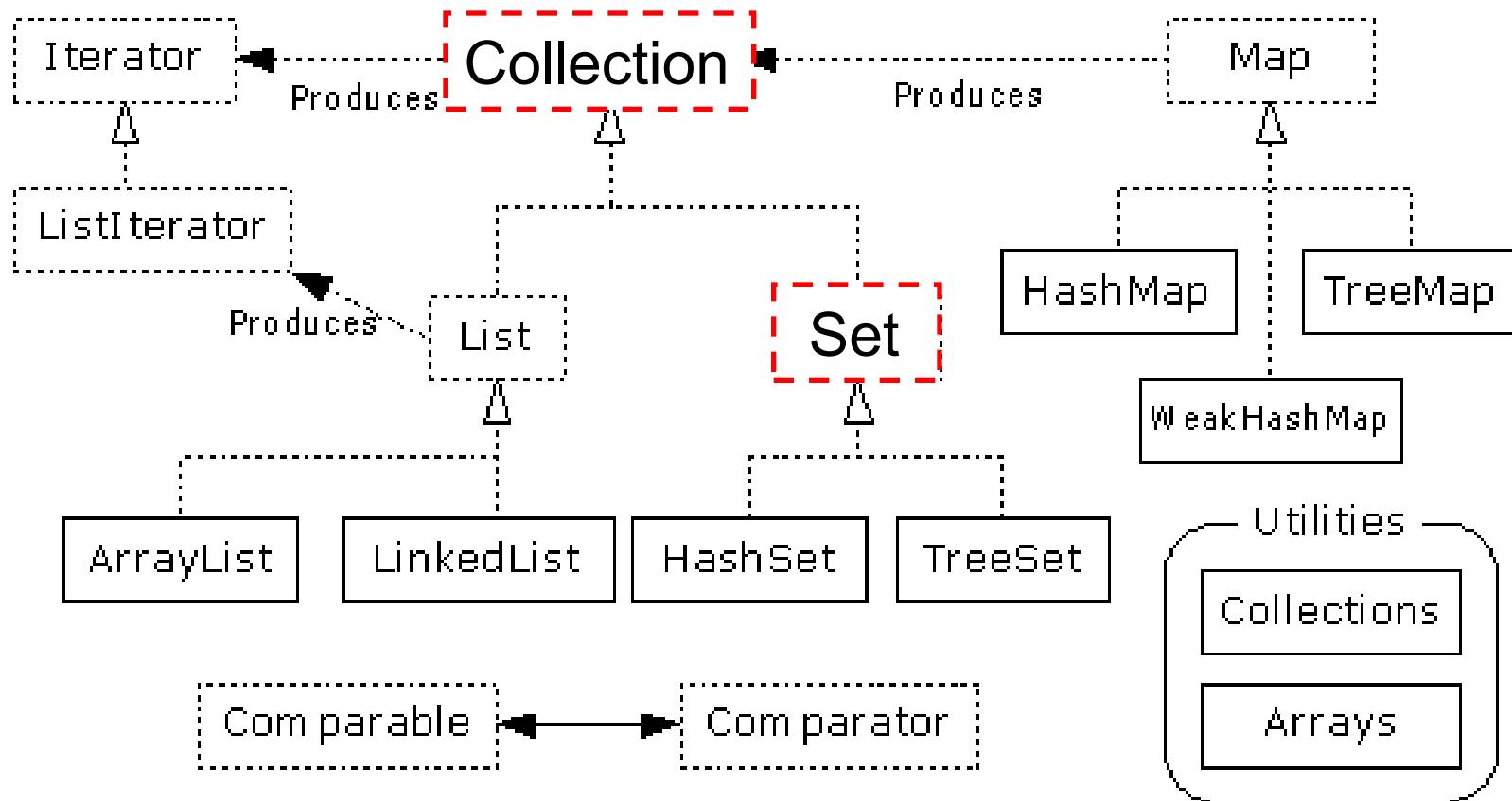
# LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - ◆ just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - ◆ Start from beginning or end and traverse each node while counting

# LinkedList methods

- The list is sequential, so access it that way
  - ◆ **ListIterator listIterator()**
- ListIterator knows about position
  - ◆ use **add()** from ListIterator to add at a position
  - ◆ use **remove()** from ListIterator to remove at a position
- LinkedList knows a few things too
  - ◆ **void addFirst(Object o), void addLast(Object o)**
  - ◆ **Object getFirst(), Object getLast()**
  - ◆ **Object removeFirst(), Object removeLast()**

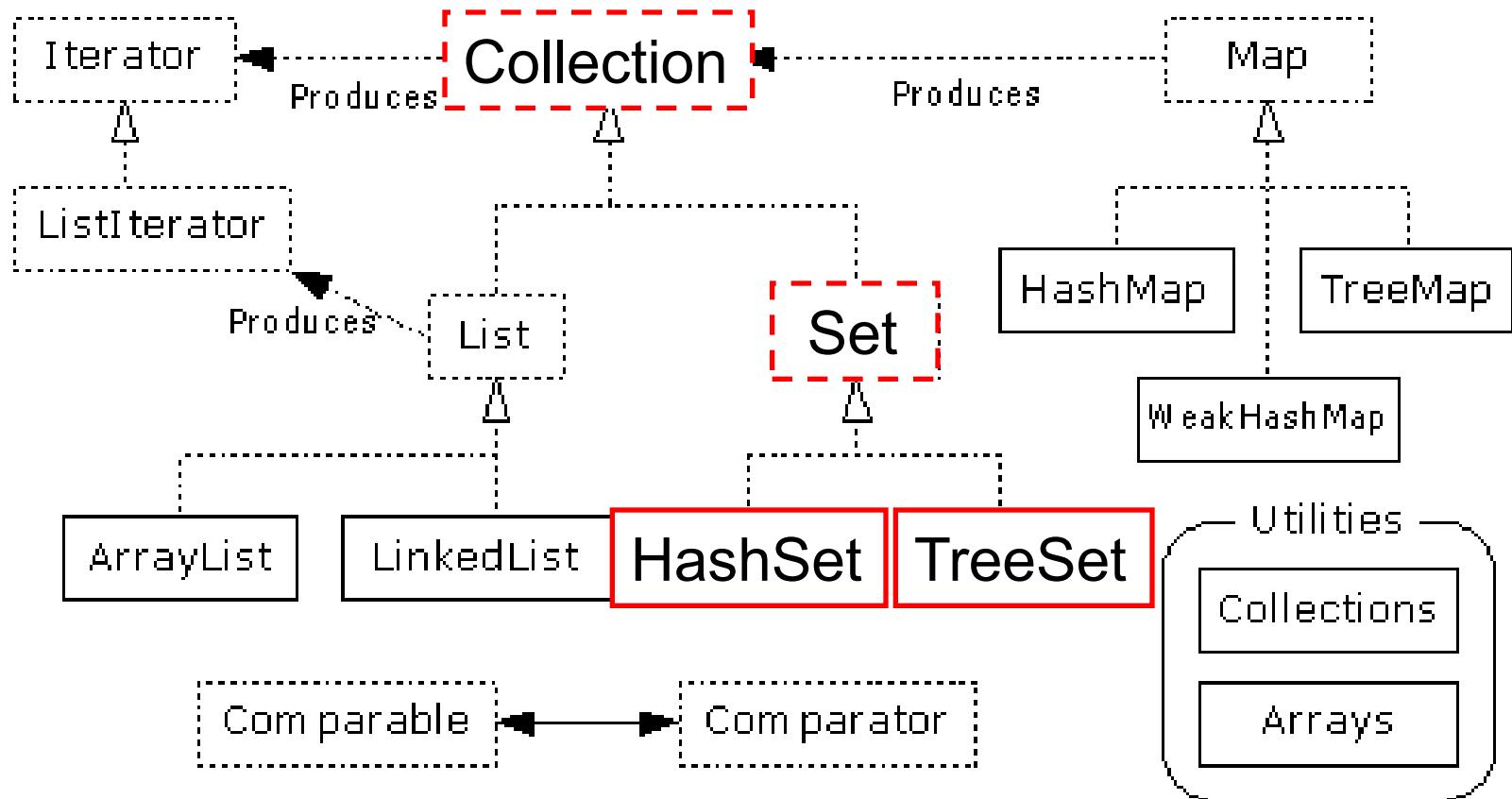
# Set Interface Context



# Set Interface

- Same methods as Collection
  - ◆ different contract - no duplicate entries
- Defines two fundamental methods
  - ◆ **boolean add(Object o)** - reject duplicates
  - ◆ **Iterator iterator()**
- Provides an Iterator to step through the elements in the Set
  - ◆ No guaranteed order in the basic Set interface
  - ◆ There is a SortedSet interface that extends Set

# HashSet and TreeSet Context



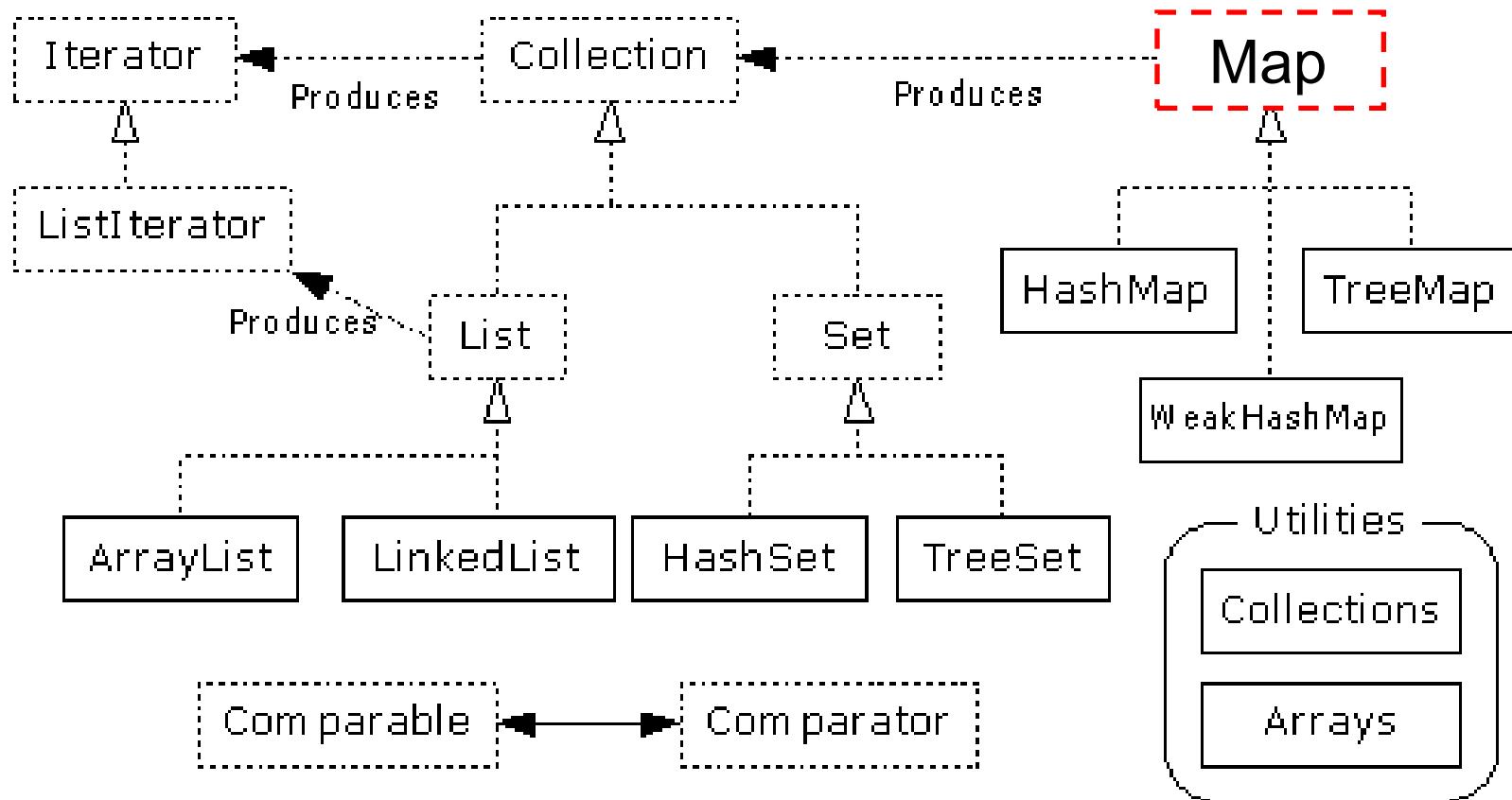
# HashSet

- Find and add elements very quickly
  - ◆ uses hashing implementation in HashMap
- 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
- The **hashCode()** method and the **equals()** method must be compatible
  - ◆ if two objects are equal, they must have the same **hashCode()** value

# TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
- An iterator always presents them in order
- Default order is defined by natural order
  - ◆ objects implement the Comparable interface
  - ◆ TreeSet uses **compareTo(Object o)** to sort

# Map Interface Context



# Map Interface

- Stores **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

# Map methods

**Object put(Object key, Object value)**

**Object get(Object key)**

**Object remove(Object key)**

**boolean containsKey(Object key)**

**boolean containsValue(Object value)**

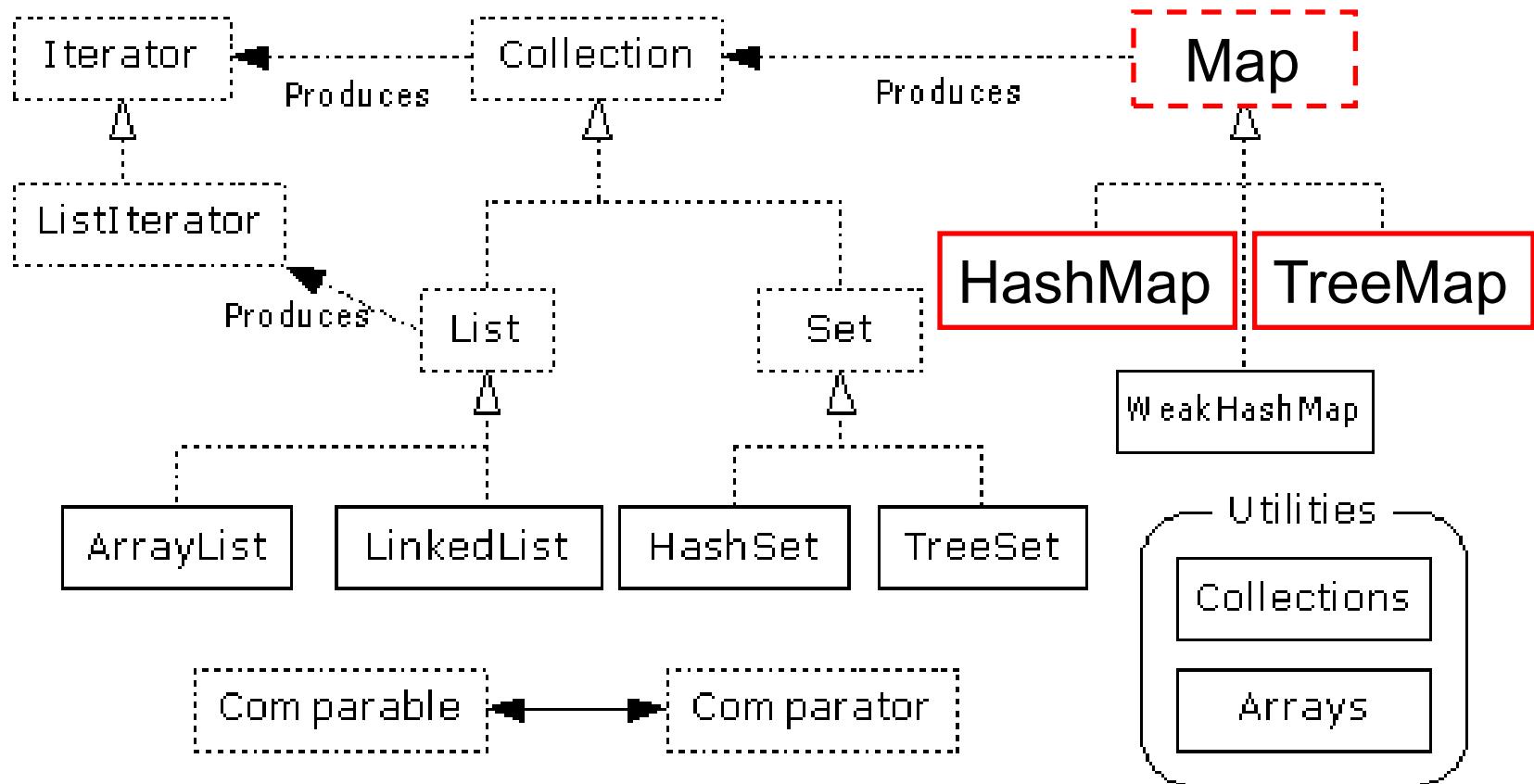
**int size()**

**boolean isEmpty()**

# Map views

- A means of iterating over the keys and values in a Map
- **Set keySet()**
  - ◆ returns the Set of keys contained in the Map
- **Collection values()**
  - ◆ returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- **Set entrySet()**
  - ◆ returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

# HashMap and TreeMap Context



# HashMap and TreeMap

- HashMap
  - ◆ The keys are a set - unique, unordered
  - ◆ Fast
- TreeMap
  - ◆ The keys are a set - unique, ordered
  - ◆ Same options for ordering as a TreeSet
    - *Natural order (Comparable, compareTo(Object))*
    - *Special order (Comparator, compare(Object, Object))*

# Bulk Operations

- In addition to the basic operations, a Collection may provide “bulk” operations

**boolean containsAll(Collection c);**

**boolean addAll(Collection c); // Optional**

**boolean removeAll(Collection c); // Optional**

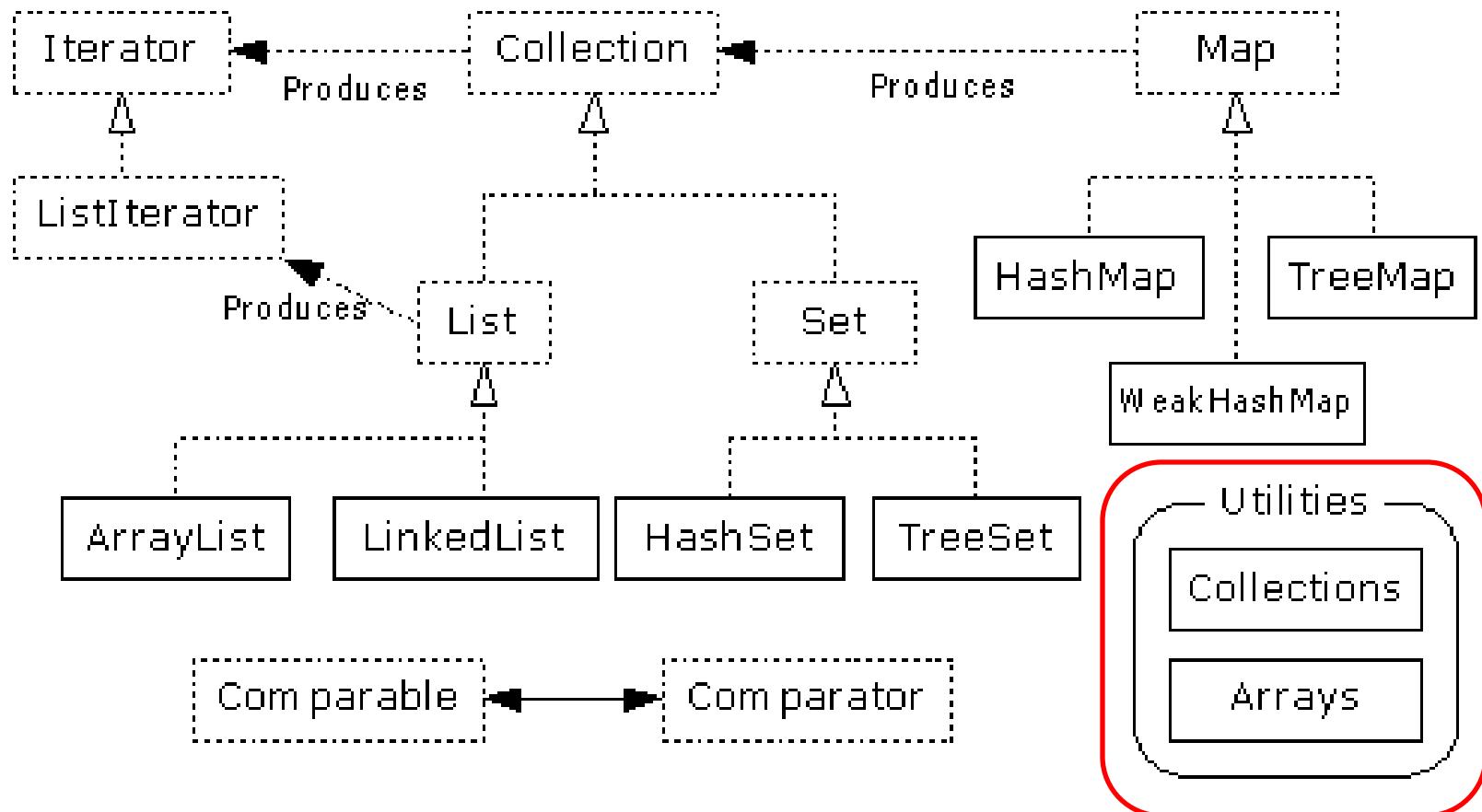
**boolean retainAll(Collection c); // Optional**

**void clear(); // Optional**

**Object[] toArray();**

**Object[] toArray(Object a[]);**

# Utilities Context



# Utilities

- The Collections class provides a number of static methods for fundamental algorithms
- Most operate on Lists, some on all Collections
  - ◆ Sort, Search, Shuffle
  - ◆ Reverse, fill, copy
  - ◆ Min, max
- Wrappers
  - ◆ synchronized Collections, Lists, Sets, etc
  - ◆ unmodifiable Collections, Lists, Sets, etc

# Concrete Collections

| concrete collection | implements | description          |
|---------------------|------------|----------------------|
| HashSet             | Set        | hash table           |
| TreeSet             | SortedSet  | balanced binary tree |
| ArrayList           | List       | resizable-array      |
| LinkedList          | List       | linked list          |
| Vector              | List       | resizable-array      |
| HashMap             | Map        | hash table           |
| TreeMap             | SortedMap  | balanced binary tree |

# General Purpose Implementations

|             |                              |                             |                                |                    |
|-------------|------------------------------|-----------------------------|--------------------------------|--------------------|
|             | <b>Hash Table</b>            | <b>Resizable array</b>      | <b>balanced tree</b>           | <b>linked list</b> |
| <b>Set</b>  | <b>HashSet</b>               |                             | <b>TreeSet<br/>(sortedSet)</b> |                    |
| <b>List</b> |                              | <b>ArrayList<br/>Vector</b> |                                | <b>LinkedList</b>  |
| <b>Map</b>  | <b>HashMap<br/>Hashtable</b> |                             | <b>TreeMap<br/>(sortedMap)</b> |                    |

# The Arrays Class

The Arrays class contains various static methods

- sorting arrays
- searching arrays
- comparing arrays
- filling array elements
- convert array to list.

# The Arrays Class UML Diagram

| Arrays                                                                                         |                                                                                                                                                  |
|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>+asList(a: Object[]): List</code>                                                        | Returns a list from an array of objects                                                                                                          |
| Overloaded binarySearch method for byte, char, short, int, long, float, double, and Object.    | Overloaded binary search method to search a key in the array of byte, char, short, int, long, float, double, and Object                          |
| <code>+binarySearch(a: xType[], key: xType): int</code>                                        |                                                                                                                                                  |
| Overloaded equals method for boolean, byte, char, short, int, long, float, double, and Object. | Overloaded equals method that returns true if a is equal to a2 for a and a2 of the boolean, byte, char, short, int, long, float, and Object type |
| <code>+equals(a: xType[], a2: xType[]): boolean</code>                                         |                                                                                                                                                  |
| Overloaded fill method for boolean char, byte, short, int, long, float, double, and Object.    | Overloaded fill method to fill in the specified value into the array of the boolean, byte, char, short, int, long, float, and Object type        |
| <code>+fill(a: xType[], val: xType): void</code>                                               |                                                                                                                                                  |
| <code>+fill(a: xType[], fromIndex: int, toIndex: xType, val: xType): void</code>               |                                                                                                                                                  |
| Overloaded sort method for char, byte, short, int, long, float, double, and Object.            | Overloaded sort method to sort the specified array of the char, byte, short, int, long, float, double, and Object type                           |
| <code>+sort(a: xType[]): void</code>                                                           |                                                                                                                                                  |
| <code>+sort(a: xType[], fromIndex: int, toIndex: int): void</code>                             |                                                                                                                                                  |

# Convert to and from an Array

```
import java.util.*;  
public class G{  
    public static void main(String[] args) {  
  
        List<String> sun = new ArrayList<String>();  
        sun.add("Feel"); sun.add("the");  
        sun.add("power");  
        sun.add("of"); sun.add("the"); sun.add("Sun");  
  
        String[] s1 = sun.toArray(new String[0]);  
        //Collection to array  
        for(int i = 0; i < s1.length; ++i){  
            String contents = s1[i];  
            System.out.print(contents); }  
  
        System.out.println();
```

# Enhanced for loop

```
List<String> sun2 = Arrays.asList(s1);  
//Array back to Collection  
for(String s2: sun2)  
{ String s3 = s2;  
    System.out.print(s3);  
}  
}  
}
```

# A note on iterators

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator () method. The following is the Iterator interface.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); //optional  
}
```

# Iterate Through Collections

- The Iterator interface:

```
interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

- The iterator() method defined in the Collection interface:

```
Iterator iterator()
```

# Iterators

- ♦ Iterators provide a general way to traverse all elements in a collection

```
ArrayList<String> list = new ArrayList<String>();  
list.add("1-FiRsT");  
list.add("2-SeCoND");  
list.add("3-ThIrD");  
Iterator<String> itr = list.iterator();  
while (itr.hasNext()) {  
    System.out.println(itr.next().toLowerCase());  
}
```

*Output*

1-first  
2-second  
3-third

# Enhanced for loop

- ◆ If a class **extends Iterable<E>**
- ◆ (e.g. class **Set<E>** implements Iterable),
- ◆ Java's enhanced for loop of this general form

```
for (E refVar : collection<E> ) {  
    refVar refers to each element in collection<E>  
}
```

— example

```
ArrayList<String> list = new ArrayList<String>();  
list.add("first");  
for (String s : list)  
    System.out.println(s.toLowerCase());
```

# Map and SortedMap

- ◆ The Map interface defines methods
  - **get, put, contains, keySet, values, entrySet**
- ◆ **TreeMap** implements **Map**
  - put, get, remove:  $O(\log n)$
- ◆ **HashMap** implements **Map**
  - put, get, remove:  $O(1)$

# Set and SortedSet

- ◆ Some Map methods return **Set**
- ◆ The Set interface
  - `add`, `addAll`, `remove`, `size`, but no `get!`
- ◆ Some implementations
  - **TreeSet**: values stored in order,  $O(\log n)$
  - **HashSet**: values in a hash table, no order,  $O(1)$

# Choosing the datatype

- When you declare a Set, List or Map, you should use Set, List or Map interface as the datatype instead of the implementing class. That will allow you to change the implementation by changing a single line of code!

```
import java.util.*;  
  
public class Test {  
    public static void main(String[] args) {  
        Set<String> ss = new LinkedHashSet<String>();  
  
        for (int i = 0; i < args.length; i++)  
            ss.add(args[i]);  
  
        Iterator i = ss.iterator();  
        while (i.hasNext())  
            System.out.println(i.next());  
    }  
}
```

```
import java.util.*;  
  
public class Test {  
  
    public static void main(String[] args)  
    {  
        //map to hold student grades  
        Map<String, Integer> theMap = new  
        HashMap<String, Integer>();  
  
        theMap.put("Korth, Evan", 100);  
        theMap.put("Plant, Robert", 90);  
        theMap.put("Coyne, Wayne", 92);  
        theMap.put("Franti, Michael", 98);  
        theMap.put("Lennon, John", 88);  
  
        System.out.println(theMap);  
        System.out.println("-----");  
        System.out.println(theMap.get("Korth,  
Evan"));  
        System.out.println(theMap.get("Franti,  
Michael"));    } }  
}
```

# *Using Sets to find duplicate elements*

```
import java.util.*;  
  
public class FindDups {  
    public static void main(String[] args) {  
        Set<String> s = new HashSet<String>();  
        for (String a : args)  
            if (!s.add(a))  
                System.out.println("Duplicate  
detected: " + a);  
  
        System.out.println(s.size() + "  
distinct words: " + s);  
    } }
```

# Which class should I use?

- The difference between the different classes is how the structure is implemented.
  - This generally has an impact on performance.
- Use Vector
  - Fast access to elements using index
  - Optimized for storage space
  - Not optimized for inserts and deletes
- Use ArrayList
  - Same as Vector except the methods are not synchronized.  
    Better performance
- Use linked list
  - Fast inserts and deletes
  - Stacks and Queues (accessing elements near the beginning or end)
  - Not optimized for random access

# Which class should I use?

---

- Use Sets
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

| Data Structures | Advantages                                              | Disadvantages                                     |
|-----------------|---------------------------------------------------------|---------------------------------------------------|
| Array           | Quick insertion,<br>very fast access if<br>index known. | Slow search, slow<br>deletion, and fixed<br>size. |
| Ordered array   | Quicker search<br>than unsorted array                   | Slow insertion and<br>deletion, fixed size        |
| Stack           | Last in first out                                       | Slow access to<br>other items                     |
| Queue           | First in first out<br>access.                           | Slow access to<br>other items                     |
| Linked list     | Quick insertion,<br>quick deletion                      | Slow search                                       |
| ArrayList       | Random Access                                           | Slow insertion,<br>deletion                       |

# Java's Collection Framework: Examples

# Using Set

```
Set set = new HashSet();           // instantiate a concrete set
set.add(obj);                     // insert an elements
int n = set.size();               // get size
if (set.contains(obj)) { ... }    // check membership

Iterator iter = set.iterator();    // iterate through the set
while (iter.hasNext()) {
    Object e = iter.next();
    // ... }
```

# Using Map

```
Map map = new HashMap(); map.put(key, val);  
                                // insert a key-value pair  
                                // get the value associated with key  
  
Object val = map.get(key);  
map.remove(key);  
// ...  
if (map.containsValue(val)) { ... }  
if (map.containsKey(key)) { ... }  
  
Set keys = map.keySet();  
                                // get the set of keys  
  
Iterator iter = keys.iterator();  
while (iter.hasNext()) {  
    Key key = (Key) iter.next();  
    // ... }
```

# Map views

- **Set<K> keySet()**
  - ◆ Returns a set view of the keys contained in this map.
- **Collection<V> values()**
  - ◆ Returns a collection view of the values contained in this map
  - ◆ Can't be a set—keys must be unique, but values may be repeated

## Map views

- `Set<Map.Entry<K, V>> entrySet()`
  - ◆ Returns a set view of the mappings contained in this map.
- A view is *dynamic access* into the Map
  - ◆ If you change the Map, the view changes
  - ◆ If you change the view, the Map changes
- The Map interface does not provide any Iterators
  - ◆ However, there are iterators for the above Sets and Collections

```
import java.util.HashMap;
import java.util.Set;

public class HashMapEntrySet1 {
    public static void main(String[] args) {
        //Creating an object of HashMap class
        HashMap<String, Integer> map = new HashMap<String, Integer>(6);

        //Putting key-value pairs inside map
        map.put("Java", 1);
        map.put("is", 2);
        map.put("the", 3);
        map.put("best", 4);
        map.put("programming", 5);
        map.put("language", 6);

        //Creating a Set
        Set set = map.entrySet();

        //Displaying all entries in Set
        System.out.println(set);
    }
}
```

[the=3, Java=1, is=2, best=4, language=6, programming=5]

# Using TreeMap

```
TreeMap tm = new TreeMap();
tm.put("Zara", new Double(3434.34)); ...
```

Set set = tm.entrySet(); *//Map does not implement the Iterator*

```
Iterator i = set.iterator();
while(i.hasNext()) {
    Map.Entry me = (Map.Entry)i.next(); //a collection-view of the map
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue()); }
```

```
double balance = ((Double)tm.get("Zara")).doubleValue();
tm.put("Zara", new Double(balance + 1000));
System.out.println("Zara's new balance: " + tm.get("Zara")); } }
```

# Using Vector

```
Vector v = new Vector(3, 2);      // initial size is 3, increment is 2
```

```
System.out.println("Initial size: " + v.size());
```

```
System.out.println("Initial capacity: " + v.capacity());
```

```
v.addElement(new Integer(1)); .....
```

```
System.out.println("Capacity after four additions: " + v.capacity());
```

```
v.addElement(new Double(5.45));
```

```
System.out.println("Current capacity: " + v.capacity());
```

```
v.addElement(new Double(6.08)); ....
```

```
System.out.println("First element: " + (Integer)v.firstElement());
```

```
System.out.println("Last element: " + (Integer)v.lastElement());
```

```
if(v.contains(new Integer(3)))
```

```
    System.out.println("Vector contains 3.");}
```

# Using ListIterator

For collections that implement List, you can also obtain an iterator by calling ListIterator which can traverse the list in either direction

```
ArrayList al = new ArrayList();
```

```
ListIterator litr = al.listIterator();
while(litr.hasNext()) {
    Object element = litr.next(); .... }
```

```
// Now, display the list backwards
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
    Object element = litr.previous();
    System.out.print(element + " "); }
```

# Ordering and Sorting

There are two ways to define orders on objects.

- Each class can define a *natural order* among its instances by implementing the Comparable interface.

```
int compareTo(Object o)
```

- Arbitrary orders among different objects can be defined by *comparators*, classes that implement the Comparator interface.

```
int compare(Object o1, Object o2)
```

This method returns zero if the objects are equal. It returns a positive value if o1 is greater than o2. Otherwise, a negative value is returned.

# User-Defined Order

Reverse alphabetical order of strings

```
public class StringComparator
    implements Comparator {
    public int compare(Object o1, Object o2)
    {
        if (o1 != null &&
            o2 != null &&
            o1 instanceof String &&
            o2 instanceof String) {
            String s1 = (String) o1;
            String s2 = (String) o2;
            return - (s1.compareTo(s2));
        } else {
            return 0;
        }
    }
}
```

# Object Oriented Programming

## Generics

# Why generics?

- `Person[] people = new Person[25]; // you must say what's in the array`  
`people[0] = "Sally"; // syntax error`
- `ArrayList people = new ArrayList(); // but anything could go in the ArrayList!`  
`people.add("Sally");`  
 `// sometime later...`  
`Person p = (Person)people.get(0); // runtime error`
- `ArrayList<Person> people = new ArrayList<Person>(); // say what's in it`  
`people.add("Sally"); // syntax error`
- Since Java 5, collections should be used only with generics

# Generics

- A **generic** is a method that is recompiled with different types as the need arises
- The bad news:
  - Instead of saying: `List words = new ArrayList();`
  - You'll have to say:  
`List<String> words = new ArrayList<String>();`
- The good news:
  - Replaces runtime type checks with compile-time checks
  - No casting; instead of  
`String title = (String) words.get(i);`  
you use  
`String title = words.get(i);`
- Some classes and interfaces that have been “genericized” are:  
`Vector`, `ArrayList`, `LinkedList`, `Hashtable`, `HashMap`, `Stack`,  
`Queue`, `PriorityQueue`, `Dictionary`, `TreeMap` and `TreeSet`

# Genericized types are still types

# Generic Iterators

- To iterate over generic collections, it's a good idea to use a generic iterator

```
– List<String> listOfStrings = new  
LinkedList<String>();  
  
...  
for (Iterator<String> i =  
listOfStrings.iterator(); i.hasNext(); ) {  
    String s = i.next();  
    System.out.println(s);  
}
```

# Type wildcards

- Here's a simple (no generics) method to print out any list:

```
– private void printList(List list) {  
    for (Iterator i = list.iterator(); i.hasNext(); ) {  
        System.out.println(i.next());  
    }  
}
```

- The above still works in Java, but now it generates warning messages
- You should eliminate *all* errors and warnings in your final code, so you need to *tell* Java that any type is acceptable:

```
– private void printListOfStrings(List<?> list) {  
    for (Iterator<?> i = list.iterator(); i.hasNext(); ) {  
        System.out.println(i.next());  
    }  
}
```

# Creating a ArrayList the new way

- Specify, in angle brackets after the name, the type of object that the class will hold
- Examples:
  - `ArrayList<String> vec1 = new ArrayList<String>();`
  - `ArrayList<String> vec2 = new ArrayList<String>(10);`
- To get the old behavior, but without the warning messages, use the `<?>` wildcard
  - Example: `ArrayList<?> vec1 = new ArrayList<?>();`

# Accessing with and without generics

- Object get(int *index*)
  - Returns the component at position *index*
- Using get the old way:
  - ArrayList myList = new ArrayList();  
myList.add("Some string");  
String s = (String)myList.get(0);
- Using get the new way:
  - ArrayList<String> myList = new ArrayList<String>();  
myList.add("Some string");  
String s = myList.get(0);
- Notice that casting is no longer necessary when we retrieve an element from a “genericized” ArrayList

# Generics and Inheritance

- Suppose you want to restrict the type parameter to express some restriction on the type parameter
- This can be done with a notion of subtypes
- expressed in Java using inheritance
- So it's a natural combination to combine inheritance with generics
- A few examples follow

# Parameterized Classes in Methods

- A parameterized class is a type just like any other class.
- It can be used in method input types and return types.

# Parameterized Classes in Methods

- If a class is parameterized, that type parameter can be used for any type declaration in that class, e.g:

```
public class Box<E>
```

```
{E data;
```

```
public Box(E data) {this.data = data;}
```

```
public E getData() {return data;}
```

```
public void copyFrom(Box<E> b)
```

```
    {this.data = b.getData();}
```

# Bounded Parameterized Types

- Sometimes we want restricted parameterization of classes.
- We want a box, called MathBox that holds only Number objects.
- We can't use Box<E> because E could be anything.
- We want E to be a subclass of Number.

# Bounded Parameterized Types

```
public class MathBox<E extends Number> extends  
    Box<Number>  
  
{public MathBox(E data)  
{super(data);  
}  
  
public double sqrt()  
{return Math.sqrt(getData().doubleValue())  
}  
}
```

# Bounded Parameterized Types

- The <E extends Number> syntax means that the type parameter of MathBox must be a subclass of the Number class
  - We say that the type parameter is **bounded**

```
new MathBox<Integer>(5); //Legal
```

```
new MathBox<Double>(32.1); //Legal
```

```
new MathBox<String>("No good!"); //Illegal
```

# Bounded Parameterized Types

- Java allows multiple inheritance in the form of implementing multiple interfaces, so multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

```
<T extends A & B & C & ...>
```

- Example

```
interface A {...}
```

```
interface B {...}
```

```
class MultiBounds<T extends A & B> {
```

```
...
```

```
}
```

# Generics

## Examples

# What is Generics

- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

**Consider this code snippet**

```
List v = new ArrayList();
v.add(new String("test"));
Integer i = (Integer) v.get(0);
```

## Consider this code snippet

```
List v = new ArrayList();  
v.add(new String("test"));  
Integer i = (Integer)v.get(0); // Runtime error .  
Cannot cast from String to Integer
```

This error comes up only when we are executing the program and not during compile time.

# Use Generics to eliminate the Runtime error

# How does Generics help

The previous snippet with Generics is

```
List<String> v = new ArrayList<String>();  
v.add(new String("test"));  
Integer i = v.get(0); // Compile time error. Converting String to Integer
```

# Wildcards

- Wildcards help in allowing more than one type of class in the Collections
- We come across setting an upperbound and lowerbound for the Types which can be allowed in the collection
- The bounds are identified using a **? Operator** which means ‘an unknown type’

# Upperbound

- **List<? extends Number>** means that the given list contains objects of some unknown type which extends the Number class

Consider the snippet

```
List<Integer> ints = new ArrayList<Integer>();  
ints.add(2);
```

**What if we want a List which allows us to put all Number Class objects**

# Upperbound

```
List<? extends Number> nums = ints;  
nums.add(3.14);  
Integer x = ints.get(1);
```

# Example 1

- Give code to iterate (using an iterator) across a List **x** containing Strings
- All the Strings in **x** should be appended to a String named **answer**

# Example 1

```
String answer = "";
for (Iterator<String> i = x.iterator(); i.hasNext();) {
    answer += i.next();
}
```

# Raw Type is Unsafe: Use Generics to make the following code safe

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum between two objects */
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

Runtime Error:

```
Max.max("Welcome", 23);
```

# Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
    /** Return the maximum between two objects */
    public static <E extends Comparable<E>> E max(E o1, E o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

```
Max1.max("Welcome", 23);
```

# Generics with subclass

- Consider a method that takes some collection of **Shapes** as a parameter, and returns the sum of all the areas.
- **public double areaOfCollection  
(Collection<Shape> c)**  
**{ double sum = 0.0;**  
**for (Shape s : c)**  
**sum += s.getArea();**  
**}**

# Generics with subclass

- What if we want to use any subclass of Shape class?
- **Collection<Shape>** or **Collection<Circle>**
- ( Circle is a subclass of shape), Verify if the given method works if we call it with a Collection<Circle> object.

# Generics with subclass

```
public double areaOfCollection (Collection<? extends Shape> c)
{
    double sum = 0.0;
    for (Shape s : c)
        sum += s.getArea();
}
```

# Generics with Comparator

Comparator interface is also generic

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object o);  
}
```

Create a comparator CompareByLength to sort Strings by length in x

```
List<String> x = new ArrayList<String>();  
Collections.sort(x, new CompareByLength())
```

# Generics with Comparator

```
public class CompareByLength implements  
Comparator<String> {  
  
    int compare(String o1, String o2)  
    {return o1.length() - o2.length();  
  
}
```

# Generics with Comparator

- Method that takes an array of objects and a collection and puts all objects in the array into the collection

# Generics with Comparator

- Method that takes an array of objects and a collection and puts all objects in the array into the collection

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {  
    for (T o : a) {  
        c.add(o);  
    }  
}
```

# Generics with wildcards

- Does the following code compile? Make suitable modifications in the add method

```
1  public class WildCardDemo3 {  
2      public static void main(String[] args) {  
3          GenericStack<String> stack1 = new GenericStack<String>();  
4          GenericStack<Object> stack2 = new GenericStack<Object>();  
5          stack2.push("Java");  
6          stack2.push(2);  
7          stack1.push("Sun");  
8          add(stack1, stack2);  
9      }  
10  
11     public static<T> void add(GenericStack<T> s1, GenericStack<T> s2) {  
12         while (!s1.isEmpty()) {  
13             s2.push(s1.pop());  
14         }  
15     }  
16 }
```

# Generics with wildcards

- Modified add method

```
1 public class WildCardDemo3 {  
2     public static void main(String[] args) {  
3         GenericStack<String> stack1 = new GenericStack<String>();  
4         GenericStack<Object> stack2 = new GenericStack<Object>();  
5         stack2.push("Java");  
6         stack2.push(2);  
7         stack1.push("Sun");  
8         add(stack1, stack2);  
9     }  
10  
11     public static<T> void add(GenericStack<T> s1, GenericStack<? super T>  
s2) {  
12         while (!s1.isEmpty()) {  
13             s2.push(s1.pop());  
14         }  
15     }  
16 }
```

# Generics with multiple bounds

- The syntax for specification of type parameter bounds is:
- <TypeParameter extends  
Class & Interface<sub>1</sub> & ... & Interface<sub>N</sub> >
- A list of bounds consists of one class and/or several interfaces.
- Example
- class Pair<A extends Comparable<A> & Cloneable ,  
B extends Comparable<B> & Cloneable >  
implements Comparable<Pair<A,B>>, Cloneable  
{ ... }

This is a generic class with two type arguments A and B , both of which have two bounds.

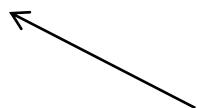
# Generics with Classes

```
public class Box<T> {  
  
    private T item;  
  
    public Box(T item) {  
        this.item=item;  
    }  
  
    public T get() {  
        return item;  
    }  
  
    public void set(T item) {  
        this.item=item;  
    }  
  
    public String toString() {  
        if(item!=null)  
            return ""+item;  
        else return "not set";  
    }  
}
```

Our box class is very basic

The type of Object is recorded as T (filled in when you declare a variable of type Box)

T is used to specify the type for item when declared as an instance datum, or passed as a parameter to a method or returned from a method



Note that T needs to have a `toString` implemented or this returns the address of item

```
public class BoxUsers {  
    public static void main(  
        String[] args) {  
        Box<String> a;  
        Box<Integer> b;  
        Box<Double> c;  
        Box<Object> d=null;  
        a=new Box<>"hi there");  
        b=new Box<>(100);  
        c=new Box<>(100.1);  
        System.out.println(a.get());  
        a.set("bye bye");  
        c.set(c.get()+1);  
        System.out.println(a);  
        System.out.println(c);  
        System.out.println(d);  
    }  
}
```

What if we want to do

```
c.set(b.get() + 1);
```

This yields an error because  
b.get() returns an Integer and  
c.set expects a Double, so  
instead use

```
c.set(new  
Double(b.get() + 1));
```

Output:

hi there

bye bye

101.1

null

# Multiple Generics

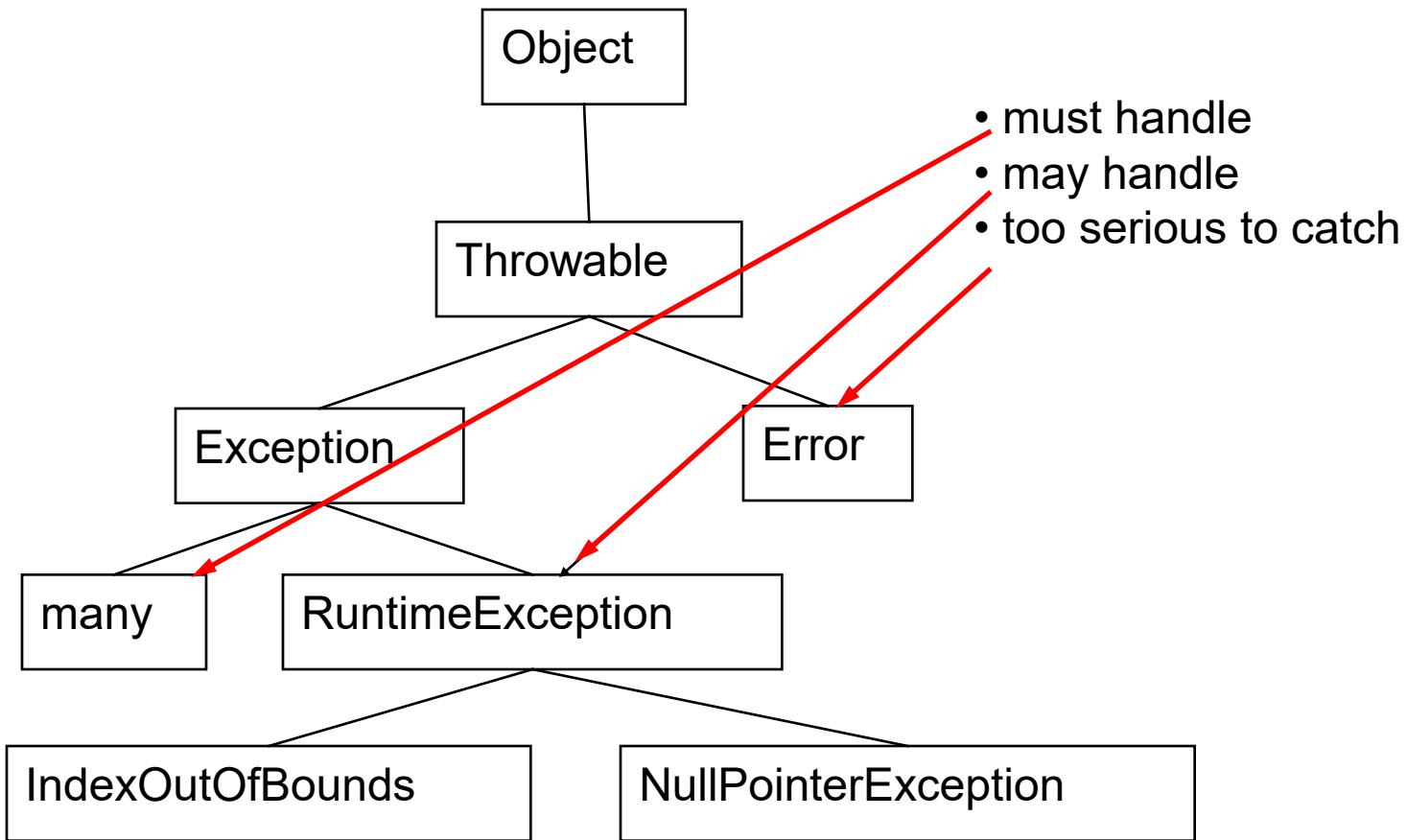
```
1 import java.util.ArrayList;
2 public class GenericStack<E, F> {
3     private ArrayList<E> list =
4         new ArrayList<E>();
5     public int getSize() {
6         return list.size();
7     }
8     public E peek() {
9         return list.get(getSize()-1);
10    }
11    public void push(E o) {
12        list.add(o);
13    }
14    public E pop() {
15        E o = list.get(getSize()-1);
16        list.remove(getSize()-1);
17        return o;
18    }
19    public boolean isEmpty() {
20        return list.isEmpty();
21    }
22    public void print(F f) {
23        System.out.println(f);
24    }
25    public static void main(String [] args)
26    {
27        GenericStack<String, Double> stack1 =
28            new GenericStack<String, Double>();
29        stack1.push("CSCI103");
30        stack1.push("CSCI104");
31        stack1.push("CSCI201");
32        stack1.print(3.5);
33        GenericStack<Integer, String> stack2
34        =
35            new GenericStack<Integer,
36            String>();
37        stack2.push(103);
38        stack2.push(104);
39        stack2.push(201);
40        stack2.print("Hello CSCI201");
41    }
42 }
```

# Java Exceptions

# Intro to Exceptions

- What are exceptions?
  - Events that occur during the execution of a program that interrupt the normal flow of control.
- One technique for handling Exceptions is to use return statements in method calls.
- This is fine, but java provides a much more general and flexible formalism that forces programmers to consider exceptional cases.
- e.g. Opening a file, does not exist.

# Exception Class hierarchy



## Throwable hierarchy

- All exceptions extend the class `Throwable`, which splits into two branches:

Error and Exception

- `Error`: internal errors and resource exhaustion inside the Java runtime system.  
Little you can do.
  - `Exception`: splits further into two branches.  
`RuntimeException` and other exceptions

# Focus on the Exception branch

- Two branches of **Exception**
  - exceptions that derived from **RuntimeException**
    - examples: a bad cast, an out-of-array access
    - happens because errors exist in your program. Your fault.
  - those not in the type of **RuntimeException**
    - example: trying to open a malformed URL
    - program is good, other bad things happen. Not your fault.

# Focus on the Exception branch

- Checked exceptions vs. unchecked exceptions
  - *Unchecked exceptions*: exceptions derived from the class **Error** or the class **RuntimeException**
  - *Checked exceptions*: all other exceptions that are not unchecked exceptions
    - If they occur, they must be dealt with in some way.
    - The compiler will check whether you provide exception handlers for checked exceptions which may occur

# Approaches to handling an exception

1. Prevent the exception from happening
2. Catch it in the method in which it occurs, and either
  - a. Fix up the problem and resume normal execution
  - b. Rethrow it
  - c. Throw a different exception
3. Declare that the method throws the exception
4. With 1. and 2.a. the caller never knows there was an error.
5. With 2.b., 2.c., and 3., if the caller does not handle the exception, the program will terminate and display a stack trace

# Example

```
public class myexception{  
    public static void main(String args[]){  
        try{  
            File f = new File("myfile");  
            FileInputStream fis = new FileInputStream(f);  
        }catch(FileNotFoundException ex){  
            File f = new File("Available File");  
            FileInputStream fis = new FileInputStream(f);  
        } finally{           // the finally block  
        } //continue processing here.  
    }  
}
```

- In this example we are trying to open a file and if the file does not exists we can do further processing in the catch block.
- The try and catch blocks are used to identify possible exception conditions. We try to execute any statement that might throw an exception and the catch block is used for any exceptions caused.
- If the try block does not throw any exceptions, then the catch block is not executed.
- The finally block is always executed irrespective of whether the exception is thrown or not.

# Using throws clause

Use the throws to handle the exception in the calling function.

```
public class myexception{  
    public static void main(String args[]){  
        try{  
            checkEx();  
        } catch(FileNotFoundException ex){  
        }  
    }  
  
    public void checkEx() throws FileNotFoundException{  
        File f = new File("myfile");  
        FileInputStream fis = new FileInputStream(f);  
        //continue processing here.  
    }  
}
```

# Using throws clause

- In this example, the main method calls the checkex() method and the checkex method tries to open a file, If the file is not available, then an exception is raised and passed to the main method, where it is handled.

# Catching Multiple exceptions

```
public class myexception{  
    public static void main(String args[]){  
        try{  
            File f = new File("myfile");  
            FileInputStream fis = new FileInputStream(f);  
        }  
        catch(FileNotFoundException ex){  
            File f = new File("Available File");  
            FileInputStream fis = new FileInputStream(f);  
        }catch(IOException ex){  
            //do something here  
        }  
    finally{  
        // the finally block  
    }  
    //continue processing here.  
    }  
}
```

# Catching Multiple exceptions

- We can have multiple catch blocks for a single try statement.
- The exception handler looks for a compatible match and then for an exact match.
- In other words, in the example, if the exception raised was myIException, a subclass of FileNotFoundException, then the catch block of FileNotFoundException is matched and executed.
- If a compatible match is found before an exact match, then the compatible match is preferred.
- We need to pay special attention on ordering of exceptions in the catch blocks, as it can lead to mismatching of exception and unreachable code.
- We need to arrange the exceptions from specific to general.

# Exception Handling Basics

- Three parts to Exception handling
  1. claiming exception
  2. throwing exception
  3. catching exception
- A method has the option of *throwing* one or more exceptions when specified conditions occur. This exception must be *claimed* by the method. Another method calling this method must either *catch* or *rethrow* the exception.  
(unless it is a RuntimeException)

# Claiming Exceptions

- Method declaration must specify every exception that the method potentially throws

MethodDeclaration throws Exception1,  
Exception2, ..., ExceptionN

- Exceptions themselves are concrete subclasses of Throwable and must be defined and locatable in regular way.

# Throwing Exception

- To throw an Exception, use the *throw* keyword followed by an instance of the Exception class

```
void foo() throws SomeException{  
    if (whatever) {...}  
    else{ throw new SomeException(...)}}
```

- Note that if a method foo has a throw clause within it, that the Exception that is thrown (or one of its superclasses) must be claimed after the signature.

# Catching Exceptions

- The third piece of the picture is catching exceptions.
- This is what you will do with most commonly, since many of java's library methods are defined to throw one or more runtime exception.
- Catching exceptions:
  - When a method is called that throws an Exception e.g SomeException, it must be called in a try-catch block:

```
try
{
    foo();
}
catch(SomeException se)
{
    ...
}
```

# Example1

```
import java.io.*;  
  
public class Exception1{  
    public static void main(String[] args){  
        InputStream f;  
        try{  
            f = new FileInputStream("foo.txt");  
        }  
        catch(FileNotFoundException fnfe){  
            System.out.println(fnfe.getMessage());  
        }  
    }  
}
```

# Example2

```
import java.io.*;
public class Exception2{
    public static void main(String[] args){
        InputStream fin;
        try{
            fin = new FileInputStream("foo.txt");
            int input = fin.read();
        }
        catch(FileNotFoundException fnfe){
            System.out.println(fnfe.getMessage());
        }
        catch(IOException ioe){
            System.out.println(ioe.getMessage());
        }
    }
}
```

# Catching exceptions

- Checked exceptions handling is strictly enforced. If you invoke a method that lists a checked exception in its throws clause, you have three choices
  1. Catch the exception and handle it
  2. Declare the exception in your own throws clause, and let the exception pass through your method (you may have a finally clause to clean up first)
  3. Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own throws clause

# try/catch clause (1)

- If no exception occurs during the execution of the statements in the `try` clause, it finishes successfully and all the `catch` clauses are skipped
- If any of the code inside the `try` block throws an exception, either directly via a `throw` or indirectly by a method invoked inside it
  1. The program skips the remainder of the code in the `try` block
  2. The `catch` clauses are examined one by one, to see whether the type of the thrown exception object is compatible with the type declared in the `catch`.
  3. If an appropriate `catch` clause is found, the code inside its body gets executed and all the remaining `catch` clauses are skipped.
  4. If no such a `catch` clause is found, then the exception is thrown into an outer `try` that might have a `catch` clause to handle it
- A `catch` clause with a superclass `exceptionType` cannot precede a `catch` clause with a subclass `exceptionType`

# finally clause

- You can use a finally clause without a catch clause
- Sometimes the finally clause can also throw an exception

## Example

```
public boolean searchFor(String file,  
String word)  
    throws StreamException  
{  
    Stream input = null;  
    try {  
        some code which may throw an  
StreamException  
    } finally {  
        input.close(); // this may throw an IOException  
    }  
}
```

# finally clause

- You may want to do some actions whether or not an exception is thrown.  
finally clause does this for you

```
Graphics g = image.getGraphics();  
try {  
    //1  
    code that might throw exceptions  
    //2  
} catch (IOException e) {  
    //3  
    show error dialog (// some code which may throw  
exceptions)  
    //4  
} finally {  
    g.dispose(); (// some code which will not throw  
exceptions)  
    //5  
} //6
```

- No exception is thrown: 1, 2, 5, 6
- An exception is thrown and caught by the catch clause
  - The catch clause doesn't throw any other exception: 1, 3, 4, 5, 6
  - The catch clause throws an exception itself: 1, 3, 5, and the exception is thrown back to the caller of this method
- An exception is thrown but not caught by the catch clause: 1, 5, and the exception is thrown back to the caller of this method

# Creating new exception types

- Exceptions are objects. New exception types should extend `Exception` or one of its subclasses
- Why creating new exception types?
  1. describe the exceptional condition in more details than just the string that `Exception` provides

E.g. suppose there is a method to update the current value of a named attribute of an object, but the object may not contain such an attribute currently. We want an exception to be thrown to indicate the occurring of if such a situation

```
public class NoSuchAttributeException extends Exception {  
    public String attrName;  
    public NoSuchAttributeException (String name) {  
        super("No attribute named \'" + name + "\' found");  
        attrName = name;  
    }  
}
```

2. the type of the exception is an important part of the exception data – programmers need to do some actions **exclusively** to one type of exception conditions, not others

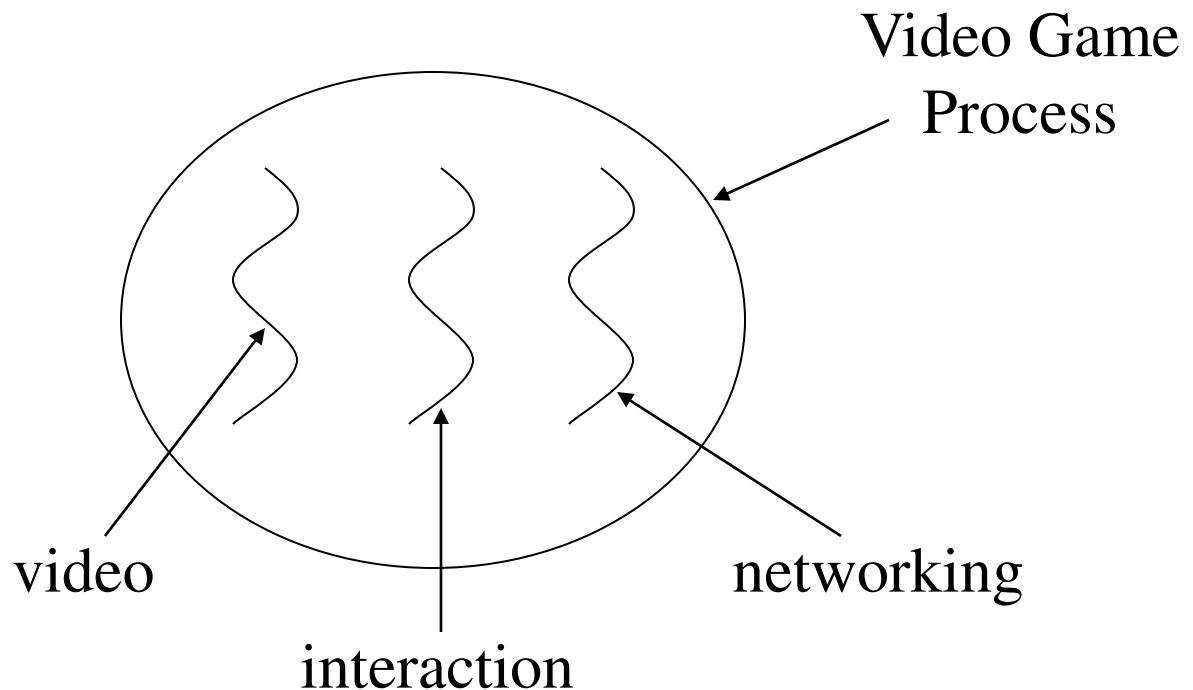
# Java Threads

Introduction to multithreading in Java

# What is a Thread?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking

# What is a Thread?



# Advantages

- easier to program
  - 1 thread per task
- can provide better performance
  - thread only runs when needed
  - no polling to decide what to do
- multiple threads can share resources
- utilize multiple processors if available

# Disadvantage

- multiple threads can lead to deadlock
  - much more on this later
- overhead of switching between threads

# Creating Threads (method 1)

- extending the **Thread** class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

# Creating Threads Example 1

```
class Output extends Thread {  
    private String toSay;  
    public Output(String st) {  
        toSay = st;  
    }  
    public void run() {  
        try {    for(;;){  
            System.out.println(toSay);  
            sleep(1000);  
        }  
    } catch(InterruptedException e) {  
        System.out.println(e);  
    }    }}
```

## Example 1 (continued)

```
class Program {  
    public static void main(String [] args) {  
        Output thr1 = new Output("Hello");  
        Output thr2 = new Output("There");  
        thr1.start();  
        thr2.start();  
    }  
}
```

- main thread is just another thread (happens to start first)
- main thread can end before the others do
- any thread can spawn more threads

# Creating Threads (method 2)

- implementing **Runnable** interface
  - virtually identical to extending Thread class
  - must still define the *run()* method
  - setting up the threads is slightly different

## Creating Threads Example 2

```
class Output implements Runnable {  
    private String toSay;  
    public Output(String st) {  
        toSay = st;  
    }  
    public void run() {  
        try {  
            for(;;) {  
                System.out.println(toSay);  
                Thread.sleep(1000);  
            }  
        } catch(InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
}
```

## Example 2 (continued)

```
class Program {  
    public static void main(String [] args) {  
        Output out1 = new Output("Hello");  
        Output out2 = new Output("There");  
        Thread thr1 = new Thread(out1);  
        Thread thr2 = new Thread(out2);  
        thr1.start();  
        thr2.start();  
    }  
}
```

- main is a bit more complex
- everything else identical for the most part

# Advantage of Using Runnable

- remember - can only extend one class
- implementing runnable allows class to extend something else

# Controlling Java Threads

- `_.start()`: begins a thread running
- `_.stop()`: kills a specific thread (deprecated)
- `_.join()`: wait for specific thread to finish

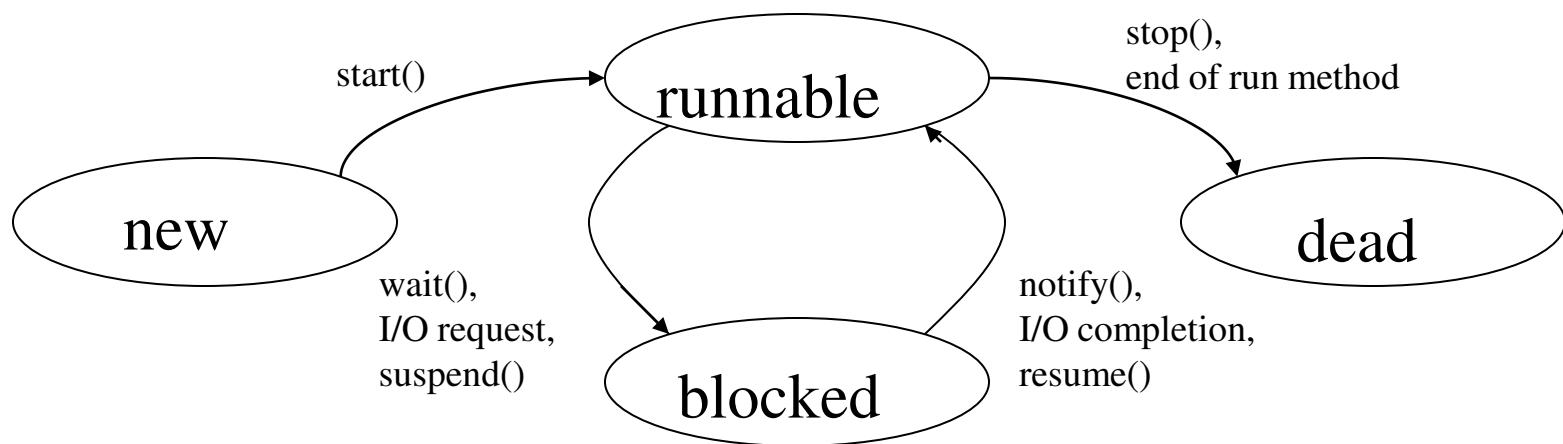
# Java Thread Scheduling

- highest priority thread runs
  - if more than one, arbitrary
- *yield()*: current thread gives up processor so another of equal priority can run
  - if none of equal priority, it runs again
- *sleep(msec)*: stop executing for set time
  - lower priority thread can run

# States of Java Threads

- 4 separate states
  - new: just created but not started
  - runnable: created, started, and able to run
  - blocked: created and started but unable to run because it is waiting for some event to occur
  - dead: thread has finished or been stopped

# States of Java Threads



# Java Thread Example 1

```
class Job implements Runnable {  
    private static Thread [] jobs = new Thread[4];  
    private int threadID;  
    public Job(int ID) {  
        threadID = ID;  
    }  
    public void run() { do something }  
    public static void main(String [] args) {  
        for(int i=0; i<jobs.length; i++) {  
            jobs[i] = new Thread(new Job(i));  
            jobs[i].start();  
        }  
        try {  
            for(int i=0; i<jobs.length; i++) {  
                jobs[i].join();  
            }  
        } catch(InterruptedException e) { System.out.println(e); }  
    }  
}
```

# Events: Mouse, Keyboard

# Event-Driven Programming

- *Procedural programming* is executed in procedural order.
- In *event-driven programming*, code is executed upon activation of events.

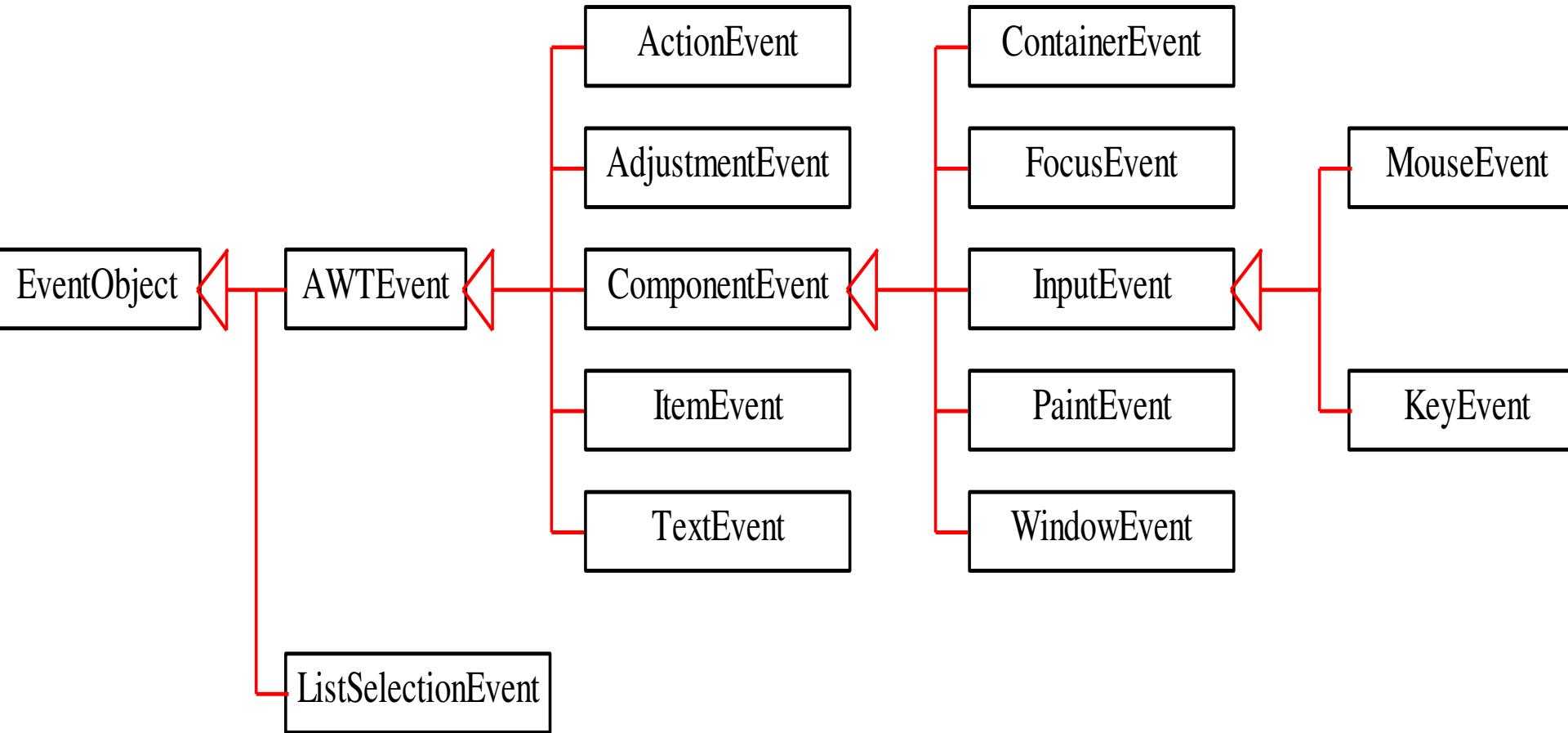
# Events

- An *event* can be defined as a type of signal to the program that something has happened.
- The event is generated by external user actions such as mouse movements, mouse button clicks, and keystrokes, or by the operating system, such as a timer.

# Event Information

- `id`: A number that identifies the event.
- `target`: The source component upon which the event occurred.
- `arg`: Additional information about the source components.
- `x, y coordinates`: The mouse pointer location when a mouse movement event occurred.
- `clickCount`: The number of consecutive clicks for the mouse events. For other events, it is zero.
- `when`: The time stamp of the event.
- `key`: The key that was pressed or released.

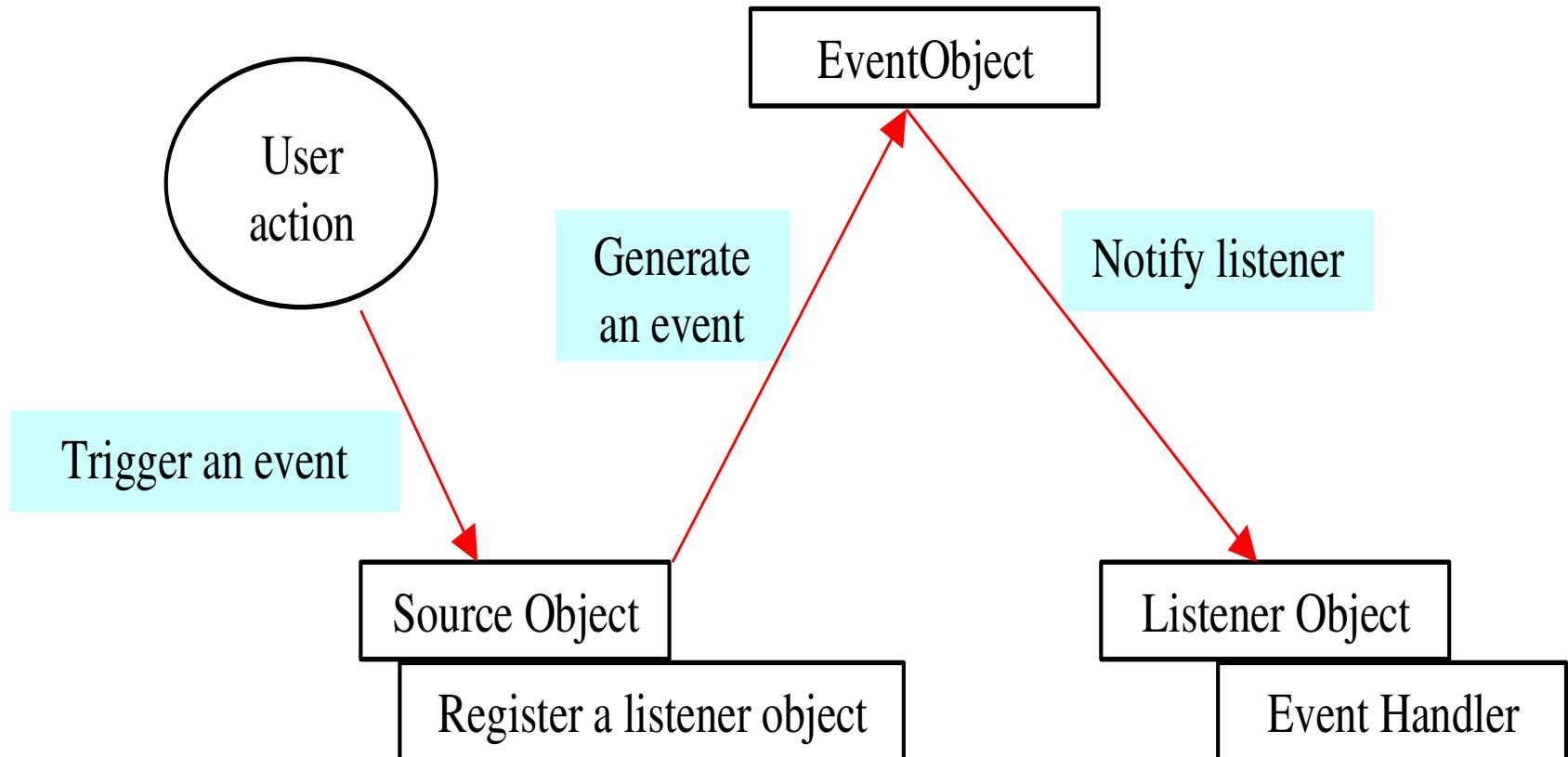
# Event Classes



# Selected User Actions

| User Action                                        | Source Object  | Event Type Generated |
|----------------------------------------------------|----------------|----------------------|
| Clicked on a button                                | JButton        | ActionEvent          |
| Changed text                                       | JTextComponent | TextEvent            |
| Double-clicked on a list item                      | JList          | ActionEvent          |
| Selected or deselected an item with a single click | JList          | ItemEvent            |
| Selected or deselected an item                     | JComboBox      | ItemEvent            |

# The Delegation Model



# Selected Event Handlers

| <b>Event Class</b> | <b>Listener Interface</b> | <b>Listener Methods (Handlers)</b>                                                                                                                                                                                       |
|--------------------|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ActionEvent        | ActionListener            | actionPerformed(ActionEvent)                                                                                                                                                                                             |
| ItemEvent          | ItemListener              | itemStateChanged(ItemEvent)                                                                                                                                                                                              |
| WindowEvent        | WindowListener            | windowClosing(WindowEvent)<br>windowOpened(WindowEvent)<br>windowIconified(WindowEvent)<br>windowDeiconified(WindowEvent)<br>windowClosed(WindowEvent)<br>windowActivated(WindowEvent)<br>windowDeactivated(WindowEvent) |
| ContainerEvent     | ContainerListener         | componentAdded(ContainerEvent)<br>componentRemoved(ContainerEvent)                                                                                                                                                       |

# Listeners

- Listener methods take a corresponding type of event as an argument.
- Event objects have useful methods. For example, `getSource` returns the object that produced this event.
- A `MouseEvent` has methods `getX`, `getY`.

# Mouse Events

- Mouse events are captured by an object which is a `MouseListener` and possibly a `MouseMotionListener`.
- A mouse listener is usually attached to a `JPanel` component.
- It is not uncommon for a panel to serve as its own mouse listener:

```
addMouseListener(this);  
addMouseMotionListener(this); // optional
```

# Mouse Events (cont'd)

- Mouse listener methods receive a `MouseEvent` object as a parameter.
- A mouse event can provide the coordinates of the event and other information:

```
public void mousePressed(MouseEvent e)
{
    int x = e.getX();
    int y = e.getY();
    int clicks = e.getClickCount();
}
```

# Mouse Events (cont'd)

- The `MouseListener` interface defines five methods:

`void mousePressed (MouseEvent e)`

`void mouseReleased (MouseEvent e)`

`void mouseClicked (MouseEvent e)`

`void mouseEntered (MouseEvent e)`

`void mouseExited (MouseEvent e)`

Called when  
the mouse  
cursor  
enters/exits  
component's  
visible area

- One click and release causes several calls. Using only `mouseReleased` is usually a safe bet.

# Mouse Events (cont'd)

- The `MouseMotionListener` interface adds two methods:

```
void mouseMoved (MouseEvent e)
```

```
void mouseDragged (MouseEvent e)
```

Called when the mouse has moved with a button held down

- These methods are usually used together with `MouseListener` methods (i.e., a class implements both interfaces).

# Event Handling Strategies: Pros and Cons

- Separate Listener
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - Separate class easier to manage
  - Disadvantage
    - Need extra step to call methods in main window
- Main window that implements interface
  - Advantage
    - No extra steps needed to call methods in main window
  - Disadvantage
    - Must implement methods you might not care about

# Event Handling Strategies: Pros and Cons, cont.

- Named inner class
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - No extra steps needed to call methods in main window
  - Disadvantage
    - A bit harder to understand
- Anonymous inner class
  - Advantages
    - Same as named inner classes
    - Even shorter
  - Disadvantage
    - Much harder to understand

# Event Listeners

# Event Listeners

- Listener objects is added to Button
- When the user clicks the button,
- Button object generates an ActionEvent object.
- This calls the listener object's **method** and passes the ActionEvent object generated.

# How to Attach an Event Listener to an Event Source

- o o is an event source
- h h is an event listener of type XXX

**o.addXXX(h)**

where XXX is one of the following:

ActionListener

ComponentListener

MouseListener

FocusListener

MouseMotionListener

TextListener

KeyListener

AdjustmentListener

WindowListener

ItemListener

# Registering Event Listeners

- To register a listener object with a source object, you use lines of code that follow the model

```
source.addEventListener(eventListenerObject  
);
```

# The ActionListener Interface

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

# version 1

```
JButton hw = new JButton("Hello World!");
panel.add(hw);
```

```
hw.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        System.exit(0);    }  });
```

# version 2

```
class MyFrame extends JFrame implements ActionListener
{ public MyFrame(){
    JButton hw = new JButton("Hello World!");
    panel.add(hw);
    hw.addActionListener(this);

}

public void actionPerformed(ActionEvent o){
    System.exit(0);
}
```

# version 3

```
class MyFrame extends Jframe {  
    Button hw;  
{ public MyFrame(){  
    JButton hw = new JButton("Hello World!");  
    panel.add(hw);  
    hw.addActionListener(new MyActionListener()); }  
}  
}
```

```
class MyActionListener implements ActionListener {  
    public void actionPerformed(ActionEvent o){  
        System.exit(0);  
    }  
}
```

# Implementing an Event Handler

- Implement a listener interface or extend a class that implements a listener interface.
- Register an instance of the event handler class as a listener upon one or more components.
- Implement the methods in the listener interface to handle the event.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class ButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Got a button press:" + e);
    }
}

public class Main {
    private static void showGUI() {
        JFrame frame = new JFrame("Swing GUI");
        java.awt.Container content = frame.getContentPane();
        content.setLayout(new FlowLayout());
        content.add(new JLabel ("Yo!"));
        JButton button = new JButton ("Click Me");
        button.addActionListener(new
ButtonListener());
        content.add(button);
        frame.pack();
        frame.setVisible(true);
    }
}
```

```
class ButtonListener implements ActionListener {  
    public void actionPerformed (ActionEvent e) {  
        if (e.getActionCommand().equals ("On")){  
            System.out.println("On!");  
        } else if (e.getActionCommand().equals("Off")) {  
            System.out.println("Off!");  
        } else {  
            System.out.println("Unrecognized button press!"); } } }
```

```
public class main {  
    private static void showGUI() {
```

```
    ...
```

```
        ButtonListener bl = new ButtonListener();  
        JButton onButton = new JButton ("On");  
        onButton.addActionListener(bl);  
        content.add(onButton);  
        JButton offButton = new JButton ("Off");  
        offButton.addActionListener(bl);  
        content.add(offButton);
```



# Keyboard Events

- Keyboard events are captured by an object which is a `KeyListener`.
- A key listener object must first obtain keyboard “focus.” This is done by calling the component’s `requestFocus` method.
- If keys are used for moving objects (as in a drawing program), the “canvas” panel may serve as its own key listener:

```
addKeyListener(this);
```

# Keyboard Events (cont'd)

- The `KeyListener` interface defines three methods:

`void keyPressed (KeyEvent e)`

`void keyReleased (KeyEvent e)`

`void keyTyped (KeyEvent e)`

- One key pressed and released causes several calls.

# Keyboard Events (cont'd)

- Use `keyTyped` to capture character keys (i.e., keys that correspond to printable characters).
- `e.getKeyChar()` returns a `char`, the typed character:

```
public void keyTyped (KeyEvent e)
{
    char ch = e.getKeyChar();
    if (ch == 'A')
        ...
}
```

# Keyboard Events (cont'd)

- Use `keyPressed` or `keyReleased` to handle “action” keys, such as cursor keys, `<Enter>`, function keys, and so on.
- `e.getKeyCode()` returns an `int`, the key’s “virtual code.”
- The `KeyEvent` class defines constants for numerous virtual keys. For example:

|                                                   |             |
|---------------------------------------------------|-------------|
| <code>VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN</code>    | Cursor keys |
| <code>VK_HOME, VK_END, VK_PAGE_UP, ...etc.</code> | Home, etc.  |

# Keyboard Events (cont'd)

- `e.isShiftDown()`, `e.isControlDown()`,  
`e.isAltDown()` return the status of the  
respective modifier keys.
- `e.getModifiers()` returns a bit pattern that  
represents the status of all modifier keys.
- `KeyEvent` defines “mask” constants  
`CTRL_MASK`, `ALT_MASK`,  
`SHIFT_MASK`, and so on.

# Standard AWT Event Listeners (Summary)

Why no adapter class?

| Listener            | Adapter Class<br>(If Any) | Registration Method    |
|---------------------|---------------------------|------------------------|
| ActionListener      |                           | addActionListener      |
| AdjustmentListener  |                           | addAdjustmentListener  |
| ComponentListener   | ComponentAdapter          | addComponentListener   |
| ContainerListener   | ContainerAdapter          | addContainerListener   |
| FocusListener       | FocusAdapter              | addFocusListener       |
| ItemListener        |                           | addItemListener        |
| KeyListener         | KeyAdapter                | addKeyListener         |
| MouseListener       | MouseAdapter              | addMouseListener       |
| MouseMotionListener | MouseMotionAdapter        | addMouseMotionListener |
| TextListener        |                           | addTextListener        |
| WindowListener      | WindowAdapter             | addWindowListener      |

# Standard AWT Event Listeners (Details)

- **ActionListener**
  - Handles buttons and a few other actions
    - actionPerformed(ActionEvent event)
- **AdjustmentListener**
  - Applies to scrolling
    - adjustmentValueChanged(AdjustmentEvent event)
- **ComponentListener**
  - Handles moving/resizing/hiding GUI objects
    - componentResized(ComponentEvent event)
    - componentMoved (ComponentEvent event)
    - componentShown(ComponentEvent event)
    - componentHidden(ComponentEvent event)

# Standard AWT Event Listeners (Details Continued)

- **ContainerListener**
  - Triggered when window adds/removes GUI controls
    - componentAdded(ContainerEvent event)
    - componentRemoved(ContainerEvent event)
- **FocusListener**
  - Detects when controls get/lose keyboard focus
    - focusGained(FocusEvent event)
    - focusLost(FocusEvent event)

# Standard AWT Event Listeners (Details Continued)

- **ItemListener**
  - Handles selections in lists, checkboxes, etc.
    - `itemStateChanged(ItemEvent event)`
- **KeyListener**
  - Detects keyboard events
    - `keyPressed(KeyEvent event)` -- any key pressed down
    - `keyReleased(KeyEvent event)` -- any key released
    - `keyTyped(KeyEvent event)` -- key for printable char released

# Standard AWT Event Listeners (Details Continued)

- **MouseListener**
  - Applies to basic mouse events
    - `mouseEntered(MouseEvent event)`
    - `mouseExited(MouseEvent event)`
    - `mousePressed(MouseEvent event)`
    - `mouseReleased(MouseEvent event)`
    - `mouseClicked(MouseEvent event)` -- Release without drag
      - Applies on release if no movement since press
- **MouseMotionListener**
  - Handles mouse movement
    - `mouseMoved(MouseEvent event)`
    - `mouseDragged(MouseEvent event)`