BIRZEIT UNIVERSITY

COMP231

Advanced Programming

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Welcome to COMP231, one of the most

interesting

programming courses offered at Computer Science Department

Course Description

In this course, you will learn some of the concepts, fundamental syntax, and thought processes behind true **O**bject-**O**riented **P**rogramming (**OOP**)

Course Description

- Upon completion of this course, you'll be able to:
  - Demonstrate understanding of classes, constructors, objects, and instantiation.
  - Access variables and modifier keywords.
  - Develop methods using parameters and return values.
  - Build control structures in an object-oriented environment.
  - Convert data types using API methods and objects.
  - Design object-oriented programs using scope, inheritance, and other design techniques.
  - Create an object-oriented application using Java packages, APIs. and interfaces, in conjunction with classes and objects.

Logistics

- Instructor: **Mamoun Nawahdah** (Masri318)
- Text book:
  - Introduction To JAVA Programming, **10th** edition.
  - Author: Y. Daniel Liang.
  - Publisher: Prentice Hall.
- Lab Manual:
  - **Title:** LABORATORY WORK BOOK (**COMP231**)

Grading Criteria

Midterm exam	30%
4 Assignments	10%
4 Quizzes	15%
Final Practical Exam	10%
Final exam	35%

Special Regulations

- **Assignments**:
  - All assignments are **individual** efforts any duplicated copies will be treated as a cheating attempt which lead to **ZERO** mark.
  - Using code from the internet will be treated as cheating as well.
  - The assignments should be **submitted through Ritaj** within the specified deadline.
  - No late submissions are accepted even by **1 minute** after the deadline.

Special Class Regulations

- **Attendance** is mandatory. University regulations will be **strictly** enforced.
- **Mobile**: Keep it off during the class. If your mobile ring you have to leave the classroom quickly, quietly and don't come back.
- **Late**: you are expected to be in the classroom before the teacher arrival. After **5** minutes you will not allowed entering the classroom.

Course Outline

Topics	Chapter 10th Edition	Chapter 9th Edition	# of lectures
Introduction to Java	1-8	1-7	5
Objects and Classes	9	8	3
Strings	4.4, 10.10, 10.11	9, 14	2
Thinking in Objects	10	10	2
Inheritance and Polymorphism	11	11	3
Midterm Exam (30%)			
Abstract Classes and Interfaces	13	15	3
Exception Handling and Text I/O	12	14	3
JavaFX Basics	14	External Material	3
Event-Driven Programming	15	External Material	3
JavaFX UI Controls	16	External Material	3
Final Exam (35%)			

Lab Outline

Lab #	Title	Quizzes
1	Program structure in Java	
2	Structure Programming - Revision	
3	Methods	
4	Arrays and Object Use	Q1
5	Object-Oriented Programming	
6-7	String	Q2
8	Inheritance and Polymorphism	
9-10	Abstract classes and Interfaces and Text I/O	Q3
11	GUI	
12	Event-Driven Programming	Q4
Practical Final Exam (10%)		

Why Java?

- Java is a general purpose programming language.
- Java is the Internet programming language.

11

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

12

JDK Versions

- JDK 1.02 (1995)
- JDK 1.1 (1996)
- JDK 1.2 (1998)
- JDK 1.3 (2000)
- JDK 1.4 (2002)
- JDK 1.5 (2004) a. k. a. JDK 5 or Java 5
- JDK 1.6 (2006) a. k. a. JDK 6 or Java 6
- JDK 1.7 (2011) a. k. a. JDK 7 or Java 7
- **JDK 8 (April 15, 2014)**

13

JDK Editions

- **Java Standard Edition (J2SE)**
  - J2SE can be used to develop client-side standalone applications or applets.
- **Java Enterprise Edition (J2EE)**
  - J2EE can be used to develop server-side applications such as Java servlets, Java ServerPages, and Java ServerFaces.
- **Java Micro Edition (J2ME).**
  - J2ME can be used to develop applications for mobile devices such as cell phones.

14

Popular Java IDEs

IDE ➔ **I**ntegrated **D**evelopment **E**nvironment

NetBeans

bluej.org

15

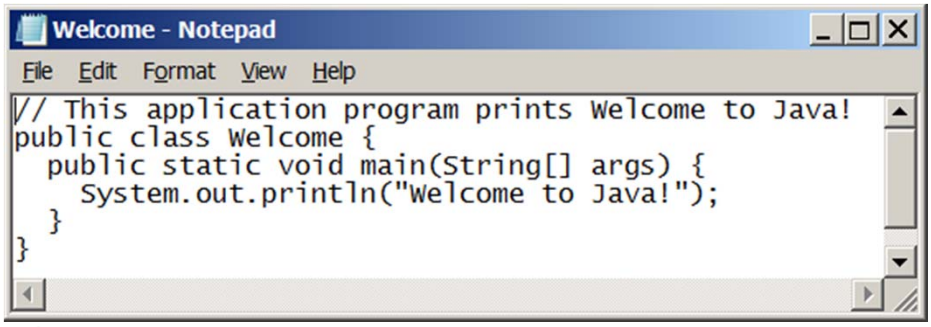
A Simple Java Program

```
// This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

16

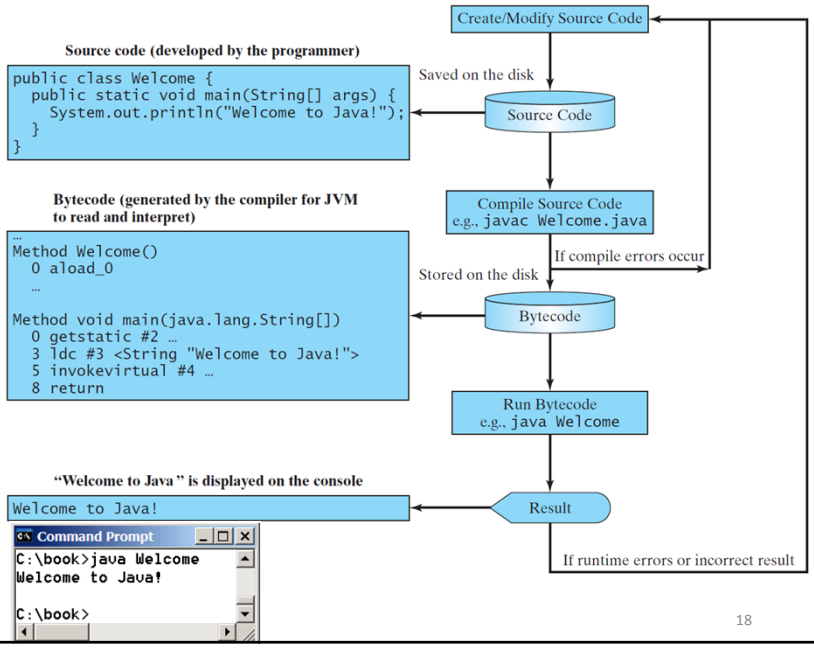
Creating and Editing Using NotePad

To use NotePad, type:
notepad Welcome.java
from the **DOS** prompt.



17

Creating, Compiling, and Running Programs



18

Compiling and Running Java from the Command Window

- Set path to JDK **bin** directory

  set path=c:\Program Files\java\jdk1.8.0\bin

- Set **classpath** to include the current directory

  set classpath=.

- Compile:

  javac Welcome.java

- Run:

  java Welcome

19

Anatomy of a Java Program

- Class name
- Main method
- Statements
- Statement terminator
- Reserved words
- Comments
- Blocks

20

Class Name

- Every Java program must have **at least** one class.
- Each class has a name.
- By **convention**, class names start with an uppercase letter.
- In this example, the class name is **Welcome**.

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

21

Main Method

- In order **to run a class**, the class must contain a method named **main**.
- The program is executed from the **main** method.

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

22

Statement

❖ A statement represents an action or a sequence of actions.

❖ The statement **System.out.println("Welcome to Java!")** in the program is a statement to display the greeting "*Welcome to Java*!".

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

23

Statement Terminator

❖ **Every** statement in Java ends with a semicolon

;

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

24

Reserved Words

❖ Reserved words or **keywords** are words that have a specific meaning to the compiler and cannot be used for other purposes in the program.

❖ For example, when the compiler sees the word **class**, it understands that the word after class is the name for the class.

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

25

Programming Style and Documentation

❖ Appropriate **Comments**.

❖ Naming **Conventions**.

❖ Proper **Indentation** and Spacing Lines.

❖ Block Styles.

26

Naming Conventions

- Choose **meaningful** and descriptive names.
- Class names:
  - Capitalize the **F**irst **L**etter of each word in the name. For example, the class name **ComputeExpression**.

27

Proper Indentation and Spacing

- **Indentation**
  - Indent **two** spaces.
- **Spacing**
  - Use blank line to separate segments of the code.

28

Block Styles

Next-line style

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

End-of-line style

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

29

Programming Errors

- **Syntax Errors**
  - Detected by the compiler
- **Runtime Errors**
  - Causes the program to abort
- **Logic Errors**
  - Produces incorrect result

30

BIRZEIT UNIVERSITY

Elementary Programming

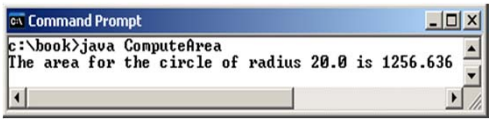
By: Mamoun Nawahdah (PhD)
2015/2016

Trace a Program Execution

```
public class ComputeArea {
  /** Main method */
  public static void main(String[] args) {
    double radius;
    double area;

    // Assign a radius
    radius = 20;

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

memory

radius	20
area	1256.636



2

Identifiers

- An **identifier** is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
  - An identifier cannot be a reserved word. (See **Appendix A**, "Java Keywords").
- An identifier cannot be **true**, **false**, or **null**.
- An identifier can be of any length.

3

Declaring Variables

```
int x;            // Declare x to be an integer variable
double radius;    // Declare radius to be a double variable
char a;           // Declare a to be a character variable
```

Assignment Statements

```
x = 1;            // Assign 1 to x
radius = 1.0;     // Assign 1.0 to radius
a = 'A';          // Assign 'A' to a
```

4

Declaring and Initializing in One Step

```
int x = 1;
double d = 1.4;
```

Named Constants

```
final datatype CONSTANTNAME = VALUE;
final double PI = 3.14159;
final int SIZE = 3;
```

5

Naming Conventions

- Choose **meaningful** and descriptive names.
- Variables and method names:
  - Use lowercase.
  - If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name.
  - For example, the variables **radius** and **area**, and the method **computeArea**.

6

Naming Conventions, cont.

- **Class names:**
  - Capitalize the first letter of each word in the name.
  - For example, the class name **ComputeArea**.

- **Constants**:
  - Capitalize all letters in constants, and use underscores to connect words.
  - For example, the constant **PI** and **MAX_VALUE**

7

Numerical Data Types

Name	Range	Storage Size
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

8

double vs. float

The double type values are more accurate than the float type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`

7 digits

Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

Numeric Type Conversion

Consider the following statements:

```
byte i = 100;

long k = i * 3 + 4;

double d = i * 3.1 + k / 2;
```

Conversion Rules

- When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is **double**, the other is converted into double.
2. Otherwise, if one of the operands is **float**, the other is converted into float.
3. Otherwise, if one of the operands is **long**, the other is converted into long.
4. Otherwise, both operands are converted into **int**.

Type Casting

Implicit casting

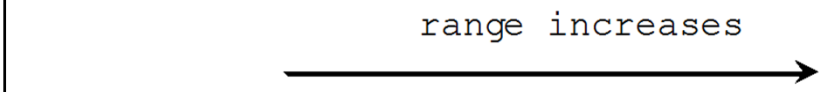
double d = 3; (type widening)

Explicit casting

int i = (int)3.0; (type narrowing)

int i = (int)3.9; (Fraction part is truncated)

What is wrong? int x = 6 / 2.0;



byte, short, int, long, float, double

13

Character Data Type

char letter = 'A'; (ASCII)

char numChar = '4'; (ASCII)

char letter = '\u0041'; (Unicode)

char numChar = '\u0034'; (Unicode)

NOTE: The increment and decrement operators can also be used on **char** variables to get the next or preceding Unicode character. For example, the following statements display character **b**.

```
char ch = 'a';
System.out.println(++ch);
```

14

The String Type

❖ The char type only represents **one** character. To represent a string of characters, use the data type called **String**. For example:

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

❖ String is actually a predefined class in the Java library.

❖ The String type is not a primitive type. It is known as a ***reference** type*.

String Concatenation

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

Console Input

- You can use the **Scanner** class for console input.
- Java uses **System.in** to refer to the standard input device (i.e. Keyboard).

```
import  java.util.Scanner;
public class Test{
   public static void main(String[] s){
      Scanner input = new Scanner(System.in);
      System.out.println("Enter  X : ");
       int x = input.nextInt();
       System.out.println("You entered: "+ x);
   }
}
```

17

Reading Numbers from the Keyboard

Scanner input = new Scanner(System.in);
int value = input.nextInt();

Method	Description
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

18

BIRZEIT UNIVERSITY

Selections

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Comparison Operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	less than	radius < 0	false
<=	≤	less than or equal to	radius <= 0	false
>	>	greater than	radius > 0	true
>=	≥	greater than or equal to	radius >= 0	true
==	=	equal to	radius == 0	false
!=	≠	not equal to	radius != 0	true

if-else

```
if (radius >= 0) {
   area = radius * radius * 3.14159;
   System.out.println("The area for the " +
   "circle of radius " + radius +  " is " + area);
}
else {
   System.out.println("Negative input");
}
```

3

Common Errors

❖ Adding a **semicolon** at the end of an **if** clause is a common mistake.

```
if (radius >= 0) ;
{
  area = radius*radius*PI;
  System.out.println( "The area for the circle is " + area);
}
```

Wrong

❖ This mistake is hard to find, because it is not a compilation error or a runtime error, it is a **logic** error.

❖ This error often occurs when you use the next-line block style.

4

Logical Operators

Operator	Name
!	not
&&	and
||	or
^	exclusive or

5

switch Statements

```
switch (status) {
      case 0:  compute taxes for single filers;
               break;
      case 1:  compute taxes for married file jointly;
               break;
      case 2:  compute taxes for married file separately;
               break;
      case 3:  compute taxes for head of household;
               break;
      default: System.out.println("Errors: invalid status");
               System.exit(1);
}
```

6

Problem: Chinese Zodiac

Write a program that prompts the user to enter a year and displays the animal for the year.

year % 12 =
0: monkey
1: rooster
2: dog
3: pig
4: rat
5: ox
6: tiger
7: rabbit
8: dragon
9: snake
10: horse
11: sheep

7

Conditional Operator

```
if (x > 0)
   y = 1;
else
   y = -1;
```

❖ is equivalent to:

```
y = (x > 0) ? 1 : -1;
```

(boolean-expression) **?** expression1 **:** expression2

8

Conditional Operator

```
if (num % 2 == 0)
    System.out.println(num + "is even");
else
    System.out.println(num + "is odd");
```

```
System.out.println(  (num % 2 == 0)  ?
      num + "is even" :   num + "is odd");
```

9

Formatting Output

❖ Use the **printf** statement:

```
System.out.printf( format, items );
```

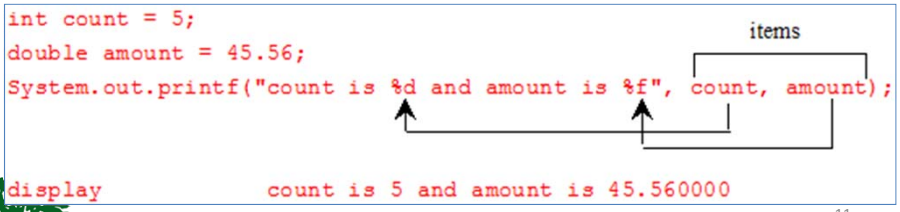
- Where format is a string that may consist of substrings and **format specifiers**.
- A format specifier specifies how an item should be displayed.
- An item may be a numeric value, character, boolean value, or a string.
- Each specifier begins with a **percent** sign.

10

Frequently-Used Specifiers

Specifier	Output	Example
%b	a boolean value	true or false
%c	a character	'a'
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	"Java is cool"

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);

display          count is 5 and amount is 45.560000
```



11

Operator Precedence

- var++, var--
- +, - (Unary plus and minus), ++var,--var
- (type) Casting
- ! (Not)
- *, /, % (Multiplication, division, and remainder)
- +, - (Binary addition and subtraction)
- <, <=, >, >= (Comparison)
- ==, !=; (Equality)
- ^ (Exclusive OR)
- && (Conditional AND) Short-circuit AND
- || (Conditional OR) Short-circuit OR
- =, +=, -=, *=, /=, %= (Assignment operator)

12

Operator Precedence and Associativity

❖ The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.)

❖ When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the **associativity rule**.

❖ If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are **left-associative**.

13

Operator Associativity

❖ When two operators with the same precedence are evaluated, the *associativity* of the operators determines the order of evaluation.

❖ All binary operators except assignment operators are ***left-associative***.

a – b + c – d is equivalent to **((a – b) + c) – d**

❖ Assignment operators are ***right-associative***. Therefore, the expression

a = b += c = 5 is equivalent to **a = (b += (c = 5))**

14

BIRZEIT UNIVERSITY

Loops

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Opening Problem

Problem:

100 times

```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");

...
...
...
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
```

Introducing while Loops

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java");
  count++;
}
```

do-while Loop

```
do {
   // Loop body;
   Statement(s);
} while (loop-continuation-condition);
```

for Loops

```
for ( initial-action ;
      loop-continuation-condition ;
      action-after-each-iteration ) {
        // loop body;
        Statement(s);
}
```

```
for (int i = 0 ; i < 100 ; i++) {
    System.out.println( "Welcome to Java!");
}
```

5

Note

❖ The **initial-action** in a for loop can be a list of zero or more comma-separated expressions.

❖ The **action-after-each-iteration** in a for loop can be a list of zero or more comma-separated statements.

❖ Therefore, the following two for loops are correct:

```
for ( int i = 1 ;  i < 100 ;  System.out.println(i++)) ;
```

```
for (  int i = 0 , j = 0 ;  (i + j < 10) ;  i++, j++  ) {
        // Do something
}
```

6

Note

❖ If the **loop-continuation-condition** in a for loop is omitted, it is implicitly **true**.

❖ Thus the statement given below in (a), which is an **infinite loop**, is correct.

```
for ( ; ; ) {
  // Do something
}
```

Equivalent

```
while (true) {
  // Do something
}
```

7

Caution

❖ Adding a **semicolon** at the end of the for clause before the loop body is a common mistake, as shown below:

Logic Error

```
for (int i=0 ; i<10 ; i++) ;
{
   System.out.println("i is " + i);
}
```

8

Caution

- Similarly, the following loop is also wrong:

```
int i=0;
while (i < 10);
{
    System.out.println("i is " + i);
    i++;
}
```

Logic Error

- In the case of the do loop, the following semicolon is needed to end the loop:

```
int i=0;
do {
    System.out.println("i is " + i);
    i++;
} while (i<10);
```

Correct

9

break

```
public class TestBreak {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      sum += number;
      if (sum >= 100)
        break;
    }

    System.out.println("The number is " + number);
    System.out.println("The sum is " + sum);
  }
}
```

10

continue

```
public class TestContinue {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      if (number == 10 || number == 11)
        continue;
      sum += number;
    }

    System.out.println("The sum is " + sum);
  }
}
```

Problem: Displaying Prime Numbers

Problem: Write a program that displays the first 50 prime numbers in five lines, each of which contains 10 numbers. An integer greater than 1 is *prime* if its only positive divisor is 1 or itself. For example, 2, 3, 5, and 7 are prime numbers, but 4, 6, 8, and 9 are not.

Solution: The problem can be broken into the following tasks:

- For number = 2, 3, 4, 5, 6, ..., test whether the number is prime.
- Determine whether a given number is prime.
- Count the prime numbers.
- Print each prime number, and print 10 numbers per line.

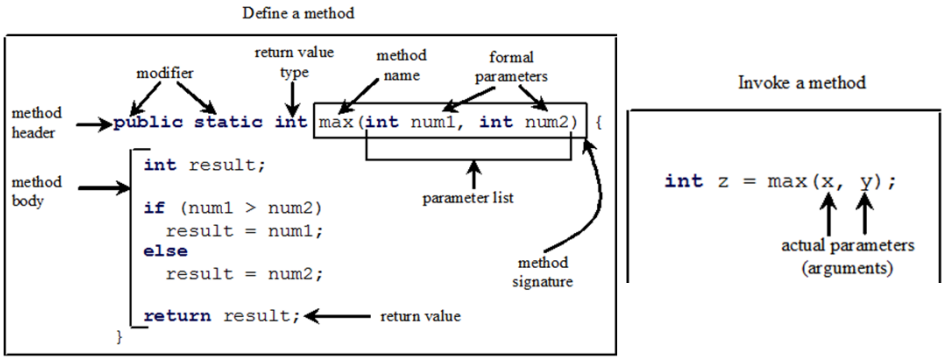
BIRZEIT UNIVERSITY

Methods

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Defining Methods

❖ A method is a collection of statements that are grouped together to perform an operation.



CAUTION

- A **return** statement is required for a value-returning method.
- The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

- To fix this problem, delete ***if (n < 0)*** in (a), so that the compiler will see a **return** statement to be reached regardless of how the **if** statement is evaluated.

3

Passing Parameters

```
public static void nPrintln(String message, int n) {
    for (int i = 0; i < n; i++)
        System.out.println(message);
}
```

- Suppose you invoke the method using
  nPrintln("Welcome to Java", 5);
  What is the output?
- Suppose you invoke the method using
  nPrintln("Computer Science", 15);
  What is the output?
- Can you invoke the method using
  nPrintln(15, "Computer Science");

4

Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
      System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

5

Scope of Local Variables

- A **local variable**: a variable defined inside a method.
- **Scope**: the part of the program where the variable can be referenced.
- The scope of a local variable **starts from its declaration and continues to the end of the block that contains the variable**.
- A local variable **must** be declared before it can be used.

6

Scope of Local Variables

❖ You can declare a local variable with the same name multiple times in different **non-nesting** blocks in a method, but you cannot declare a local variable twice in nested blocks.

It is fine to declare i in two non-nesting blocks

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

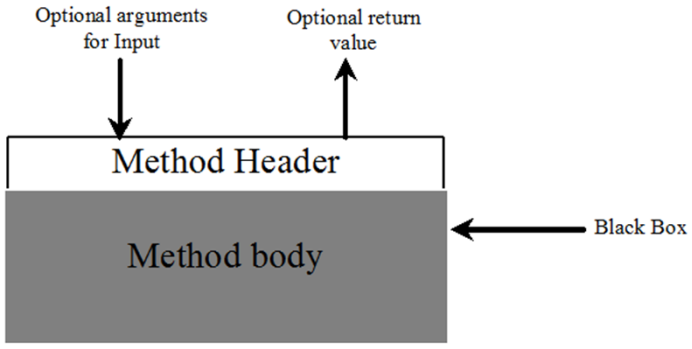
It is wrong to declare i in two nesting blocks

```
public static void method2() {
  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++) {
    sum += i;
  }
}
```

7

Method Abstraction

❖ You can think of the method body as a black box that contains the detailed implementation for the method.



8

Benefits of Methods

- Write a method once and **reuse** it anywhere.
- **Information hiding**. Hide the implementation from the user.
- **Reduce complexity**.

9

The Math Class

- Class constants:
  - **PI**
  - **E**
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

10

Trigonometric Methods

- sin(double a)
- cos(double a)
- tan(double a)
- acos(double a)
- asin(double a)
- atan(double a)

Radians

Math.toRadians(90)

Examples:

Math.sin(0)	returns 0.0
Math.sin(Math.PI / 6)	returns 0.5
Math.sin(Math.PI / 2)	returns 1.0
Math.cos(0)	returns 1.0
Math.cos(Math.PI / 6)	returns 0.866
Math.cos(Math.PI / 2)	returns 0.0

11

Exponent Methods

- exp(double a)
  Returns e raised to the power of a.
- log(double a)
  Returns the natural logarithm of a.
- log10(double a)
  Returns the 10-based logarithm of a.
- pow(double a, double b)
  Returns a raised to the power of b.
- sqrt(double a)
  Returns the square root of a.

Examples:

Math.exp(1)	returns 2.71
Math.log(2.71)	returns 1.0
Math.pow(2, 3)	returns 8.0
Math.pow(3, 2)	returns 9.0
Math.pow(3.5, 2.5)	returns 22.917
Math.sqrt(4)	returns 2.0
Math.sqrt(10.5)	returns 3.24

12

Rounding Methods

- **double ceil(double x)** x rounded up to its nearest integer. This integer is returned as a double value.
- **double floor(double x)** x is rounded down to its nearest integer. This integer is returned as a double value.
- **double rint(double x)** x is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double.
- **int round(float x)** Return (int)Math.floor(x+0.5).
- **long round(double x)** Return (long)Math.floor(x+0.5).

13

min, max, and abs

- **max(a, b) and min(a, b)**
  Returns the maximum or minimum of two parameters.
- **abs(a)**
  Returns the absolute value of the parameter.
- **random()**
  Returns a random double value in the range [0.0, 1.0).

Examples:

Math.max(2, 3)	returns 3
Math.max(2.5, 3)	returns 3.0
Math.min(2.5, 3.6)	returns 2.5
Math.abs(-2)	returns 2
Math.abs(-2.1)	returns 2.1

14

The random Method

❖ Generates a random double value greater than or equal to 0.0 and less than 1.0

(0 <= Math.random() < 1.0)

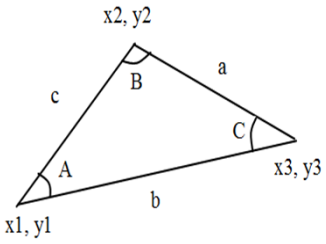
```
(int)(Math.random() * 10)
```
→ Returns a random integer between 0 and 9.

```
50 + (int)(Math.random() * 50)
```
→ Returns a random integer between 50 and 99.

In general:

```
a + Math.random() * b
```
→ Returns a random number between a and a + b, excluding a + b.

Case Study: Computing Angles of a Triangle




```
A = acos((a * a - b * b - c * c) / (-2 * b * c))
B = acos((b * b - a * a - c * c) / (-2 * a * c))
C = acos((c * c - b * b - a * a) / (-2 * a * b))
```

Write a program that prompts the user to enter the x- and y-coordinates of the three corner points in a triangle and then displays the triangle's angles.

BIRZEIT UNIVERSITY

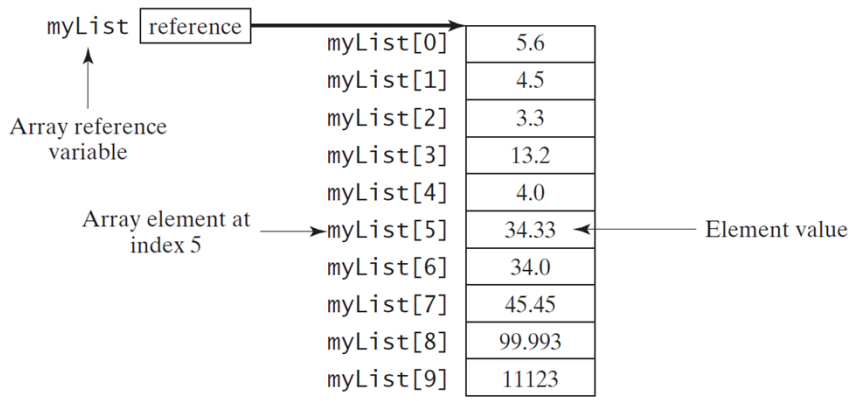
Arrays

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Introducing Arrays

❖ Array is a data structure that represents a collection of the **same** types of data.

```
double[] myList = new double[10];
```



Declaring Array Variables

datatype[] arrayRefVar;
Example:

double[] myList;

datatype arrayRefVar**[]**; **// This style is allowed, but not preferred**
Example:

double myList[];

3

Creating Arrays

arrayRefVar = new datatype[arraySize];

Example:

myList = new double[10];

- **myList[0]** references the 1st element in the array.
- **myList[9]** references the last element in the array.

4

Declaring and Creating in 1 Step

datatype[] arrayRefVar = new datatype[arraySize];

double[] myList = new double[10];

datatype arrayRefVar[] = new datatype[arraySize];

double myList[] = new double[10];

5

The Length of an Array

- Once an array is created, its **size is fixed**.
- It cannot be changed.
- You can find its size using:

arrayRefVar.length

For example:

myList.length ➔ returns 10

6

Default Values

❖ When an array is created, its elements are assigned the **default value** of :

- **0** for the numeric data types.
- '**\u0000**' for **char** types.
- **false** for **boolean** types.

7

Indexed Variables

❖ The array elements are accessed through the **index**.

❖ The array indices are ***0-based***, i.e., it starts from **0** to **arrayRefVar.length-1**.

❖ Each element in the array is represented using the following syntax, known as an *indexed variable*:

arrayRefVar[index];

8

Using Indexed Variables

❖ After an array is created, an indexed variable can be used in the same way as a regular variable.

❖ For example, the following code adds the value in **myList[0]** and **myList[1]** to **myList[2]**:

```
myList[2] = myList[0] + myList[1];
```

9

Array Initializers

❖ Declaring, creating, initializing in 1 step:

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

❖ This shorthand notation is equivalent to the following statements:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```

10

Trace Program with Arrays

```
public class Test {
  public static void main(String[] args) {
      int[] values = new int[5];
      for (int i = 1; i < 5; i++) {
          values[i] = i + values[i-1];
      }
      values[0] = values[1] + values[4];
  }
}
```

Initializing arrays with input values

```
Scanner input = new Scanner(System.in);

System.out.print("Enter " + myList.length + " values: ");

for (int i = 0 ; i < myList.length ; i++)

    myList[ i ] = input.nextDouble();
```

Initializing arrays with random values

```
for (int i = 0; i < myList.length; i++)
    myList[i] = Math.random() * 100;
```

Printing arrays

```
for (int i = 0; i < myList.length; i++)
    System.out.print(myList[i] + " ");
```

13

Summing all elements

```
double total = 0;
for (int i = 0; i < myList.length; i++)
    total += myList[i];
```

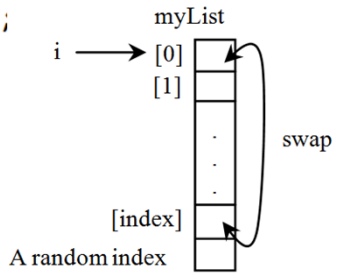
Finding the largest element

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
    if (myList[i] > max)
        max = myList[i];
}
```

14

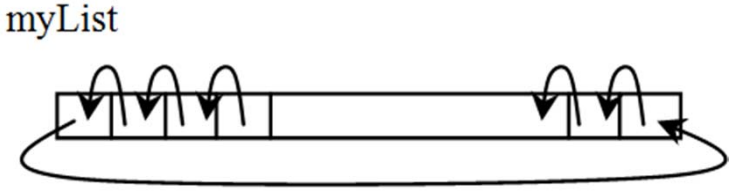
Random Shuffling

```
for (int i = 0; i < myList.length; i++) {
  // Generate an index j randomly
  int index = (int)(Math.random()
    * myList.length);

  // Swap myList[i] with myList[index]
  double temp = myList[i];
  myList[i] = myList[index];
  myList[index] = temp;
}
```



Shifting Elements

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



16

Enhanced for Loop (for-each loop)

❖ **JDK 1.5** introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable.

❖ For example, the following code displays all elements in the array **myList**:

```
for (double value: myList)   System.out.println(value);
```

❖ In general, the syntax is:

```
for (elementType value: arrayRefVar) {
    // Process the value
}
```

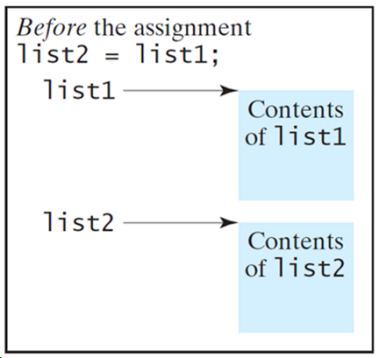
❖ You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

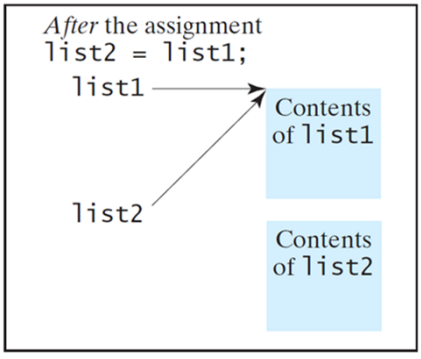
17

Copying Arrays

❖ Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```





18

Copying Arrays

❖ **Using a loop:**

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];

for (int i = 0; i < sourceArrays.length; i++)
    targetArray[i] = sourceArray[i];
```

19

The arraycopy Utility

```
System.arraycopy(sourceArray, src_pos,
        targetArray, tar_pos, length);
```

❖ Example:

```
System.arraycopy(sourceArray, 0,
  targetArray, 0, sourceArray.length);
```

20

Passing Arrays to Methods

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

- **Invoke the method**

int[] list = {3, 1, 2, 6, 4, 2};

printArray(list);

21

Anonymous Array

- The statement

  printArray(**new int[]{3, 1, 2, 6, 4, 2}**);

- Creates array using the following syntax:

  new dataType[]{literal0, literal1, ..., literalk}

- There is no explicit reference variable for the array.

- Such array is called an ***anonymous array***.

22

Pass by Value

❖ For a parameter of a **primitive type value**, **the actual value is passed**.

- Changing the value of the local parameter inside the method **does not affect** the value of the variable outside the method.

❖ For a parameter of an **array type**, the value of the parameter contains a reference to an array; **this reference is passed to the method**.

- Any changes to the array that occur inside the method body **will affect** the original array that was passed as the argument.

23

Simple Example

```
public class Test {
  public static void main(String[] args) {
    int x = 1;
    int[] y = new int[10];

    m(x, y);

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001;
    numbers[0] = 5555;
  }
}
```

24

Returning an Array from a Method

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];
  for (int i=0, j=result.length - 1; i < list.length/2; i++, j--) {
        result[j] = list[i];
  }
  return result;
}
```

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

25

Linear Search

- The linear search approach compares the key element, **key**, ***sequentially*** with each element in the array **list**.
- The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found.
- If a match is made, the linear search returns the **index** of the element in the array that matches the key.
- If no match is found, the search returns **-1**.

26

From Idea to Solution

```
public static int linearSearch(int[] list, int key) {
  for (int i = 0; i < list.length; i++)
      if (key == list[i])     return i;
  return -1;
}
```

Trace the method:

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // returns 1
int j = linearSearch(list, -4); // returns -1
int k = linearSearch(list, -3); // returns 5
```

27

The Arrays.binarySearch Method

- Since binary search is frequently used in programming, Java provides several **binarySearch** methods for searching a key in an array of int, double, char, short, long, and float in the **java.util.Arrays** class.

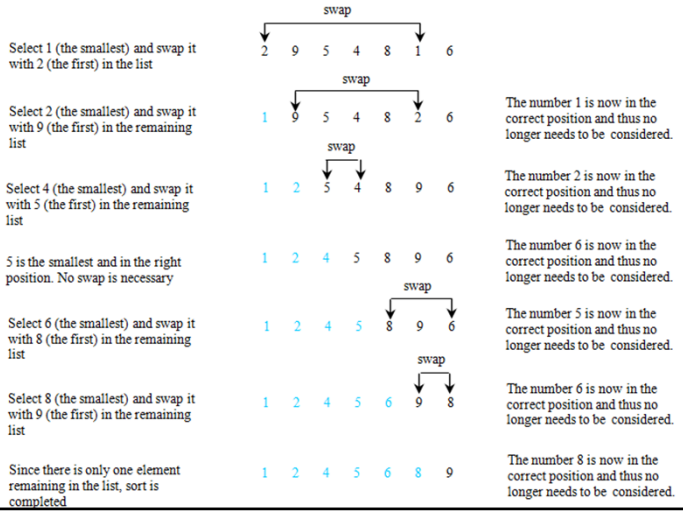
```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("Index is " +  Arrays.binarySearch(list, 11));

char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("Index is " + Arrays.binarySearch(chars, 't'));
```

- For the **binarySearch** method to work, the array must be pre-sorted in increasing order.

28

Selection Sort

❖ Selection sort finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it second, and so on until the list contains only a single number.



29

From Idea to Solution

```
for (int i = 0; i < list.length; i++)  {
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

30

The Arrays.sort Method

- Java provides several sort methods for sorting an array of **int**, **double**, **char**, **short**, **long**, and **float** in the **java.util.Arrays** class.
- For example, the following code sorts an array of numbers and an array of characters:

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```

main Method is just a Regular Method

- You can call a regular method by passing actual parameters.
- You can pass **arguments** to **main**.
- For example, the main method in class **B** is invoked by a method in **A**, as shown below:

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    B.main(strings);
  }
}
```

```
class B {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

Command-Line Parameters

```
class TestMain {
   public static void main(String[] s) {
        ...
   }
}
```

java TestMain arg0 arg1 arg2 ... argn

❖ In the **main** method, get the arguments from **s[0], s[1], ..., s[n]**, which corresponds to **arg0, arg1, ..., argn** in the command line.

33

Problem: Calculator

❖ Objective: Write a program that will perform binary operations on integers. The program receives three parameters: an operator and two integers.

```
java Calculator 2 + 3
java Calculator 2 - 3
java Calculator 2 / 3
java Calculator 2 . 3
```

34

Declare/Create 2D Arrays

```
// Declare array refvar
dataType[][] refVar;

// Create array and assign its reference to variable
refVar = new dataType[10][10];

// Combine declaration and creation in one statement
dataType[][] refVar = new dataType[10][10];

// Alternative syntax
dataType refVar[][] = new dataType[10][10];
```

35

Creating 2D Arrays

```
int[][] matrix = new int[10][10];

for (int i = 0; i < matrix.length; i++)
  for (int j = 0; j < matrix[i].length; j++)
    matrix[i][j] = (int)(Math.random() * 1000);
```

36

Declaring, Creating, and Initializing Using Shorthand Notations

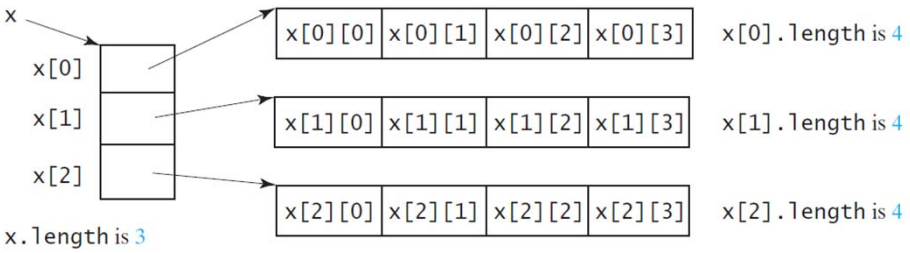
- You can also use an array initializer to declare, create and initialize a 2-dimensional array.
- For example:

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

37

Lengths of 2D Arrays

int[][] x = new int[3][4];



38

Lengths of 2D Arrays, cont.

```
int[][] array = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9},
      {10, 11, 12}
};
```

array.length
array[0].length
array[1].length
array[2].length
array[3].length

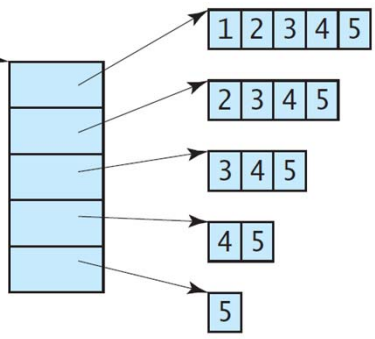
array[4].length → **ArrayIndexOutOfBoundsException**

Ragged Arrays

- Each row in a 2D array is **itself** an array. So, the **rows can have different lengths**.
- Such an array is known as *a* ***ragged array***.

For example:

```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



Printing arrays

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length;
                    column++) {
        System.out.print(matrix[row][column] + " ");
  }

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

41

What is Sudoku?

Checking Whether a Solution Is Correct

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6							
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

42

Multidimensional Arrays

❖ Occasionally, you will need to represent

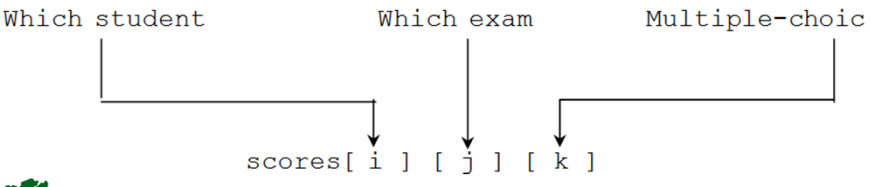
n-dimensional data structures.

❖ In Java, you can create n-dimensional arrays for any integer n.

❖ The way to declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare n-dimensional array variables and create n-dimensional arrays for **n > 2**.

Multidimensional Arrays

```
double[][][] scores = {
  {{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},
  {{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},
  {{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},
  {{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},
  {{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},
  {{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}}};
```



BIRZEIT UNIVERSITY

OO Basic Concepts

By: Mamoun Nawahdah (Ph.D.)
2015/2016

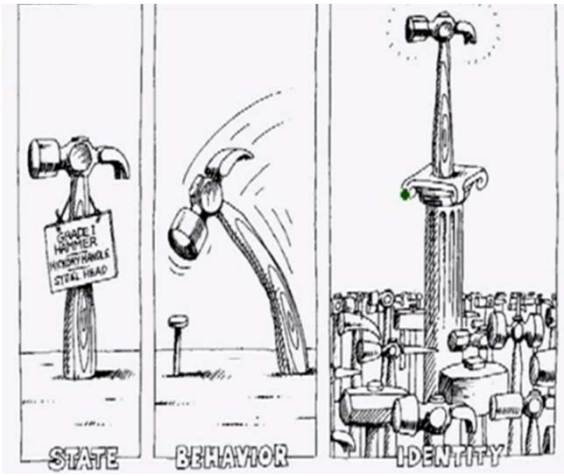
Problems with Procedural Languages

- Data does not have an owner.
- Difficult to maintain data integrity.
- Functions are building blocks.
- Many functions can modify a given block of data.
- Difficult to trace bug sources when data is corrupted.

What is Object?

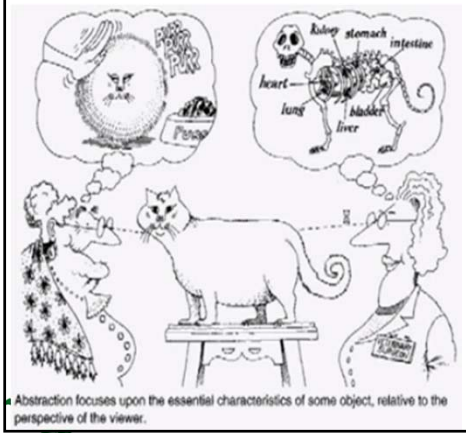
- An object has **state**, exhibits some well defined **behaviour**, and has a unique **identity**.

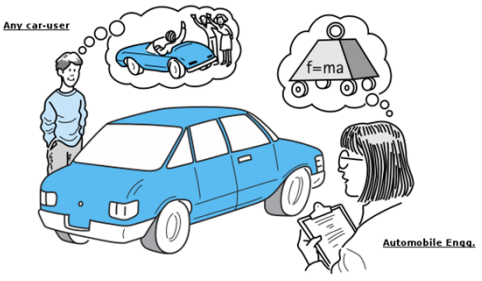
  - State
    - data members
    - fields
    - properties
  - Behavior
    - member functions
    - methods



Abstraction - Modeling

- **Abstraction** focuses upon the **essential** characteristics of some object, relative to the perspective of the viewer.

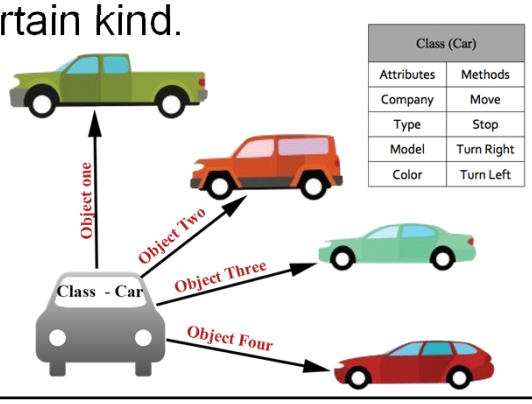




An abstraction includes the essential details relative to the perspective of the viewer

What is Class?

• A **class** represents a set of objects that share common structure and a common behavior.
• A **class** is a **blueprint** or **prototype** that defines the variables and methods common to all objects of a certain kind.



Class Access

PROBLEM: You have a garden and it is **public**. Anyone can take the properties of the garden when they want.

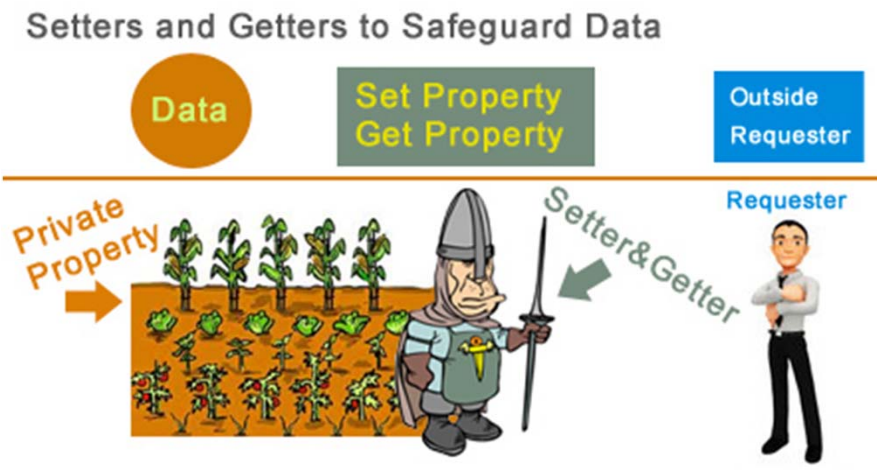
Class Access cont.

SOLUTION? Put a high fence around my garden, now it is safe! But waite, I can no longer access my own garden.

Class Access cont.

SOLUTION: Hire a **private** guard and give him **rules** on who is able to access the garden. Anyone wanting to use the garden must get permission from guard. garden is now **safe** and **accessible**.

Class Access cont.



Initialization of Objects

- **Constructors** ensure correct initialization of all data. They are automatically called at the time of object creation.
- **Destructors** on the other hand ensure the de allocation of resources before an object dies or goes out of scope.

Lifecycle of an Object

❖ Some call it the holy trinity of OOP:

> Born Healthy

Using constructors

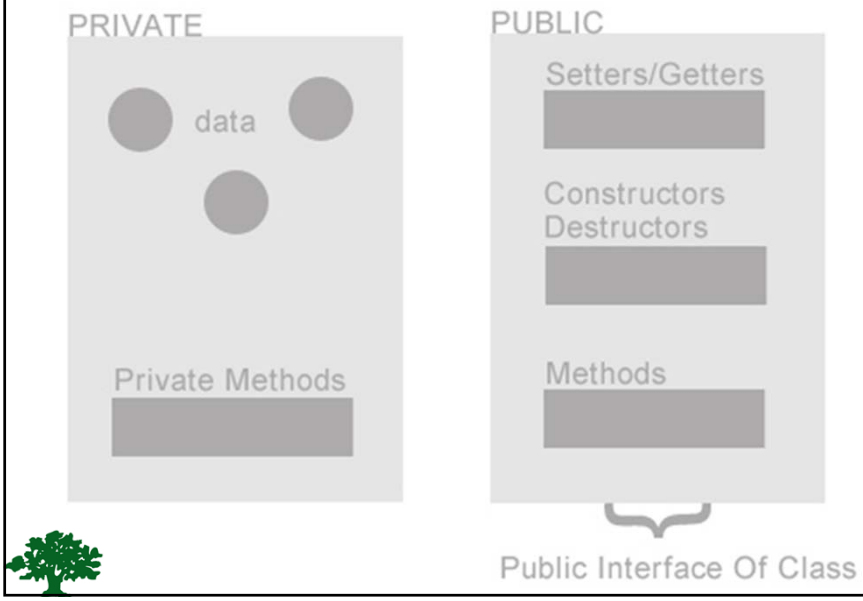
>Lives safely

Using setters and getters

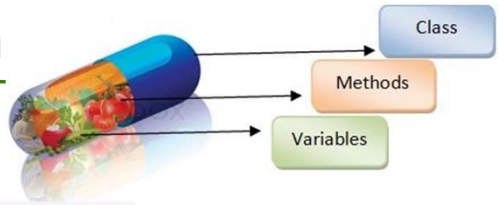
>Dies cleanly

Using destructors

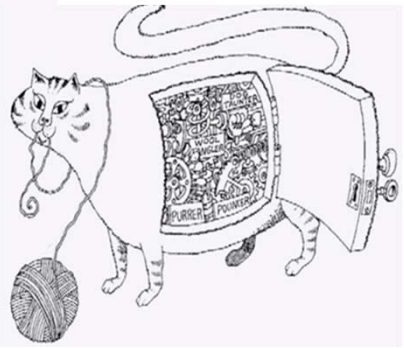
Anatomy of a Class



Encapsulation



- FIRST LAW OF OOP: Data must be hidden, i.e., PRIVATE
- Read access through read functions
- Write access through write functions
- For every piece of data, 4 possibilities
  >> read and write allowed
  >> read only
  >> write only
  >> no access



Encapsulation

- ❖ Encapsulation is used to hide unimportant implementation details from other objects.
- ❖ In real world
  - When you want to change gears on your car:
    - You don't need to know how the gear mechanism works.
    - You just need to know which lever to move.

Encapsulation cont.

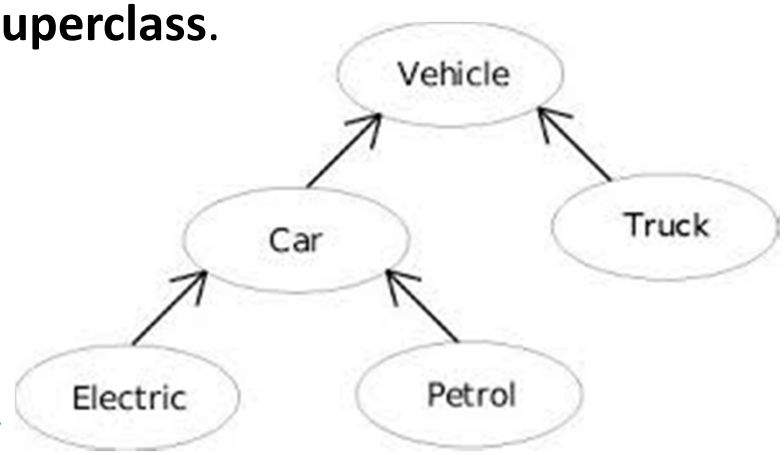
- In software programs:
  - You don't need to know how a class is implemented.
  - You just need to know which methods to invoke.
  - Thus, the implementation details can change at any time without affecting other parts of the program.

Inheritance

- Extending the functionality of a class or
- Specializing the functionality of the class.

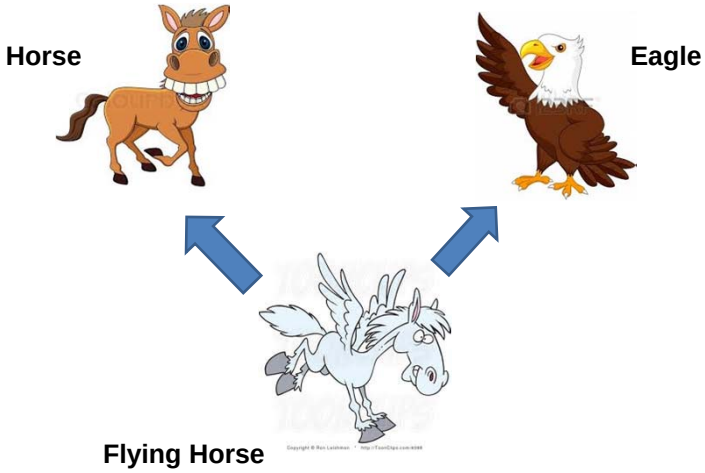
Inheritance cont.

❖ **Subclasses**: a subclass may inherit the structure and behaviour of it's **superclass**.



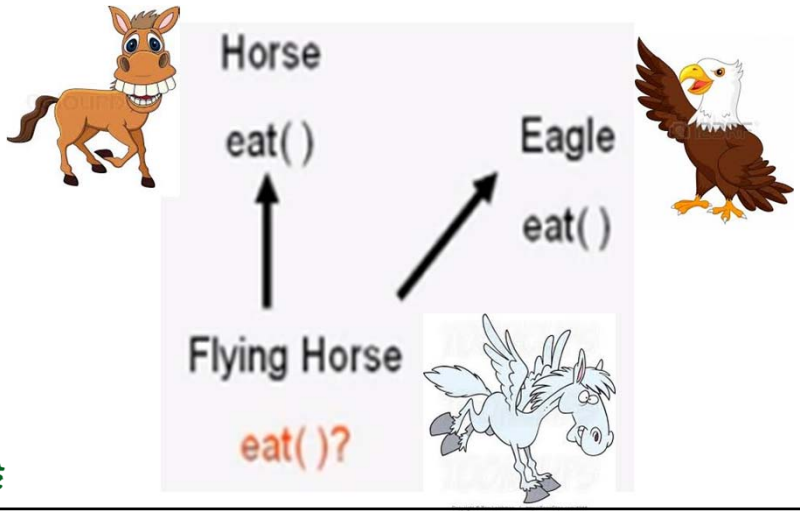
Multiple Inheritance

❖ One class have more than one base class.



Multiple Inheritance cont.

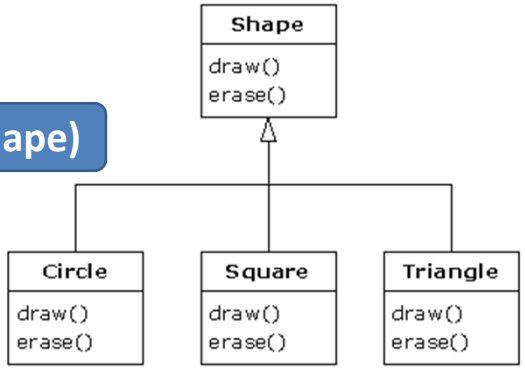
❖ Ambiguity in multiple inheritance:



Polymorphism

❖ **Polymorphism** refers to the ability of an object to provide different behaviours (use different implementations) depending on its own nature. Specifically, depending on its position in the class hierarchy.

drawShape (class Shape)



BIRZEIT UNIVERSITY

Objects & Classes

By: Mamoun Nawahdah (Ph.D.)
2015/2016

OO Programming Concepts

❖ Object-oriented programming (OOP) involves programming using objects.

❖ An ***object*** represents an entity in the real world that can be distinctly identified.

❖ For example, a ***student***, a ***desk***, a ***circle***, a ***button***, and even a **loan** can all be viewed as objects.

❖ An object has a unique **identity**, **state**, and **behaviors**.

- The ***state*** of an object consists of a set of *data fields* (also known as ***properties***) with their current values.
- The ***behavior*** of an object is defined by a set of **methods**.

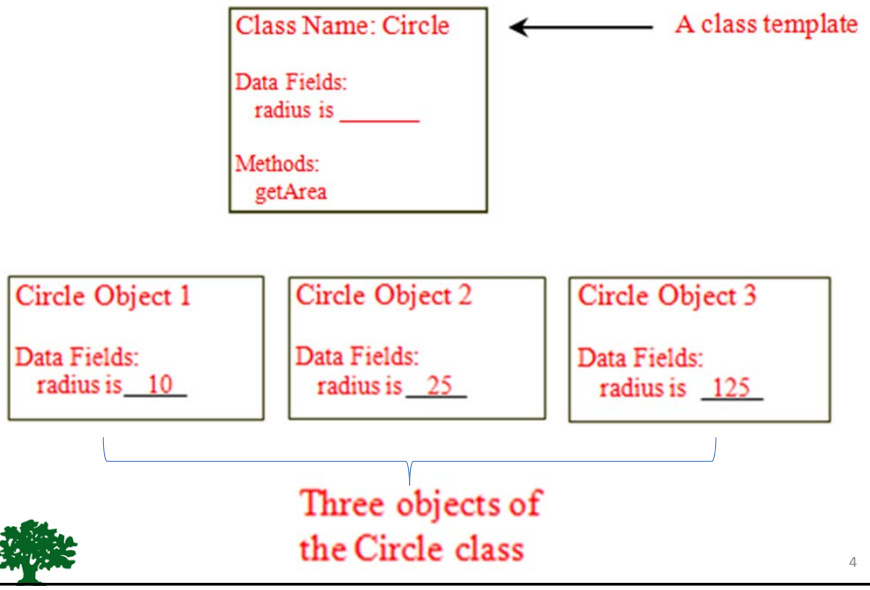
2

Objects and Classes

- An object has both a ***state*** and ***behavior***.
- The ***state*** defines the object, and the ***behavior*** defines what the object does.
- ***Classes*** are constructs that define objects of the same type.
- A Java class uses ***variables*** to define data fields and ***methods*** to define behaviors.
- Additionally, a class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class.

3

Objects and Classes cont.



4

Classes

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

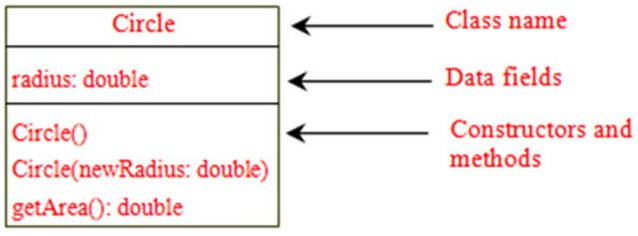
Constructors

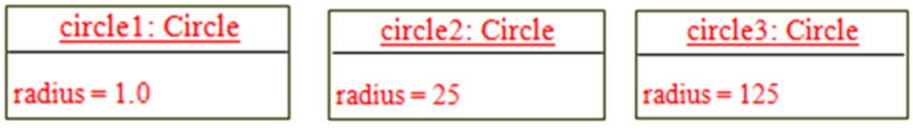
Method

5

UML Class Diagram

UML Class Diagram





UML notation for objects

6

Constructors

❖ Constructors are a ***special kind of methods*** that are invoked to construct objects.

```
Circle() {
}

Circle(double newRadius) {
    radius = newRadius;
}
```

7

Constructors cont.

❖ A constructor with no parameters is referred to as a ***no-arg constructor***.

❖ Constructors **must** have the same name as the class itself.

❖ Constructors do not have a return type—not even void.

❖ Constructors are invoked using the **new** operator when an object is created.

❖ Constructors play the role of initializing objects.

8

Creating Objects Using Constructors

new ClassName();

Example:

new Circle();

new Circle(5.0);

Default Constructor

❖ A class maybe defined **without** constructors.

❖ In this case, a **no-arg constructor** with an empty body is **implicitly** declared in the class.

❖ This constructor, called *a **default constructor***, is provided **automatically**

ONLY IF *no constructors are **explicitly** defined in the class*.

Declaring Object Reference Variables

❖ To reference an object, assign the object to a reference variable.

❖ To declare a reference variable, use the syntax:

ClassName objectRefVar;

Example:

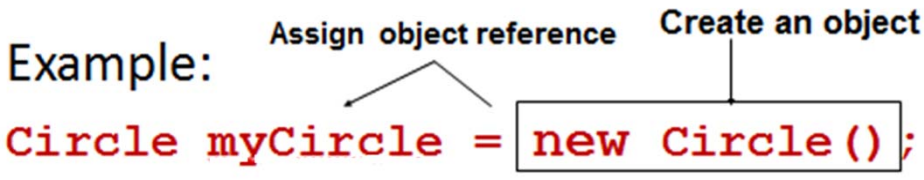
Circle myCircle;

11

Declaring/Creating Objects in a Single Step

ClassName objectRefVar = new ClassName();

Example:



12

Accessing Object's Members

- Referencing the object's data:

  objectRefVar.data

  e.g., **myCircle.radius**

- Invoking the object's method:

  objectRefVar.methodName(arguments)

  e.g., **myCircle.getArea()**

13

Reference Data Fields

- The data fields can be of reference types.
  - If a data field of a **reference** type does not reference any object, the data field holds a special literal value, **null**.
  - For example, the following **Student** class contains a data field **name** of the **String** type.

```
public class Student {
  String name; // name has default value null
  int age;     // age has default value 0
  boolean isScienceMajor; // default false
  char gender; // default value '\u0000'
}
```

14

Default Value for a Data Field

- The default value of a data field is:
  - **null** for a ***reference*** type
  - **0** for a ***numeric*** type
  - **false** for a ***boolean*** type
  - '**\u0000**' for a ***char*** type
- However, **Java assigns no default value to a local variable inside a method.**

15

Example

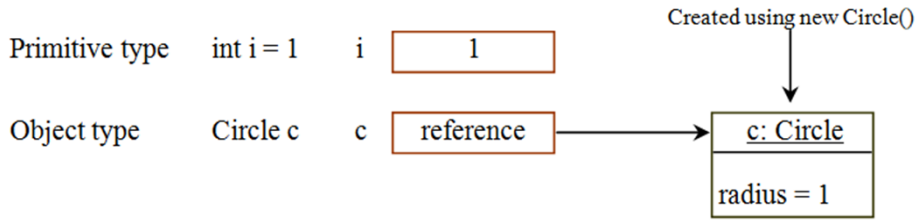
- Java assigns **no** default value to a local variable inside a method.

```
public class Test {
  public static void main(String[] args) {
      int x;       // x has no default value
      String y;          // y has no default value
      System.out.println("x is " + x);
      System.out.println("y is " + y);
  }
}
```

Compilation error: variables not initialized

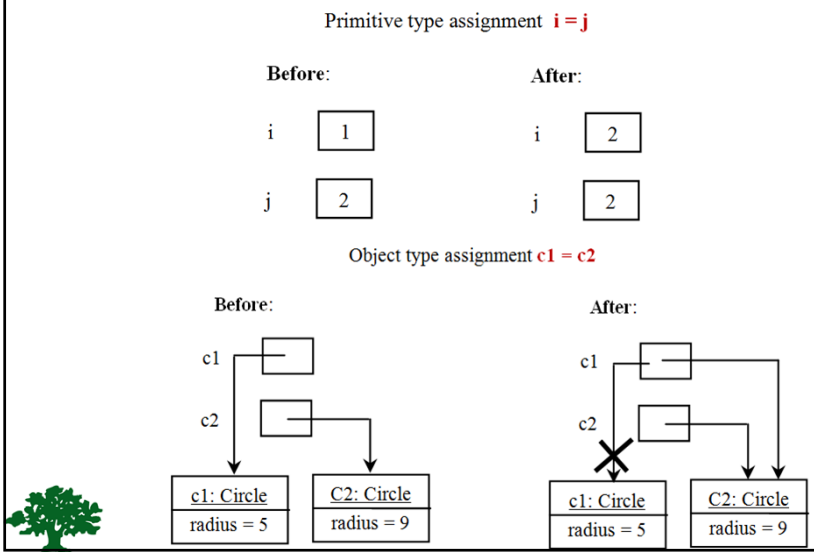
16

Differences between Variables of Primitive Data Types and Object Types



17

Copying Variables of Primitive Data Types and Object Types



18

Garbage Collection

❖ As shown in the previous figure, after the assignment statement **c1 = c2**, **c1** points to the same object referenced by **c2**.

❖ The object previously referenced by **c1** is no longer referenced.

❖ This object is known as **garbage**.

❖ Garbage is automatically collected by **JVM**.

19

The Date Class

❖ Java provides a system-independent encapsulation of date and time in the **java.util.Date** class.

❖ You can use the **Date** class to create an instance for the current date and time and use its **toString** method to return the date and time as a **string**.

The + sign indicates public modifer →

java.util.Date	
+Date()	Constructs a Date object for the current time.
+Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
+toString(): String	Returns a string representing the date and time.
+getTime(): long	Returns the number of milliseconds since January 1, 1970, GMT.
+setTime(elapseTime: long): void	Sets a new elapse time in the object.

20

The Date Class Example

❖ For example, the following code:

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

- displays a string like:

Mon Nov 04 19:50:54 IST 2013

21

The Random Class

❖ You have used **Math.random()** to obtain a random double value between **0.0** and **1.0** (excluding 1.0).

❖ A more useful random number generator is provided in the **java.util.Random** class.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

22

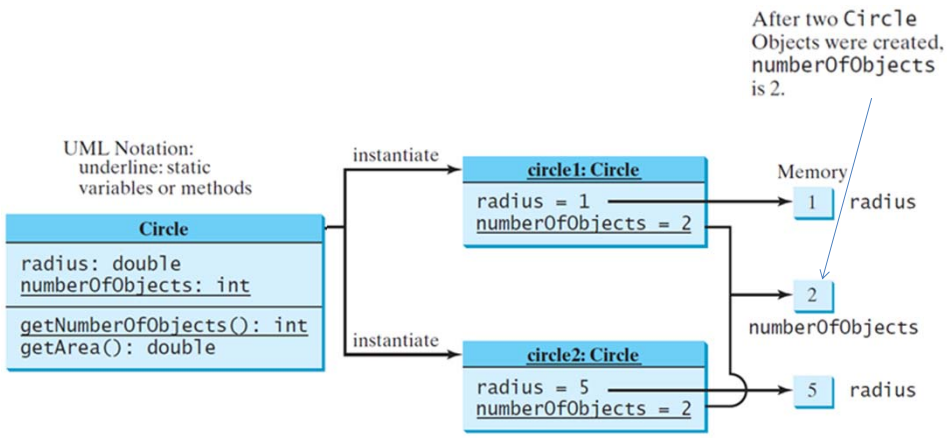
Instance Variables, and Methods

- **Instance variables** belong to a specific instance.
- **Instance methods** are invoked by an instance of the class.

Static Variables, Constants, and Methods

- **Static variables** are shared by all the instances of the class.
- **Static methods** are not tied to a specific object.
- **Static constants** are final variables shared by all the instances of the class.
- To declare static ***variables***, ***constants***, and ***methods***, use the **static** modifier.

Static



25

Static Variable

- It is a variable which belongs to the **class** and not to the **object (instance**).
- Static variables are **initialized only once**, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
- A **single copy** to be shared by all instances of the class.
- A static variable can be **accessed directly** by the **class name** and doesn't need any object.

Syntax : ***<class-name>.<static-variable-name>***

Static Method

- It is a method which **belongs to the class** and **not** to the **object** (instance).
- A static method **can access only static data**. It can not access non-static data (instance variables).
- A static method **can call only** other **static methods** and can not call a non-static method from it.
- A static method can be **accessed directly** by the **class name** and doesn't need any object.

  Syntax : ***<class-name>.<static-method-name>***
- A static method cannot refer to "**this**" or "**super**" keywords in anyway.

main method is static, since it must be accessible for an application to run, before any instantiation takes place.

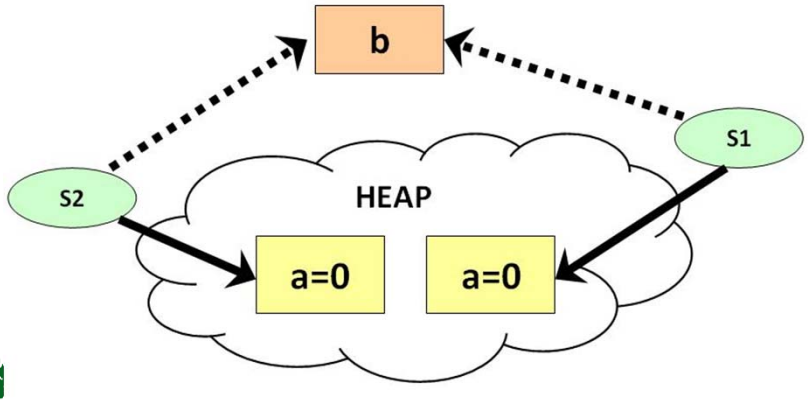
Static example

```
1  class Student {
2  int a; //initialized to zero
3  static int b; //initialized to zero only when class is loaded
4
5    Student(){
6     //Constructor incrementing static variable b
7     b++;
8    }
9
10    public void showData(){
11       System.out.println("Value of a = "+a);
12       System.out.println("Value of b = "+b);
13    }
14 //public static void increment(){
15 //a++;
16 //}
17
18 }
19
20 class Demo{
21    public static void main(String args[]){
22       Student s1 = new Student();
23       s1.showData();
24       Student s2 = new Student();
25       s2.showData();
26       //Student.b++;
27       //s1.showData();
28    }
29 }
```

```
C:\WINDOWS\system32\cmd.exe
C:\workspace>java Demo
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```

Static example cont.

❖ Following diagram shows , how reference variables & objects are created and static variables are accessed by the different instances.



Visibility Modifiers

❖ **By default**, the ***class***, ***variable***, or ***method*** can be accessed by any class in the same package.

☞ **public**: The ***class***, ***data***, or ***method*** is visible to any class in any package.

☞ **private**: The ***data*** or ***methods*** can be accessed only by the declaring class.

❖ The **get** and **set** methods are used to read and modify private properties.

30

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

The **private** modifier restricts access to **within a class**.

The **default** modifier restricts access to **within a package**.

The **public** modifier enables **unrestricted access**.

31

NOTE

❖ An object **cannot** access its private members, as shown in (**b**). It is OK, however, if the object is declared in its own class, as shown in (**a**).

```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

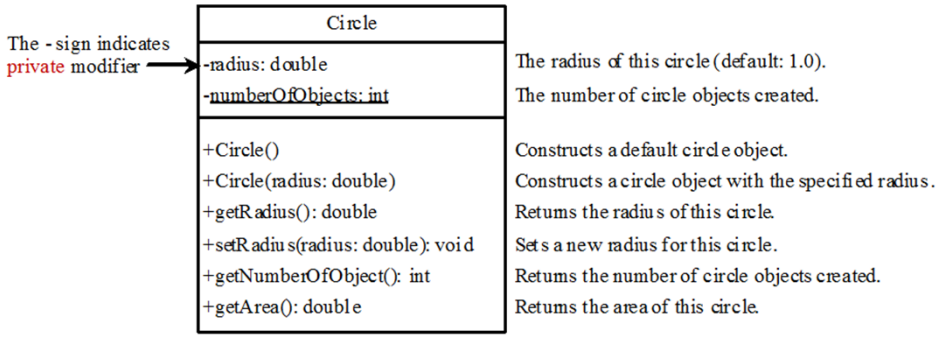
(a) This is okay because object `c` is used inside the class `C`.

```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because `x` and `convert` are private in class `C`.

32

Example of Data Field Encapsulation



33

Overloading Methods and Constructors

- In a class, there can be **several methods with the same name**. However they **must** have **different *signature***.
- The signature of a method is comprised of its ***name***, its ***parameter types*** and the ***order of its parameter***.
- The signature of a method is **not** comprised of its ***return type*** nor ***its visibility*** nor its ***thrown exceptions***.

Passing Objects to Methods

- Passing by value for primitive type value (the **value** is passed to the parameter).
- Passing by value for reference type value (the value is the **reference** to the object).

Passing Objects to Methods

```
public class TestPassObject {
  public static void main(String[] args) {
      Circle  myCircle = new Circle(1);
     // Print areas for radius 1, 2, 3, 4, and 5.
     int n = 5;
     printAreas(myCircle, n);
     System.out.println("\n" + "Radius is " + myCircle.getRadius());
     System.out.println("n is " + n);
  }

  /** Print a table of areas for radius */
  public static void printAreas( Circle c, int times) {
    System.out.println("Radius \t\tArea");
    while (times >= 1) {
        System.out.println(c.getRadius() + "\t\t" + c.getArea());
        c.setRadius(c.getRadius() + 1);
         times--;
    }
  }
}
```

Array of Objects

Circle[] circleArray = new Circle[10];

- An array of objects is actually an ***array of reference variables***.
- So invoking **circleArray[1].getArea()** involves two levels of referencing as shown in the next figure.

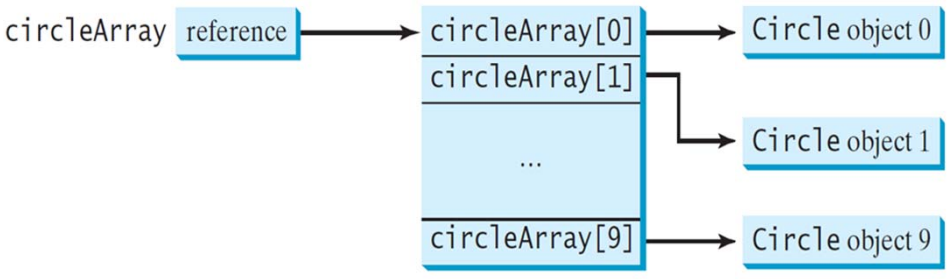
circleArray references to the entire array.

circleArray[1] references to a Circle object.

37

Array of Objects

Circle[] circleArray = new Circle[10];




```
circleArray[0] = new Circle();
circleArray[1] = new Circle();
        :
circleArray[9] = new Circle();
```

Immutable Objects and Classes

❖ If the contents of an object (instance) **can't** be changed once the object is created, the object is called an ***immutable object*** and its class is called an ***immutable class***.

39

Immutable Objects and Classes

❖ If you delete the **set** method in the **Circle** class, the class would be **immutable** because **radius** is private and cannot be changed without a **set** method.

```
public class Circle {
  private double radius = 1;

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public void setRadius(double r) {
    radius = r;
  }
}
```

40

Immutable Objects and Classes

- A class with **all private data** fields and without **mutators** is not necessarily immutable.
- For example, the following class **Student** has all **private** data fields and no **mutators**, but it is mutable!!!

Example

```
import java.util.Date;
public class Student {
  private int id;
  private Date birthDate;

  public Student(int ssn, Date newBD) {
     id = ssn;
     birthDate = newBD;
   }

   public int getId() {    return id;    }

   public Date getBirthDate() {    return birthDate;    }
}
```

```
public class Test {
  public static void main(String[] args) {
    java.util.Date bd = new java.util.Date();
    Student student = new Student(111223333, bd);
    java.util.Date date = student.getBirthDate();
    date.setMonth(5); // Now the student birthdate is changed!
  }
}
```

What Class is Immutable?

- For a class to be immutable:
  - It must mark all data fields **private.**
  - Provide **no mutator** methods.
  - No accessor methods that would return a reference to a mutable data field object.

43

Scope of Variables

- The scope of **instance** and **static** variables is the entire class. They can be declared anywhere inside a class.
- The scope of a **local** variable starts from its declaration and continues to the end of the block that contains the variable.
- A local variable **must** be initialized explicitly before it can be used.

44

Scope of Variables

❖ What is the output?

```
public class A{
  int year = 2014;   // instance variable

  void p() {
     System.out.println("Year: "+ year);
     int year = 2015;   // local variable
     System.out.println("Year: "+ year);
  }
}
```

The this Keyword

❖ The **this** keyword is the name of a **reference** that refers to an **object itself**.

❖ One common use of the **this** keyword is reference a class's ***hidden data fields***.

❖ Another common use of the **this** keyword to enable a **constructor** to invoke another **constructor** of the same class.

46

Reference the Hidden Data Fields

```
public class F {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    F.k = k;
  }
}
```

Suppose that f1 and f2 are two objects of F.
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute
this.i = 10, where *this* refers f1

Invoking f2.setI(45) is to execute
this.i = 45, where *this* refers f2

47

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

48

BIRZEIT UNIVERSITY

Strings

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Constructing Strings

String newString = **new** **String**(stringLiteral);

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

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

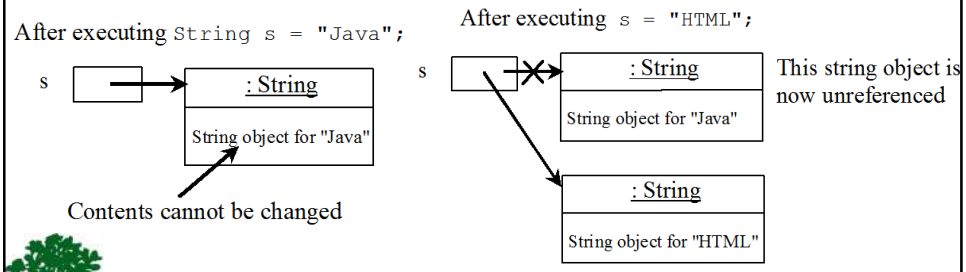
String message = "Welcome to Java";

Strings Are Immutable

❖ A **String** object is immutable; its contents cannot be changed.

❖ Does the following code change the contents of the string **s**?

```
String s = "Java";
s = "HTML";
```

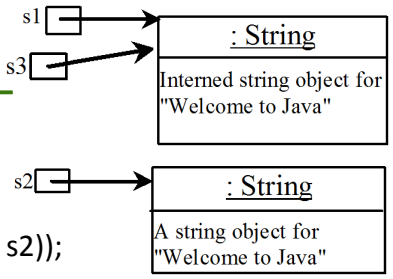


Interned Strings

❖ Since strings are immutable and are frequently used, **to improve efficiency and save memory**, the **JVM** uses a **unique** instance for string literals with the same character sequence.

❖ Such an instance is called **interned**.

Example




```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));

System.out.println("s1 == s3 is " + (s1 == s3));
```

Display:

s1 == s2 is **false**

s1 == s3 is **true**

- A new object is created if you use the **new** operator.
- If you use the string **initializer**, no new object is created **if** the interned object is already created.

5

String Comparisons

java.lang.String	
+equals(s1: Object): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case-insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case-insensitive.
+regionMatches(toffset: int, s1: String, offset: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, toffset: int, s1: String, offset: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case-sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

6

String Comparisons

```
String s1 = new  String("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
}
```

7

String Comparisons

compareTo(Object object)

```
String s1 = new String("Welcome");
String s2 = “Welcome";

if (s1.compareTo(s2) > 0) {
     // s1 is greater than s2
}
else if (s1.compareTo(s2) == 0) {
        // s1 and s2 have the same contents
      }
      else {
         // s1 is less than s2
      }
```

8

String Length, Characters, and Combining Strings

java.lang.String	
+length(): int	Returns the number of characters in this string.
+charAt(index: int): char	Returns the character at the specified index from this string.
+concat(s1: String): String	Returns a new string that concatenate this string with string s1.

Finding String Length

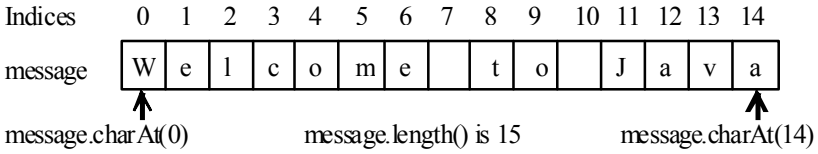
Finding string length using the **length()** method:

```
message = "Welcome to Java";
message.length();  // returns 15
```

9

Retrieving Individual Characters in a String

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



10

String Concatenation

String s3 = s1.concat(s2);

String s3 = s1 + s2;

s1 + s2 + s3 + s4 + s5

same as

(((s1.concat(s2)).concat(s3)).concat(s4)).concat(s5);

11

Extracting Substrings

java.lang.String	
+substring(beginIndex: int): String	Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string, as shown in Figure 8.6.
+substring(beginIndex: int, endIndex: int): String	Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1, as shown in Figure 8.6. Note that the character at endIndex is not part of the substring.

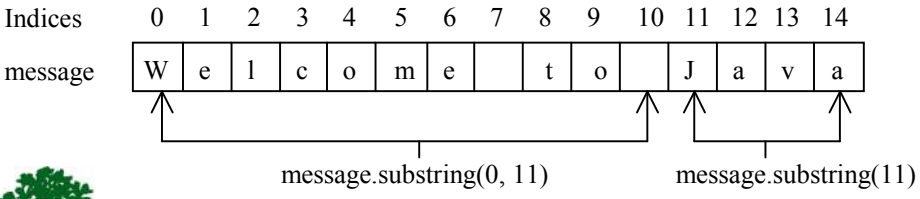
12

Extracting Substrings

❖ You can extract a single character from a **string** using the **charAt** method.

❖ You can also extract a substring from a **string** using the **substring** method in the **String** class.

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

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



13

Converting, Replacing, and Splitting Strings

java.lang.String	
+toLowerCase(): String	Returns a new string with all characters converted to lowercase.
+toUpperCase(): String	Returns a new string with all characters converted to uppercase.
+trim(): String	Returns a new string with blank characters trimmed on both sides.
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

14

Examples

"Welcome".toLowerCase()

returns a new string, **welcome**

"Welcome".toUpperCase()

returns a new string, **WELCOME**

" Welcome ".trim()

returns a new string, **Welcome**

"Welcome".replace('e', 'A')

returns a new string, **WAlcomA**

"Welcome".replaceFirst("e", "AB")

returns a new string, **WABlcome**

"Welcome".replaceAll("e", "AB")

returns a new string, **WABlcomAB**

15

Splitting a String

```
String s1 =  "Java#HTML#Perl";
String[] tokens = s1.split( "#" , 0);
for (int i = 0; i < tokens.length; i++)
        System.out.println( tokens[i]  );
```

Displays:

Java
HTML
Perl

16

Matching, Replacing and Splitting by Patterns

❖ You can **match**, **replace**, or **split** a string by specifying a pattern.

❖ This is an extremely useful and powerful feature, commonly known as ***regular expression***.

```
"Java".matches("Java")
"Java".equals("Java")
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
```

17

Matching, Replacing and Splitting by Patterns

❖ The **replaceAll**, **replaceFirst**, and **split** methods can be used with a regular expression.

❖ For example, the following statement returns a new string that **replaces** **$**, **+**, or **#** in "**a+b$#c**" by the string **NNN**.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression **[$+#]** specifies a pattern that matches **$**, **+**, or **#**.

So, the output is **aNNNbNNNNNNc**

18

Matching, Replacing and Splitting by Patterns

❖ The following statement **splits** the string into an array of strings delimited by some punctuation marks:

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");
for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

```
Java
C
C#
C++
```

19

Finding a Character or a Substring in a String

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

Finding a Character or a Substring in a String

String s = "Welcome to Java";

s.indexOf('W')	returns **0**
s.indexOf('x')	returns **-1**
s.indexOf('o', 5)	returns **9**
s.indexOf("come")	returns **3**
s.indexOf("Java", 5)	returns **11**
s.indexOf("java", 5)	returns **-1**
s.lastIndexOf('a')	returns **14**

21

Convert Character and Numbers to Strings

❖ The **String** class provides several static **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**.

❖ For example, to convert a **double** value to a **string**, use **String.valueOf(5.44)**. The return value is string consists of characters **'5', '.', '4',** and **'4'**.

22

The Character Class

java.lang.Character	
+Character(value: char)	Constructs a character object with char value
+charValue(): char	Returns the char value from this object
+compareTo(anotherCharacter: Character): int	Compares this character with another
+equals(anotherCharacter: Character): boolean	Returns true if this character equals to another
+isDigit(ch: char): boolean	Returns true if the specified character is a digit
+isLetter(ch: char): boolean	Returns true if the specified character is a letter
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter
+toLowerCase(ch: char): char	Returns the lowercase of the specified character
+toUpperCase(ch: char): char	Returns the uppercase of the specified character

23

Examples

Character c = new Character('b');

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

24

StringBuilder and StringBuffer

- The **StringBuilder/StringBuffer** class is an alternative to the **String** class.
- In general, a **StringBuilder/StringBuffer** can be used wherever a **String** is used.
- **StringBuilder/StringBuffer** is more **flexible** than **String**.
- You can **add**, **insert**, or **append** new contents into a string buffer, whereas the value of a **String** object is fixed once the string is created.

25

StringBuilder Constructors

java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

26

Modifying Strings in the Builder

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

27

Examples

```
StringBuilder sb = new StringBuilder("Welcome to ");

sb.append("Java");
sb.insert(11, "HTML and ");
sb.delete(8, 11) ;
                    // changes the builder to Welcome Java
sb.deleteCharAt(8) ;
                    // changes the builder to Welcome o Java
sb.reverse() ;
                    // changes the builder to avaJ ot emocleW
sb.replace(11, 15, "HTML") ;
                // changes the builder to Welcome to HTML
sb.setCharAt(0, 'w') ;
                    // sets the builder to welcome to Java
```

28

The toString, capacity, length, setLength, and charAt Methods

java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex-1.
+trimToSize(): void	Reduces the storage size used for the string builder.

29

BIRZEIT UNIVERSITY

Thinking in Objects

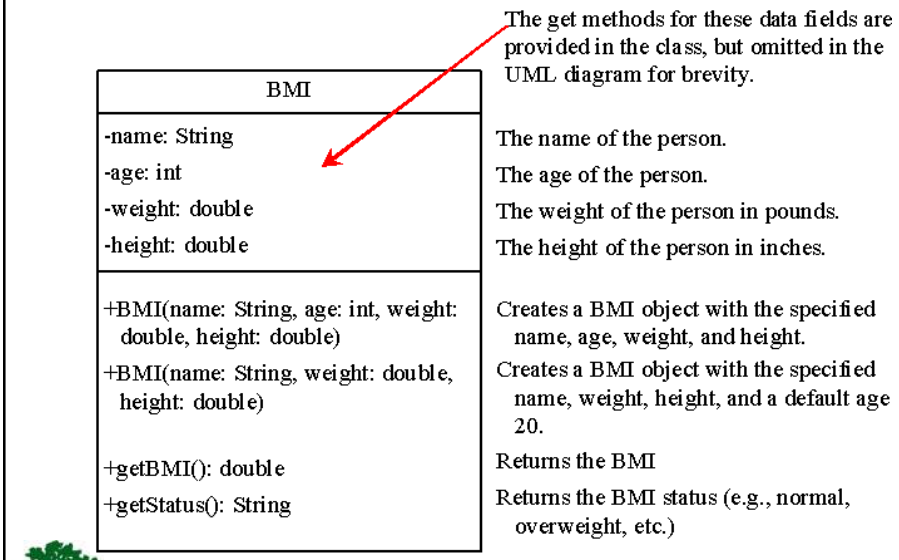
By: Mamoun Nawahdah (PhD)
2015/2016

Class Abstraction and Encapsulation

- Class ***abstraction*** means to separate class implementation from the use of the class.
- The creator of the class provides a description of the class and let the user know how the class can be used.
- The user of the class does not need to know how the class is implemented.
- The detail of implementation is encapsulated and hidden from the user.

Class implementation is like a black box hidden from the clients → **Class** | Class Contract (Signatures of public methods and public constants) ↔ Clients use the class through the contract of the class

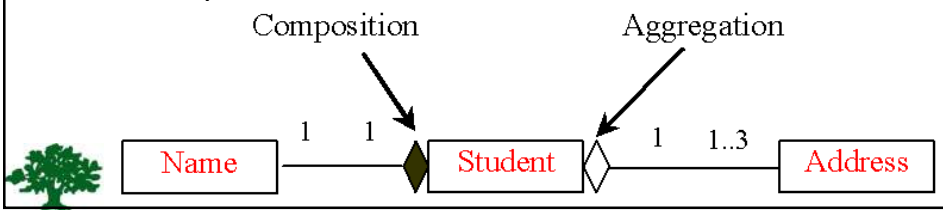
Case Study: The BMI Class



3

Object Composition

- **Aggregation** models ***has-a*** relationships and represents an **ownership** relationship between two objects.
- The owner object is called an ***aggregating object*** and its class an ***aggregating class***.
- The subject object is called an ***aggregated object*** and its class an ***aggregated class***.
- **Composition** is actually a special case of the aggregation relationship.



Class Representation

❖ An **aggregation** relationship is usually represented as a data field in the aggregating class.

❖ For example, the relationship in the previous Figure can be represented as follows:

```
public class Name {
  ...
}
```

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

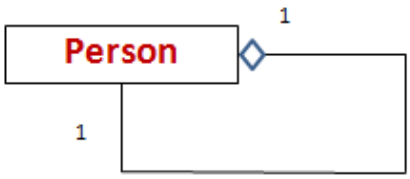
```
public class Address {
  ...
}
```

5

Aggregation Between Same Class

❖ Aggregation may exist between objects of the same class.

❖ For example, a **person** may have a **supervisor**:

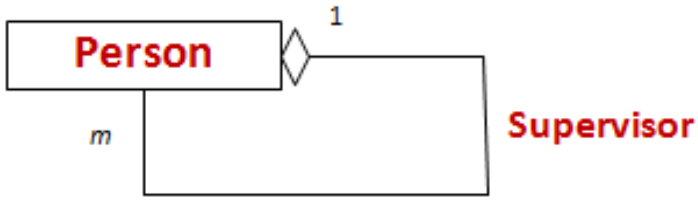


Supervisor

```
public class Person {
  // The type for the data is the class itself
    private Person  supervisor;
...
}
```

Aggregation Between Same Class

❖ What happens if a person has several supervisors?




```
public class Person {
  private Person[ ] supervisors;
  …
}
```

7

Example: The Course Class

Course
-courseName: String -students: String[] -numberOfStudents: int
+Course(courseName: String) +getCourseName(): String +addStudent(student: String): void +dropStudent(student: String): void +getStudents(): String[] +getNumberOfStudents(): int

The name of the course.
An array to store the students for the course.
The number of students (default: 0).

Creates a course with the specified name.
Returns the course name.
Adds a new student to the course.
Drops a student from the course.
Returns the students in the course.
Returns the number of students in the course

8

Designing a Class

- (**Coherence**) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
- You can use a class for **students**, for example, but you should not combine **students** and **staff** in the same class, because students and staff have different entities.

9

Designing a Class cont.

- (**Separating responsibilities**) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- Example: the classes **String**, **StringBuilder**, and **StringBuffer** all deal with strings, for example, but have different responsibilities:
  - **String** class deals with immutable strings.
  - **StringBuilder** class is for creating mutable strings.
  - **StringBuffer** class is similar to **StringBuilder** except that **StringBuffer** contains synchronized methods for updating strings.

10

Designing a Class cont.

❖ Classes are designed for **reuse**.

❖ Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it:

- Design the **properties** to ensure that the user can set properties in any order, with any combination of values.
- Design **methods** to function independently of their order of occurrence.

11

Designing a Class cont.

❖ **Follow standard Java programming style and naming conventions:**

- Choose **informative names** for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and **initialize** variables to avoid programming errors.

12

Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE:

(1) The wrapper classes **do not** have **no-arg** constructors.

(2) The instances of all wrapper classes are **immutable**, i.e., their internal values cannot be changed once the objects are created.

13

The Integer and Double Classes

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int

java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double

14

Numeric Wrapper Class Constructors

❖ You can construct a wrapper object either from a **primitive data type value** or from a **string** representing the numeric value.

❖ The constructors for **Integer** and **Double** are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

15

Numeric Wrapper Class Constants

❖ Each numerical wrapper class has the constants **MAX_VALUE** and **MIN_VALUE**.

❖ **MAX_VALUE** represents the maximum value of the corresponding primitive data type.

❖ For **Byte**, **Short**, **Integer**, and **Long**, **MIN_VALUE** represents the minimum **byte**, **short**, **int**, and **long** values.

❖ For **Float** and **Double**, **MIN_VALUE** represents the minimum *positive* **float** and **double** values.

16

Conversion Methods

❖ Each numeric wrapper class implements the abstract methods **doubleValue**, **floatValue**, **intValue**, **longValue**, and **shortValue**, which are defined in the **Number** class.

❖ These methods "**convert**" objects into primitive type values.

17

The Static valueOf Methods

❖ The numeric wrapper classes have a useful class method, **valueOf(String s)**.

❖ This method creates a new object initialized to the value represented by the specified string.

❖ For example:

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

18

The Methods for Parsing Strings into Numbers

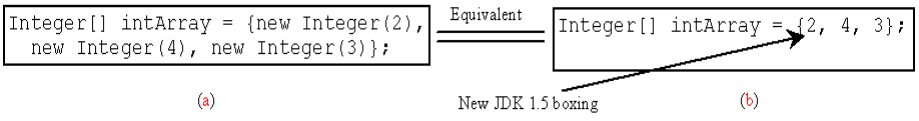
❖ You have used the **parseInt** method in the **Integer** class to parse a numeric string into an **int** value and the **parseDouble** method in the **Double** class to parse a numeric string into a **double** value.

❖ Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

19

Automatic Conversion Between Primitive Types and Wrapper Class Types

❖ **JDK 1.5** allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):




```
Integer[] arr = {1, 2, 3};

System.out.println(arr[0] + arr[1] + arr[2]);
```

Unboxing

BigInteger and BigDecimal

❖ If you need to compute with **very large integers** or **high precision floating-point** values, you can use the **BigInteger** and **BigDecimal** classes in the **java.math** package.

❖ Both are ***immutable***.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

BIRZEIT UNIVERSITY

Inheritance and Polymorphism

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Motivations

❖ Suppose you will define classes to model ***circles***, ***rectangles***, and ***triangles***.

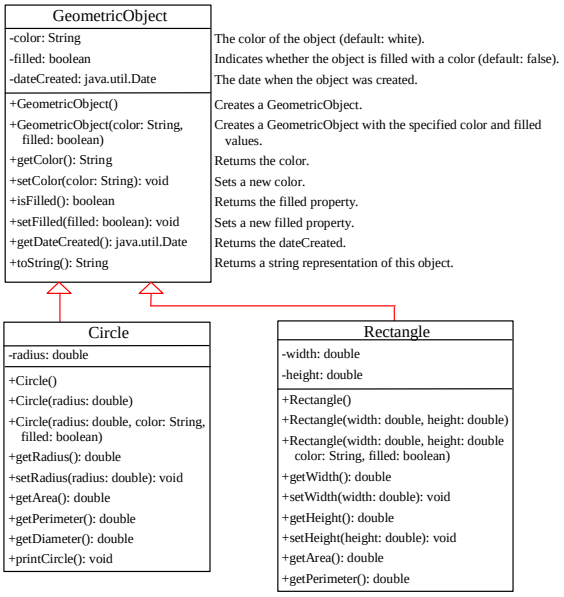
❖ These classes have **many common** features.

❖ What is the best way to design these classes so to **avoid redundancy**?

The answer is to use inheritance.

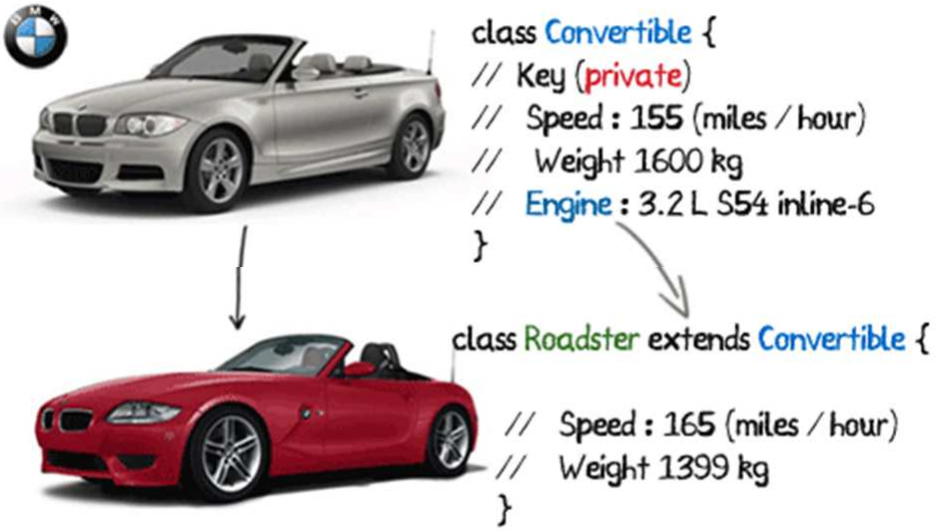
2

Superclasses and Subclasses



3

Superclass



Subclass

Are Superclass's Constructor Inherited?

❖ **No**. Unlike properties and methods, a superclass's **constructors are not inherited** in the subclass.

❖ They are invoked **explicitly** or **implicitly**.

❖ Explicitly using the **super** keyword.

❖ They can only be invoked from the subclasses' constructors, using the keyword **super**.

*If the keyword **super** is not **explicitly** used, the superclass's **no-arg constructor** is **automatically** invoked.*

5

Superclass's Constructor is Always Invoked

❖ A constructor may invoke an **overloaded** constructor **or** its superclass's constructor.

❖ **If** none of them is invoked explicitly, the compiler puts **super()** as the first statement in the constructor.

❖ For example:

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword super

❖ The keyword **super** refers to the superclass of the class in which super appears.

❖ **super** keyword can be used in two ways:

- **To call a superclass constructor.**
- **To call a superclass method.**

7

Caution

❖ You **must** use the keyword **super** to call the superclass constructor.

- Invoking a superclass constructor's name in a subclass causes a **syntax error**.

❖ Java requires that the statement that uses the keyword **super** appear **first** in the constructor.

8

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called ***constructor chaining***.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
     Faculty f = new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Super(); → (Faculty)

Super(); → (Employee)

Super(); → (Person)

Example on the Impact of a Superclass without no-arg Constructor

❖ Find out the **errors** in the following program:

```
public class Apple extends Fruit {
}

public class Fruit {
  public Fruit(String name) {
     System.out.println("Fruit's constructor is invoked");
  }
}
```

Defining a Subclass

❖ A **subclass** inherits from a superclass. You can also:

- ☞ **Add new properties.**
- ☞ **Add new methods.**
- ☞ **Override the methods of the superclass.**

11

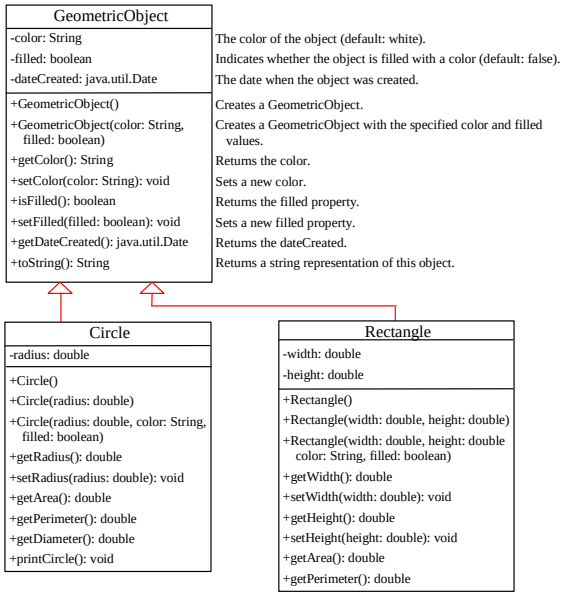
Calling Superclass Methods

❖ You could rewrite the **printCircle()** method in the **Circle** class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
                     super.getDateCreated() +
                     " and the radius is " + radius);
}
```

12

Superclasses and Subclasses



13

Overriding Methods in the Superclass

- Sometimes it is necessary for the subclass to **modify** the implementation of a method defined in the superclass.
- This is referred to as **method overriding**.

```
public class Circle extends GeometricObject {
  // Other methods are omitted
  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\n radius is " + radius;
  }
}
```

Note

❖ An **instance method** can be overridden **only if** it is accessible.

- Thus a **private method** cannot be overridden, because it is not accessible outside its own class.
- If a method defined in a subclass is **private** in its superclass, the two methods are completely unrelated.

15

Note cont.

❖ Like an instance method, a **static** method can be inherited.

- However, a **static** method **cannot** be overridden.
- If a **static** method defined in the superclass is redefined in a subclass, the method defined in the superclass is **hidden**.

16

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

17

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

18

The Object Class

❖ Every class in **Java** is descended from the **java.lang.Object** class.

❖ If no inheritance is specified when a class is defined, the superclass of the class is **Object**.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object{
  ...
}
```

19

The toString() method in Object

❖ The **toString()** method returns a string representation of the **object**.

❖ The default implementation returns a string consisting of:

- A **class name** of which the object is an instance.
- The at sign (**@**).
- A **number** representing this object.

20

The toString() method in Object

```
Circle c = new Circle();
System.out.println(c.toString());
```

- The code displays something like:

Circle@15037e5

- This message is not very helpful or informative.
- Usually you should **override** the **toString** method so that it returns an informative string representing the object.

21

```
class GraduateStudent extends Student {
}

class Student extends Person {
   public String toString() {
      return "Student";
   }
}

class Person extends Object {
   public String toString() {
      return "Person";
   }
}
```

Polymorphism

```
public class Demo {
  public static void main(String[] a) {
      m(new Object());
      m(new Person());
      m(new Student());
      m(new GraduateStudent());
  }
  public static void m(Object x){
      System.out.println(x.toString());
  }
}
```

Method **m** takes a parameter of the **Object** type.

You can invoke it with any object.

❖ An object of a **subtype** can be used wherever its **supertype** value is required.

❖ This feature is known as **polymorphism**.

Dynamic Binding

```
public class Demo {
  public static void main(String[] a) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
      System.out.println(x.toString());
  }
}
```

This capability is known as ***dynamic binding***.

❖ When the method **m(Object x)** is executed, the argument **x**'s **toString** method is invoked. **x** may be an instance of **GraduateStudent**, **Student**, **Person**, or **Object**.

❖ Classes **GraduateStudent**, **Student**, **Person**, and **Object** have their own implementation of the **toString** method. Which implementation is used will be determined **dynamically** by the JVM at **runtime**.

24

Dynamic Binding

- Dynamic binding works as follows:
  - Suppose an object **o** is an instance of classes $C_1, C_2, ..., C_{n-1}$, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
  - That is, C_n is the most general class, and C_1 is the most specific class.

Dynamic Binding cont.

- Dynamic binding works as follows:
  - If **o** invokes a method **p**, the **JVM** searches the implementation for the method **p** in $C_1, C_2, ..., C_{n-1}$ and C_n, in this order, until it is found.
  - Once an implementation is found, the search stops and the first-found implementation is invoked.

Generic Programming

```
public class Demo {
  public static void main(String[] a) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x){
      System.out.println(x.toString());
  }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments.

This is known as:

generic programming

- If a method's parameter type is a superclass (e.g., **Object**), you may pass an object to this method of any of the parameter's subclasses (e.g., **Student**).
- When an **object** (e.g., a **Student** object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., **toString**) is determined **dynamically**.

27

Casting Objects

- **Casting** can also be used to convert an object of one class type to another **within an inheritance hierarchy**.

m(new Student());

assigns the object **new Student()** to a parameter of the **Object** type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m( o );
```

The statement **Object o = new Student(),** known as **implicit casting,** is legal because an instance of **Student** is automatically an instance of **Object**.

28

Why Casting is Necessary?

❖ Suppose you want to assign the object reference **o** to a variable of the **Student** type using the following statement:

```
Student b = o ;   // A compile error would occur.
```

❖ Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?

- This is because a **Student** object is always an instance of **Object**, but an **Object** is not necessarily an instance of **Student**.
- Even though you can see that **o** is really a **Student** object, the compiler is not so clever to know it.

29

Why Casting Is Necessary?

❖ To tell the compiler that **o** is a **Student** object, use an **explicit casting**.

❖ The syntax is similar to the one used for casting among primitive data types.

❖ Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student) o ; // Explicit casting
```

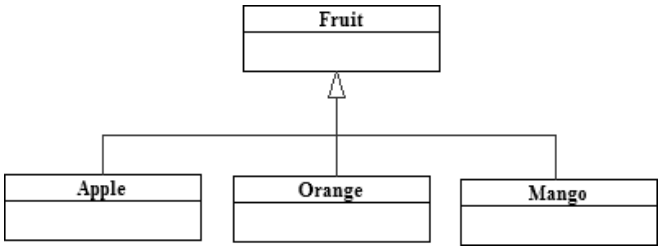
30

Casting from Superclass to Subclass

❖ Explicit casting **must** be used when casting an object from a superclass to a subclass.

```
Fruit fruit = new Apple();
Apple a = (Apple) fruit;
Orange o = (Orange) fruit;
```

❖ This type of casting **may not** always succeed.



31

The instanceof Operator

❖ Use the **instanceof** operator to test whether an object is an instance of a class:

```
Object  myObject = new Circle();
:
// Perform casting if myObject is an instance of Circle
if (myObject  instanceof  Circle) {
  System.out.println("The circle diameter is " +
    (  (Circle)myObject).getDiameter()   );
}
```

32

The equals Method

❖ The **equals()** method compares the contents of two objects.

❖ The default implementation of the **equals** method in the **Object** class is as follows:

```
public boolean equals (Object obj) {
    return ( this == obj );
}
```

❖ For example, the **equals** method is **overridden** in the **Circle** class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

Note

❖ The == comparison operator is used for comparing two **primitive data type** values or for determining whether two objects have the **same references**.

❖ The **equals** method is intended to test whether two objects have the **same contents**, provided that the method is modified in the defining class of the objects.

34

The ArrayList Class

- You can create an array to store objects.
- But the array's **size is fixed** once the array is created.
- Java provides the **ArrayList** class that can be used to store an **unlimited** number of objects.

35

The ArrayList Class

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	Sets the element at the specified index.

Generic Type <E>

❖ **ArrayList** is known as a generic class with a generic type **E**.

❖ You can specify a concrete type to replace **E** when creating an **ArrayList**.

❖ For example, the following statement creates an **ArrayList** and assigns its reference to variable **cities**. This **ArrayList** object can be used to store **strings:**

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

37

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

38

ArrayLists from/to Arrays

- Creating an **ArrayList** from an array of objects:

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new
    ArrayList<>(Arrays.asList(array));
```

- Creating an array of objects from an **ArrayList**:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

39

max and min in an ArrayList

```
java.util.Collections.max(list)
java.util.Collections.min(list)
```

Shuffling an ArrayList

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new
    ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

40

The protected Modifier

- The **protected** modifier can be applied on **data** and **methods** in a class.
- A **protected** data/method in a **public** class can be accessed by any class in the same package **or** its subclasses, **even if** the subclasses are in a different package.

Visibility increases

private, none (if no modifier is used), protected, public

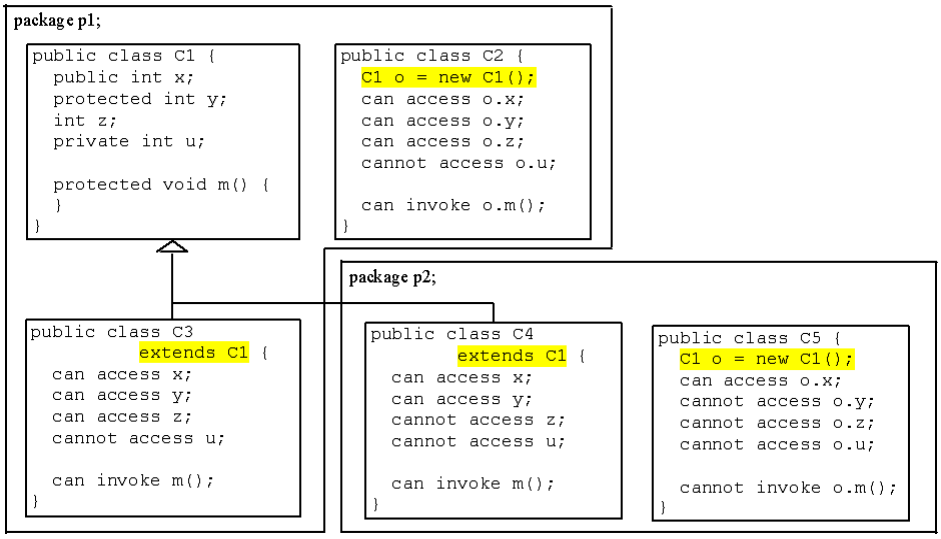
41

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

42

Visibility Modifiers



43

A Subclass Cannot Weaken the Accessibility

- A subclass may override a **protected** method in its superclass and change its visibility to **public**.
- However, a subclass **cannot weaken** the accessibility of a method defined in the superclass.
- For example, if a method is defined as **public** in the superclass, it must be defined as **public** in the subclass.

44

The final Modifier

- The **final class** cannot be extended:

```
final class Math {
    …
}
```

- The **final variable** is a **constant**:

```
final static double PI = 3.14159;
```

- The **final method** cannot be overridden by its subclasses.

45

Note

- The modifiers are used on classes and class **members** (data and methods), except that the **final** modifier can also be used on local variables in a method.
- A **final** local variable is a constant inside a method.

46

BIRZEIT UNIVERSITY

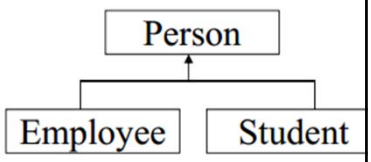
Abstract Classes and Interfaces

By: Mamoun Nawahdah (Ph.D.)
2015/2016

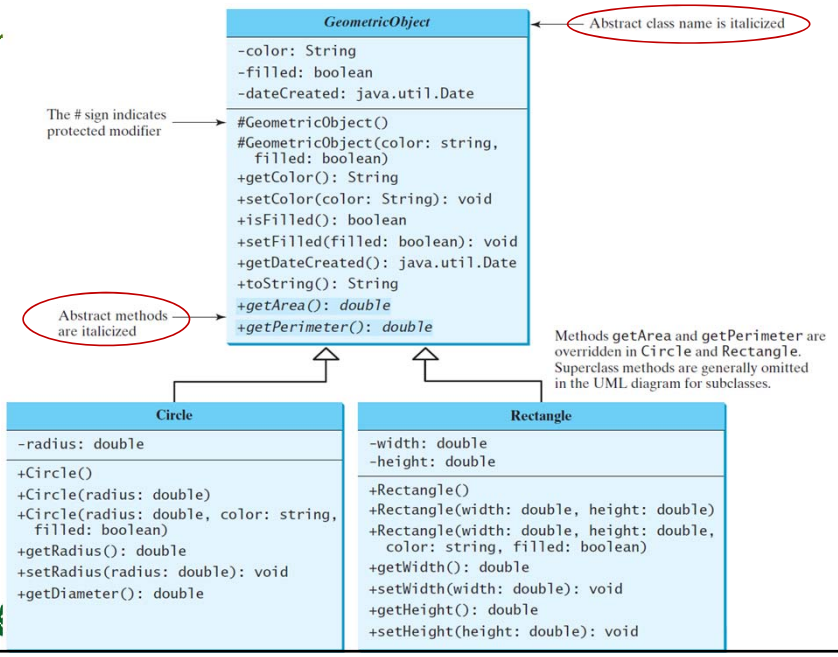
abstract Classes and Methods

- **Abstract classes**: some methods are **only** declared, but no **concrete** implementations are provided.
- Those methods called **abstract methods** and they need to be implemented by the extending classes.

```
abstract class Person {
  protected String name;
  . . .
  public abstract String getDescription() ;
  . . .
}
Class Student extends Person {
  private String major;
  . . .
  public String getDescription() {
        return "a student major in " + major;
  }
}
Class Employee extends Person {
  private float salary;
  . . .
  public String getDescription() {
        return "an employee with a salary of $ " + salary;
  }
}
```



abstract Classes and abstract Methods



abstract Method in abstract Class

❖ An **abstract** method **cannot** be contained in a non-abstract class.

❖ If a subclass of an **abstract** superclass does not implement all the **abstract** methods, the subclass **must** be defined **abstract**.

❖ In other words, in a nonabstract subclass extended from an abstract class, **all** the abstract methods **must** be implemented, **even if** they are not used in the subclass.

5

Object Can't be Created from abstract Class

❖ An **abstract** class **can't** be instantiated using the **new** operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.

❖ For instance, the constructors of **GeometricObject** are invoked in the **Circle** class and the **Rectangle** class.

6

Abstract Class without Abstract Method

❖ A class that contains **abstract** methods **must** be **abstract**.

❖ However, it is possible to define an **abstract** class that contains no **abstract** methods.

- ➢ In this case, you **cannot** create instances of the class using the **new** operator.
- ➢ This class is used as a **base** class for defining a new subclass.

7

Superclass of abstract Class may be Concrete

❖ A subclass can be **abstract** even if its superclass is **concrete**.

❖ For example, the **Object** class is concrete, but its subclasses, such as **GeometricObject**, may be **abstract**.

8

Concrete Method Overridden to be abstract

❖ A subclass can **override** a method from its superclass to define it **abstract**.

❖ This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined **abstract**.

9

abstract Class as Type

❖ You **can't** create an instance from an **abstract** class using the **new** operator, but an **abstract** class can be used as a data type.

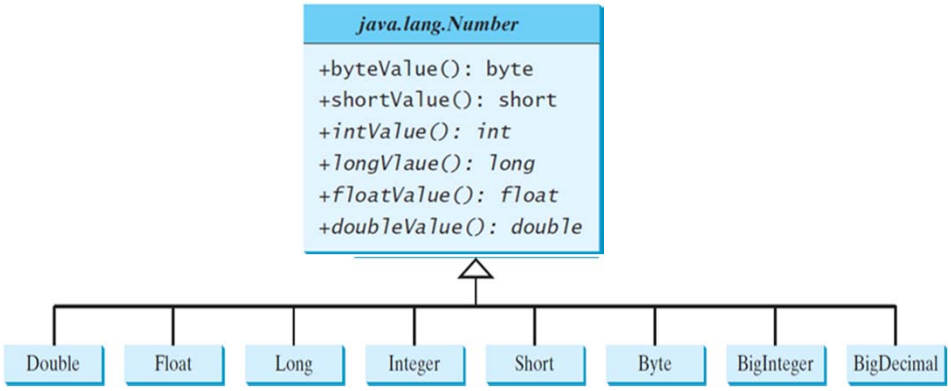
❖ Therefore, the following statement, which creates an array whose elements are of **GeometricObject** type, is correct:

```
GeometricObject[]  geo = new GeometricObject[10];
```

10

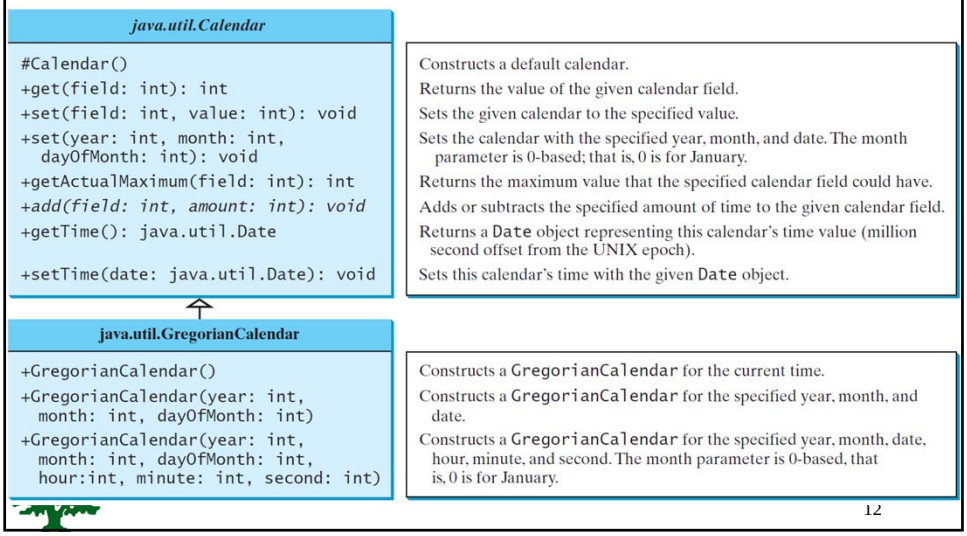
Case Study:

The Abstract Number Class



11

The Abstract Calendar Class and Its GregorianCalendar subclass



12

GregorianCalendar subclass

- An instance of **java.util.Date** represents a specific instant in time with millisecond precision.
- **java.util.Calendar** is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object.
- Subclasses of Calendar can implement specific calendar systems such as **Gregorian calendar**, **Lunar Calendar** and **Jewish calendar**.
- Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in the Java API.

13

The GregorianCalendar Class

- You can use **new GregorianCalendar()** to construct a default GregorianCalendar with the current time
- Use **new GregorianCalendar(year, month, date)** to construct a GregorianCalendar with the specified year, month, and date.
- The month parameter is **0-based**, i.e., 0 is for ***January***.

14

The get Method in Calendar Class

❖ The **get(int field)** method defined in the **Calendar** class is useful to extract the date and time information from a **Calendar** object. The fields are defined as constants, as shown in the following.

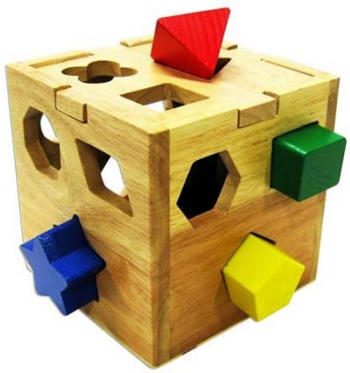
Constant	*Description*
YEAR	The year of the calendar.
MONTH	The month of the calendar, with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week, with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year, with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month, with 1 for the first week.
WEEK_OF_YEAR	The week number within the year, with 1 for the first week.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).

15

Interfaces

❖ An **interface** is a way to describe what classes should do, without specifying how they should do it.

❖ It is not a **class** but a set o **requirements** for classes that want to conform to the **interface**.



16

What is an interface?

❖ An **interface** is a **class-like** construct that contains **only constants** and **abstract** methods.

❖ In many ways, an **interface** is similar to an **abstract** class, but the intent of an interface is to specify **common behavior** for objects.

❖ For example, you can specify that the objects are ***comparable***, ***edible***, ***cloneable*** using appropriate interfaces.

17

Define an interface

❖ To distinguish an **interface** from a **class**, Java uses the following syntax to define an **interface**:

```
public interface InterfaceName {
    // constant declarations;
    // method signatures;
}
```

Example:

```
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```

18

Interface is a Special Class

❖ An **interface** is treated like a special class in **Java**.

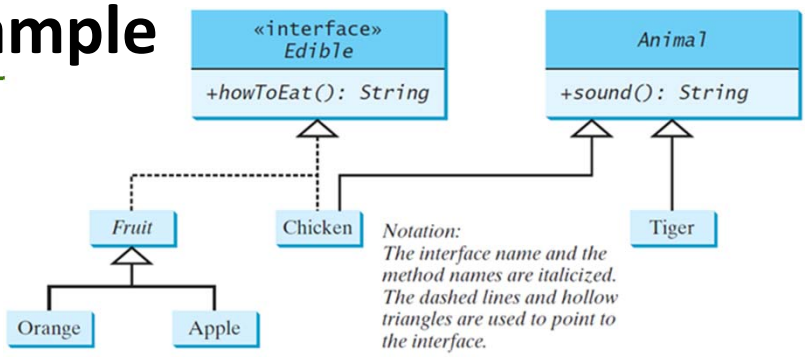
❖ Each **interface** is compiled into a separate **bytecode** file, just like a regular class.

❖ Like an **abstract** class, you **cannot** create an instance from an **interface** using the **new** operator, but in most cases you can use an **interface** more or less the same way you use an **abstract** class.

❖ For example, you can use an **interface** as a data type for variable, as the result of casting, and so on.

19

Example



❖ You can now use the **Edible** interface to specify whether an **object** is edible.

❖ This is accomplished by letting the class **implement** this interface using the **implements** keyword.

- For example, the classes **Chicken** and **Fruit** implement the **Edible** interface.

20

Omitting Modifiers in Interfaces

❖ All data fields are ***public final static*** and all methods are ***public abstract*** in an **interface**.

❖ For this reason, these modifiers can be **omitted**, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

❖ A constant defined in an **interface** can be accessed using syntax:

InterfaceName.CONSTANT_NAME

21

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

22

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

23

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

24

Examples

```
Integer i1 = new Integer(3),    i2 = new Integer(3);
System.out.println(i1.compareTo( i2 ));

System.out.println("ABC".compareTo("ABE"));

Date date1 = new Date(2013, 1, 1);
Date date2 = new Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

instanceof

❖ Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object.

❖ All the following expressions are **true:**

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

The toString, equals, and hashCode Methods

- Each wrapper class overrides the **toString**, **equals**, and **hashCode** methods defined in the **Object** class.
- Since all the numeric wrapper classes and the Character class implement the **Comparable** interface, the **compareTo** method is implemented in these classes.

27

Generic sort Method

java.util.Arrays.sort(array)

- This method requires that the elements in an array are instances of **Comparable<E>**.

28

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 – Constants

- If a **superinterface** and a **subinterface** contain two constants with the same name, then the one belonging to the superinterface is **hidden**:

```
interface X {
            int  val  = 1;
}
interface Y extends X {
            int  val  = 2;
            int sum = val  +  X.val;
}
```

Extending Interfaces – 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**.

The Cloneable Interfaces

- A class that implements the **Cloneable** interface is marked **cloneable**, and its objects can be cloned using the **clone()** method defined in the **Object** class.
- **clone** method returns a **new object** whose initial state is a **copy** of the current state of the object on which clone was invoked.
- Subsequent changes to the new clone object **should not** affect the state of the original object.

```
package java.lang;
public interface Cloneable {
}
```

32

Examples

❖ Many classes (e.g., **Date** and **Calendar**) implement **Cloneable**. Thus, the instances of these classes can be cloned. For example:

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
                                    (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
                                    calendar.equals(calendarCopy));
```

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

33

Implementing Cloneable Interface

❖ To define a custom class that implements the **Cloneable** interface, the class **must** override the **clone()** method in the **Object** class.

❖ The following code defines a class named **House** that implements **Cloneable** and **Comparable**.

34

```
public  class House implements Cloneable, Comparable<House> {
  private  int  id;
  private  double  area;
  private  java.util.Date   whenBuilt;

  public House(int id, double  area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId()      {      return id;     }

  public double getArea()      {    return area;  }

  public java.util.Date   getWhenBuilt()   {    return whenBuilt;    }
```

35

```
@Override // Override the clone method defined in the Object class
  public Object clone() {
        return  super.clone();
  }

  @Override // Implement the compareTo method defined in Comparable
  public int  compareTo(House  o) {
       if (area  >  o.area)
         return 1;
       else if (area < o.area)
         return -1;
       else
         return 0;
  }
}
```

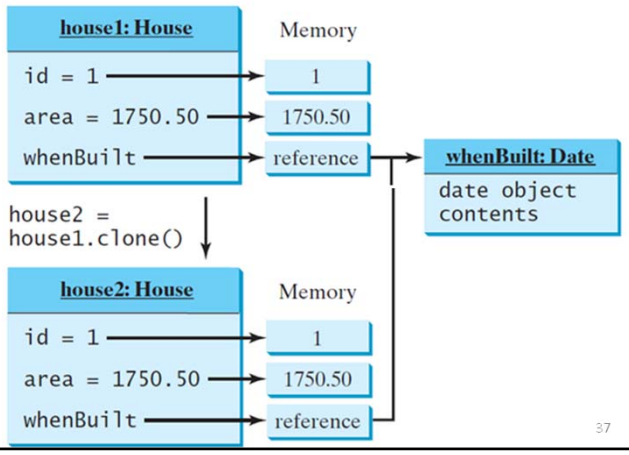
36

Shallow vs. Deep Copy

House **house1** = new House(1, 1750.50);

House **house2** = (House)house1.**clone**();

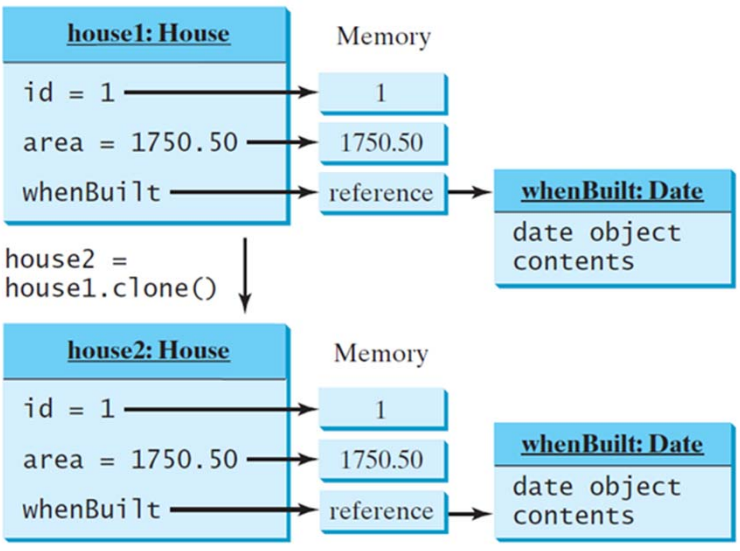
Shallow Copy



37

Shallow vs. Deep Copy

Deep Copy



38

Interfaces vs. Abstract Classes

❖ In an **interface**, the data must be **constants**; an **abstract** class can have all types of data.

❖ Each method in an **interface** has only a signature without implementation; an **abstract** class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class **cannot** be instantiated using the new operator.	No restrictions.
Interface	All variables must be **public static final**	No constructors. An interface **cannot** be instantiated using the new operator.	All methods **must** be public abstract instance methods

39

Interfaces vs. Abstract Classes cont.

❖ All classes share a single root, the **Object** class, but there is no single root for interfaces.

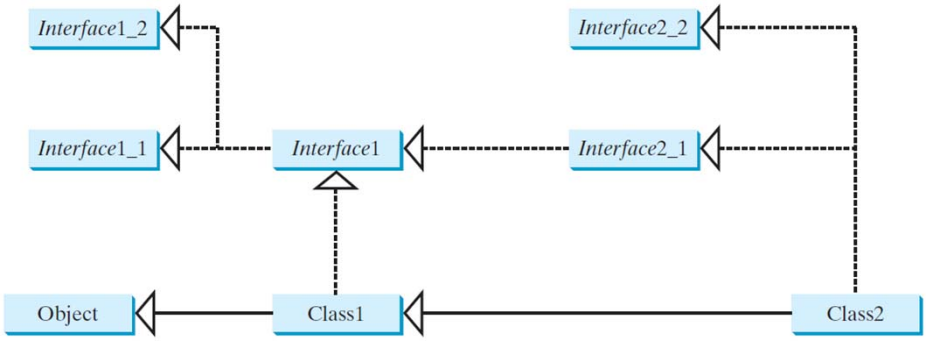
❖ Like a **class**, an **interface** also defines a type. A variable of an **interface** type can reference any instance of the class that implements the **interface**.

❖ If a **class** extends an **interface**, this interface plays the same role as a superclass.

❖ You can use an **interface** as a data type and cast a variable of an **interface** type to its subclass, and vice versa.

40

instanceof



❖ Suppose that **c** is an instance of **Class2**.

❖ **c** is also an instance of **Object**, **Class1**, **Interface1**, **Interface1_1**, **Interface1_2**, **Interface2_1**, and **Interface2_2**.

41

Caution: conflict interfaces

❖ In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). This type of errors will be detected by the compiler.

42

Whether to use an interface or a class?

- Abstract classes and interfaces can both be used to model common features.
- How do you decide whether to use an interface or a class?
- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.
- For example, a staff member is a person.

43

Whether to use an interface or a class?

- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property.
- A weak is-a relationship can be modeled using interfaces.
- For example, all strings are comparable, so the String class implements the Comparable interface.
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.
- In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

44

BIRZEIT UNIVERSITY

Exception Handling and Text IO

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Runtime Error?

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
      (number1 / number2));
  }
}
```

Fix it Using an if Statement

```
import java.util.Scanner;

public class QuotientWithIf {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    if (number2 != 0)
      System.out.println(number1 + " / " + number2 + " is " +
        (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

3

Exception Handling

❖ Exception handling technique enables a method to **throw** an exception to its caller.

❖ Without this capability, a method must handle the exception or terminate the program.

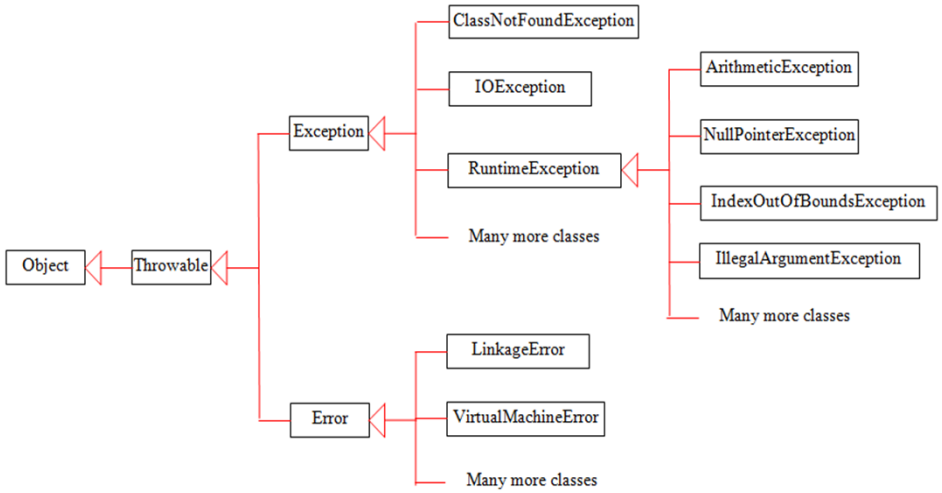
ex·cep·tion *noun* \ik-'sep-shən\

: someone or something that is different from others : someone or something that is not included

: a case where a rule does not apply

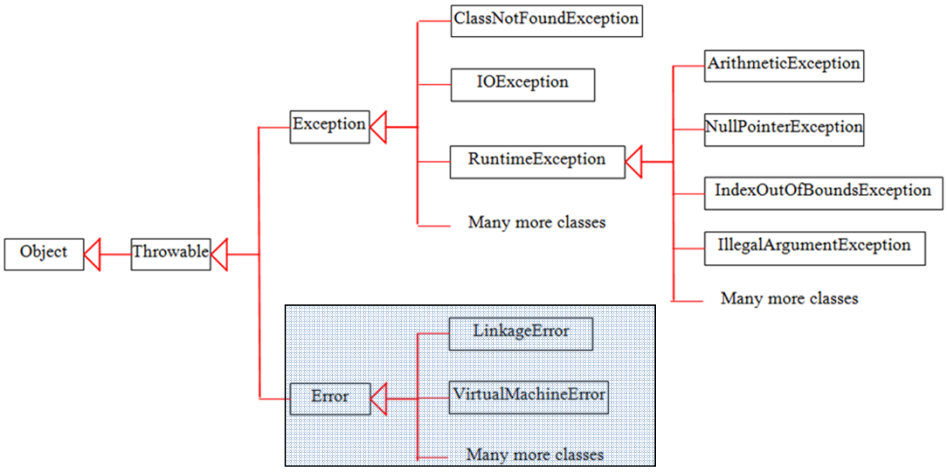
4

Exception Types



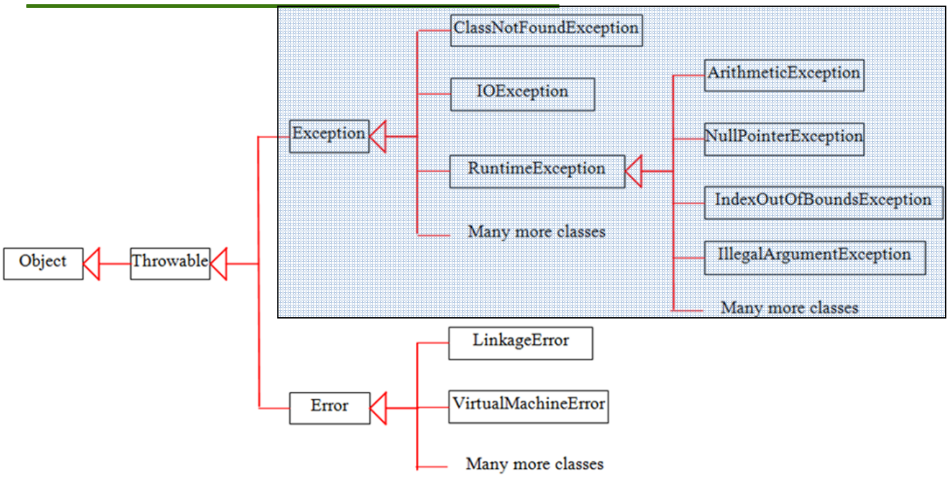
5

System Errors



System errors are thrown by **JVM** and represented in the **Error** class. The Error class describes internal system errors.

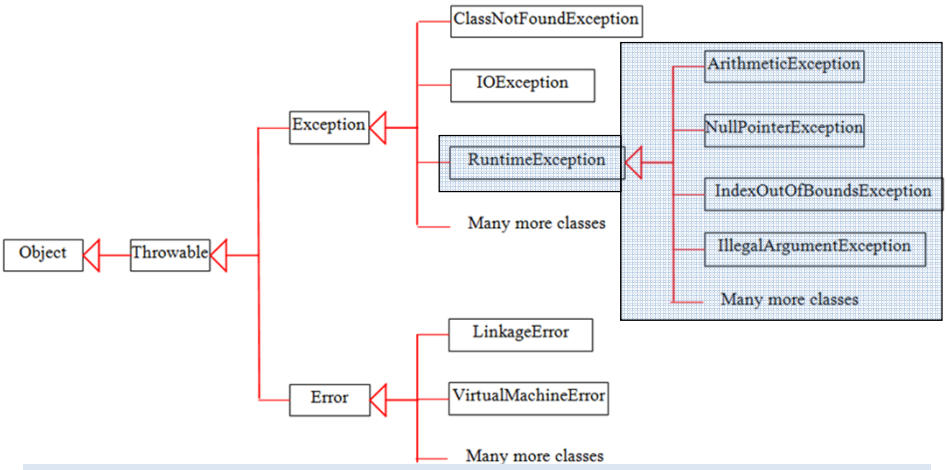
Exceptions



- **Exception** describes errors caused by **your program** and external circumstances.
- These errors can be caught and handled by your program.

7

Runtime Exceptions



- **RuntimeException** is caused by **programming errors**, such as bad casting, accessing an out-of-bounds array, and numeric errors.

8

Checked Exceptions vs. Unchecked Exceptions

❖ **RuntimeException**, **Error** and their subclasses are known as **unchecked exceptions**.

❖ All other exceptions are known as **checked exceptions**, meaning that the compiler forces the programmer to check and deal with the exceptions.

9

Unchecked Exceptions

❖ In most cases, unchecked exceptions reflect programming **logic errors** that are not recoverable.

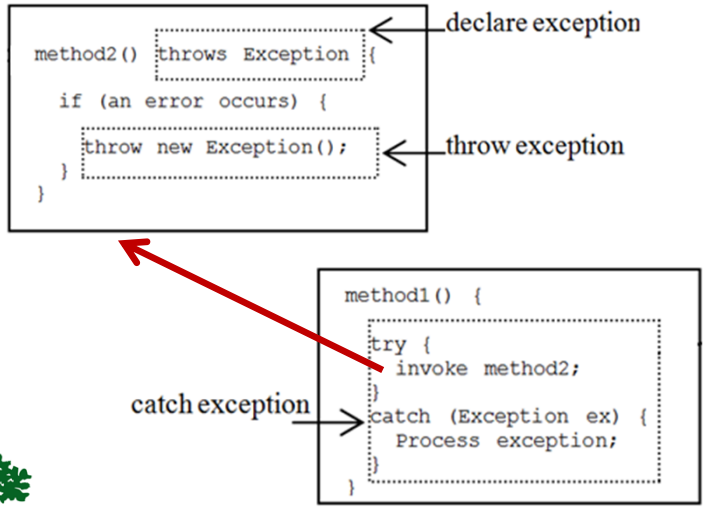
❖ For example:

- a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it.
- an **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array.

❖ These are the logic errors that should be corrected in the program.

10

Declaring, Throwing, and Catching Exceptions



11

Declaring Exceptions

❖ Every method **must** state the types of checked exceptions it might **throw**.

❖ This is known as **declaring exceptions.**

```
public void x()  throws IOException

public void y()  throws IOException, OtherException
```

12

Throwing Exceptions

- When the program detects an error, the program can create an **instance** of an appropriate exception type and throw it.
- This is known as **throwing an exception**.

```
throw new TheException();

TheException   ex = new TheException();
throw ex;
```

Throwing Exceptions Example

```
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
     radius =  newRadius;
  else
     throw new IllegalArgumentException(
              "Radius cannot be negative");
 }
```

Catching Exceptions

```
try {
    statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

15

Catch or Declare Checked Exceptions

❖ Java forces you to deal with checked exceptions.

❖ You must invoke it in a **try-catch** block **or** declare to **throw** the exception in the calling method.

❖ For example, suppose that method **p1** invokes method **p2** and **p2** may throw a checked exception (e.g., **IOException**), you have to write the code as follow:

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

```
1  public class CircleWithException {
2    /** The radius of the circle */
3    private double radius;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithException() {
10     this(1.0);
11   }
12
13   /** Construct a circle with a specified radius */
14   public CircleWithException(double newRadius) {
15     setRadius(newRadius);
16     numberOfObjects++;
17   }
18
19   /** Return radius */
20   public double getRadius() {
21     return radius;
22   }
```

throws IllegalArgumentException

```
24   /** Set a new radius */
25   public void setRadius(double newRadius)
26       throws IllegalArgumentException {
27     if (newRadius >= 0)
28       radius =  newRadius;
29     else
30       throw new IllegalArgumentException(
31         "Radius cannot be negative");
32   }
33
34   /** Return numberOfObjects */
35   public static int getNumberOfObjects() {
36     return numberOfObjects;
37   }
38
39   /** Return the area of this circle */
40   public double findArea() {
41     return radius * radius * 3.14159;
42   }
43 }
```

```
1  public class TestCircleWithException {
2    public static void main(String[] args) {
3      try {
4        CircleWithException c1 = new CircleWithException(5);
5        CircleWithException c2 = new CircleWithException(-5);
6        CircleWithException c3 = new CircleWithException(0);
7      }
8      catch (IllegalArgumentException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithException.getNumberOfObjects());
14   }
15 }
```

Rethrowing Exceptions

```
try {
    statements;
}
catch(TheException ex) {
    perform operations before exits;
    throw ex;
}
```

20

The finally Clause

```
try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
```

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

25

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

26

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

27

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

28

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

29

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

30

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

31

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

32

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs **easier** to read and to modify.
- Be aware, however, that exception handling usually requires **more time and resources** because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

33

When to Throw Exceptions

- An exception occurs in a method.
- If you want the exception to be processed by its caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw it.

34

When to Use Exceptions

❖ You should use it to deal with **unexpected** error conditions.

❖ Do not use it to deal with simple, expected situations. For example, the following code:

```
try {
    System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
    System.out.println("refVar is null");
}
```

35

When to Use Exceptions

❖ is better to be replaced by:

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
```

36

Defining Custom Exception Classes

- Use the exception classes in the **API** whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

37

Custom Exception Class Example

```
1  public class InvalidRadiusException extends Exception {
2    private double radius;
3
4    /** Construct an exception */
5    public InvalidRadiusException(double radius) {
6      super("Invalid radius " + radius);
7      this.radius = radius;
8    }
9
10   /** Return the radius */
11   public double getRadius() {
12     return radius;
13   }
14 }
```

```
/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new InvalidRadiusException(newRadius);
}
```

38

The File Class

- The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- The filename is a string.
- The **File** class is a wrapper class for the file name and its directory path.

39

File class

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.

40

File class

`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object. The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

41

Text I/O

- A **File** object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes.
- The objects contain the methods for reading/writing data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes.

42

PrintWriter class

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.6, "Formatting Console Output and Strings."

43

Scanner class

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

44

BIRZEIT UNIVERSITY

JavaFX Basics

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Motivations

- **JavaFX** is a new framework for developing Java graphical user interface (**GUI)** programs.
- The **JavaFX API** is an **excellent** example of how the **OO** principle is applied.
- This chapter serves two purposes:
  - First, it presents the basics of **JavaFX** programming.
  - Second, it uses **JavaFX** to demonstrate **OOP**.
- Specifically, this chapter introduces the framework of **JavaFX** and discusses **JavaFX GUI** components and their relationships.

2

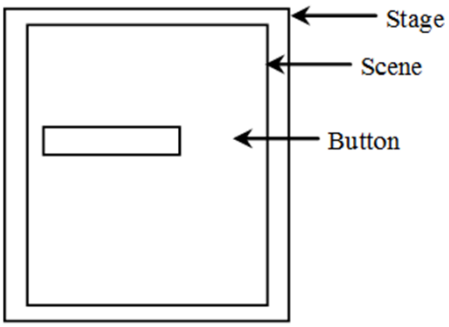
JavaFX vs Swing and AWT

- When Java was introduced, the **GUI** classes were bundled in a library known as the ***Abstract Windows Toolkit (AWT)***.
  - **AWT** is fine for developing simple graphical user interfaces, but not for developing comprehensive **GUI**.
  - In addition, **AWT** is prone to platform-specific bugs.
- The **AWT** components were replaced by a more robust, versatile, and flexible library known as ***Swing***.
  - **Swing** components depend less on the target platform and use less of the native **GUI** resource.
- With the release of Java 8, **Swing** is replaced by a completely new **GUI** platform known as ***JavaFX***.

3

Basic Structure of JavaFX

- Application
- Override the start (**Stage**) method
- **Stage**, **Scene**, and **Nodes**



4

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MyJavaFX extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a button and place it in the scene
    Button btOK = new Button("OK");
    Scene scene = new Scene(btOK, 200, 250);
    primaryStage.setTitle("MyJavaFX"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE
   * JavaFX support. Not needed for running fro
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

Extend Application

Override start

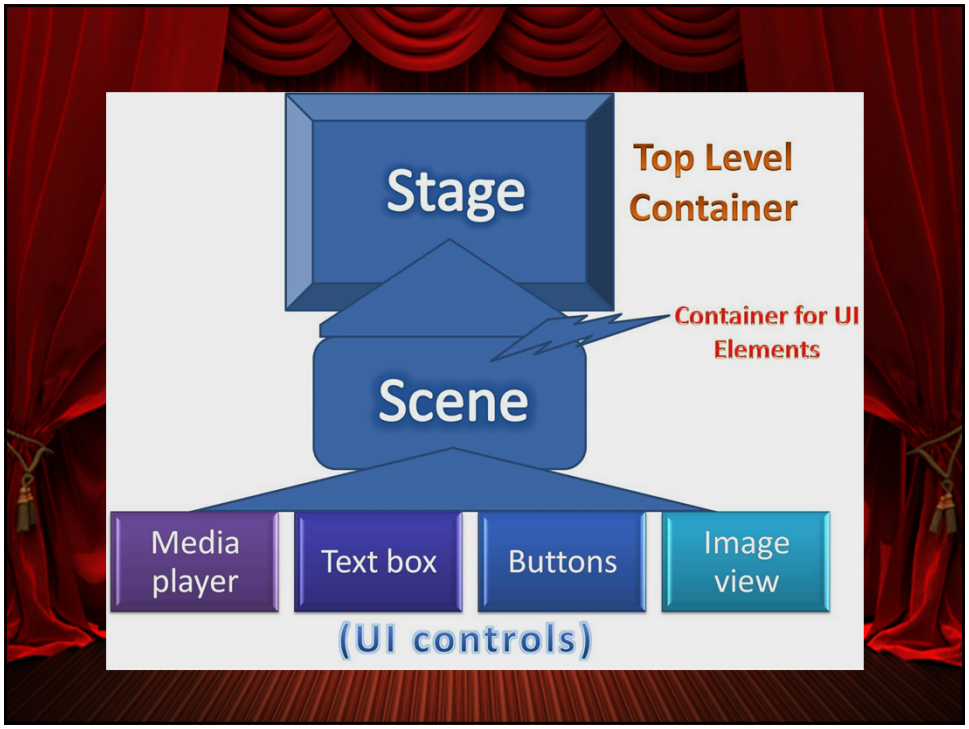
Create a button

Create a scene

Set a scene

Display stage

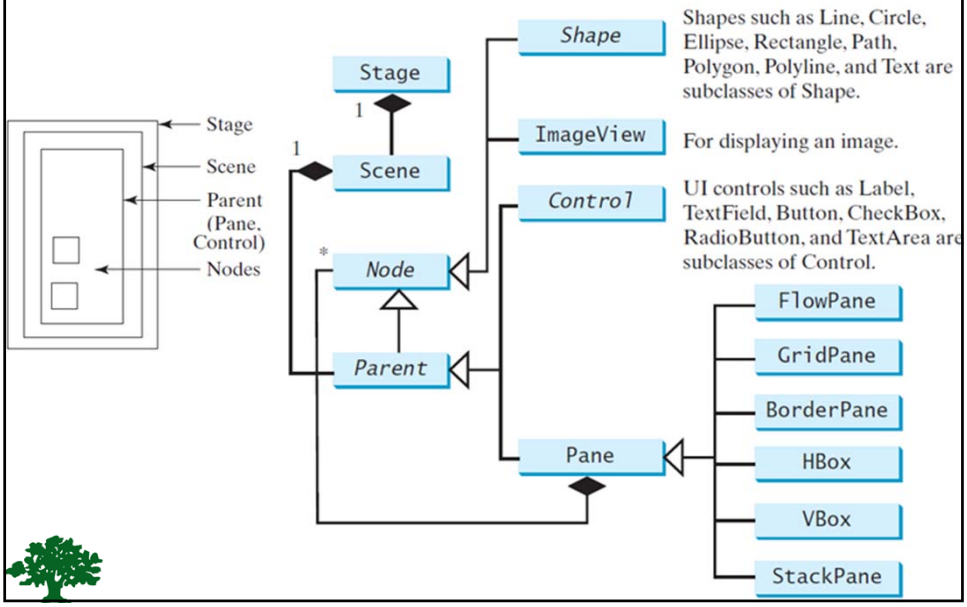
Launch application



```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MultipleStageDemo extends Application {
  public void start(Stage primaryStage) {
    Scene scene = new Scene(new Button("OK"), 200, 250);
    primaryStage.setTitle("MyJavaFX");
    primaryStage.setScene(scene);
    primaryStage.show();
    // Create a new stage
    Stage stage = new Stage();
    stage.setTitle("Second Stage");
    stage.setScene(new Scene(new Button("New Stage"), 100, 100));
    stage.show(); // Display the stage
  }

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

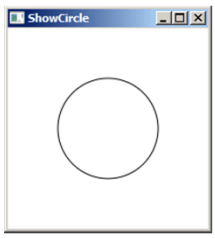
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ButtonInPane extends Application {
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    pane.getChildren().add(new Button("OK"));
    Scene scene = new Scene(pane, 200, 50);
    primaryStage.setTitle("Button in a pane");
    primaryStage.setScene(scene);
    primaryStage.show();
  }

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

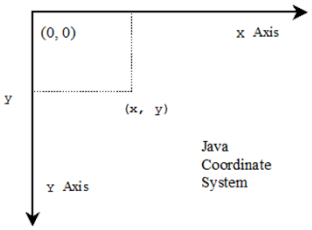
Panes, UI Controls, and Shapes



Display a Shape

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ShowCircle extends Application {
  public void start(Stage primaryStage) {
    Circle circle = new Circle();
    circle.setCenterX(100);
    circle.setCenterY(100);
    circle.setRadius(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(null);

    Pane pane = new Pane();
    pane.getChildren().add(circle);

    Scene scene = new Scene(pane, 200, 200);
    primaryStage.setTitle("ShowCircle");
    primaryStage.setScene(scene);
    primaryStage.show();
  }
```

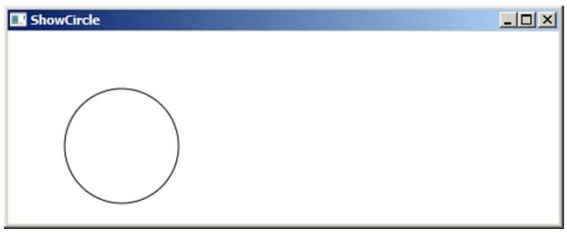




10

Binding Properties

- **JavaFX** introduces a new concept called ***binding property*** that enables a ***target object*** to be bound to a ***source object***.
  - If the value in the source object changes, the target property is also changed automatically.
- The target object is simply called a ***binding object*** or a ***binding property***.



11

Binding Properties

```
public void start(Stage primaryStage) {
  Pane pane = new Pane();
  Circle circle = new Circle();
  circle.centerXProperty().bind(pane.widthProperty().divide(2));
  circle.centerYProperty().bind(pane.heightProperty().divide(2));
  circle.setRadius(50);
  circle.setStroke(Color.BLACK);
  circle.setFill(Color.WHITE);
  pane.getChildren().add(circle);

  Scene scene = new Scene(pane, 200, 200);
  primaryStage.setTitle("ShowCircleCentered");
  primaryStage.setScene(scene);
  primaryStage.show();
}
```

12

Binding Property:

getter, setter, and property getter

```
public class SomeClassName {

  private PropertyType x;

  /** Value getter method */
  public propertyValueType getX() { ... }

  /** Value setter method */
  public void setX(propertyValueType value) { ... }

  /** Property getter method */
  public PropertyType
    xProperty() { ... }
}
```

(a) x is a binding property

```
public class Circle {

  private DoubleProperty centerX;

  /** Value getter method */
  public double getCenterX() { ... }

  /** Value setter method */
  public void setCenterX(double value) { ... }

  /** Property getter method */
  public DoubleProperty centerXProperty() { ... }
}
```

(b) centerX is binding property

13

Binding Property

- **JavaFX** defines binding properties for primitive types and strings.
- For a **double/float/long/int/boolean** value, its binding property type is **DoubleProperty/ FloatProperty/ LongProperty/ IntegerProperty/ BooleanProperty**.
- For a **String**, its binding property type is **StringProperty**.

14

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class BindingDemo {
  public static void main(String[] args) {
    DoubleProperty d1 = new SimpleDoubleProperty(1);
    DoubleProperty d2 = new SimpleDoubleProperty(2);
    d1.bind(d2);
    System.out.println("d1 is " + d1.getValue()
      + " and d2 is " + d2.getValue());
    d2.setValue(70.2);
    System.out.println("d1 is " + d1.getValue()
      + " and d2 is " + d2.getValue());
  }
}
```

```
d1 is 2.0 and d2 is 2.0
d1 is 70.2 and d2 is 70.2
```

Common Properties and Methods for Nodes

- The abstract **Node** class defines many properties and methods that are common to all nodes.
  - **style**: set a **JavaFX CSS** style

```
circle.setStyle("-fx-stroke: black; -fx-fill: red;");
```

  - **rotate**: Rotate a node

```
button.setRotate(80);
```

16

The Color Class

javafx.scene.paint.Color	
-red: double	The red value of this Color (between 0.0 and 1.0).
-green: double	The green value of this Color (between 0.0 and 1.0).
-blue: double	The blue value of this Color (between 0.0 and 1.0).
-opacity: double	The opacity of this Color (between 0.0 and 1.0).
+Color(r: double, g: double, b: double, opacity: double)	Creates a Color with the specified red, green, blue, and opacity values.
+brighter(): Color	Creates a Color that is a brighter version of this Color.
+darker(): Color	Creates a Color that is a darker version of this Color.
+color(r: double, g: double, b: double): Color	Creates an opaque Color with the specified red, green, and blue values.
+color(r: double, g: double, b: double, opacity: double): Color	Creates a Color with the specified red, green, blue, and opacity values.
+rgb(r: int, g: int, b: int): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255.
+rgb(r: int, g: int, b: int, opacity: double): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

```
Color color = new Color(0.25, 0.14, 0.333, 0.51);
```

17

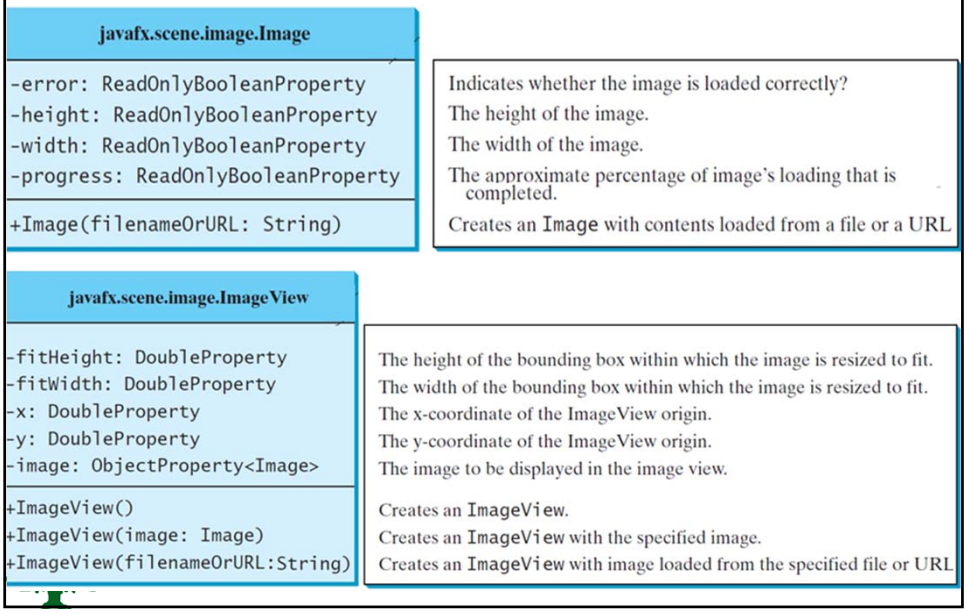
The Font Class

javafx.scene.text.Font	
-size: double	The size of this font.
-name: String	The name of this font.
-family: String	The family of this font.
+Font(size: double)	Creates a Font with the specified size.
+Font(name: String, size: double)	Creates a Font with the specified full font name and size.
+font(name: String, size: double)	Creates a Font with the specified name and size.
+font(name: String, w: FontWeight, size: double)	Creates a Font with the specified name, weight, and size.
+font(name: String, w: FontWeight, p: FontPosture, size: double)	Creates a Font with the specified name, weight, posture, and size.
+getFamilies(): List<String>	Returns a list of font family names.
+getFontNames(): List<String>	Returns a list of full font names including family and weight.

```
Font font1 = new Font("SansSerif", 16);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
  FontPosture.ITALIC, 12);
```

18

The Image, ImageView Class



Layout Panes

❖ **JavaFX** provides many types of panes for organizing nodes in a container.

Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

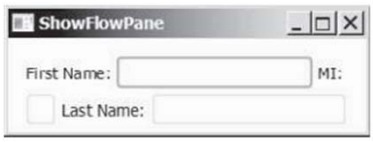
20

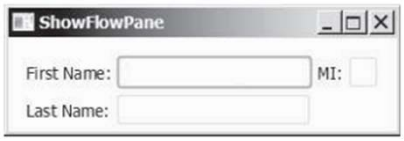
FlowPane

javafx.scene.layout.FlowPane

```
-alignment: ObjectProperty<Pos>
-orientation:
   ObjectProperty<Orientation>
-hgap: DoubleProperty
-vgap: DoubleProperty
+FlowPane()
+FlowPane(hgap: double, vgap:
   double)
+FlowPane(orientation:
   ObjectProperty<Orientation>)
+FlowPane(orientation:
   ObjectProperty<Orientation>,
   hgap: double, vgap: double
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).	
The orientation in this pane (default: `Orientation.HORIZONTAL`).	
The horizontal gap between the nodes (default: 0).	
The vertical gap between the nodes (default: 0).	
Creates a default `FlowPane`.	
Creates a `FlowPane` with a specified horizontal and vertical gap.	
Creates a `FlowPane` with a specified orientation.	
Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.	





21

GridPane

javafx.scene.layout.GridPane

```
-alignment: ObjectProperty<Pos>
-gridLinesVisible:
   BooleanProperty
-hgap: DoubleProperty
-vgap: DoubleProperty
+GridPane()
+add(child: Node, columnIndex:
   int, rowIndex: int): void
+addColumn(columnIndex: int,
   children: Node...): void
+addRow(rowIndex: int,
   children: Node...): void
+getColumnIndex(child: Node):
   int
+setColumnIndex(child: Node,
   columnIndex: int): void
+getRowIndex(child:Node): int
+setRowIndex(child: Node,
   rowIndex: int): void
+setHalighnment(child: Node,
   value: HPos): void
+setValighnment(child: Node,
   value: VPos): void
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).	
Is the grid line visible? (default: `false`)	
The horizontal gap between the nodes (default: 0).	
The vertical gap between the nodes (default: 0).	
Creates a `GridPane`.	
Adds a node to the specified column and row.	
Adds multiple nodes to the specified column.	
Adds multiple nodes to the specified row.	
Returns the column index for the specified node.	
Sets a node to a new column. This method repositions the node.	
Returns the row index for the specified node.	
Sets a node to a new row. This method repositions the node.	
Sets the horizontal alignment for the child in the cell.	
Sets the vertical alignment for the child in the cell.	

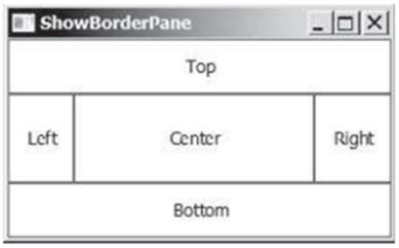
BorderPane

javafx.scene.layout.BorderPane

```
-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>
+BorderPane()
+setAlignment(child: Node, pos: Pos)
```

The node placed in the top region (default: `null`).
The node placed in the right region (default: `null`).
The node placed in the bottom region (default: `null`).
The node placed in the left region (default: `null`).
The node placed in the center region (default: `null`).

Creates a `BorderPane`.
Sets the alignment of the node in the `BorderPane`.



23

Hbox , VBox

javafx.scene.layout.HBox

```
-alignment: ObjectProperty<Pos>
-fillHeight: BooleanProperty
-spacing: DoubleProperty
+HBox()
+HBox(spacing: double)
+setMargin(node: Node, value: Insets): void
```

The overall alignment of the children in the box (default: `Pos.TOP_LEFT`).
Is resizable children fill the full height of the box (default: `true`).
The horizontal gap between two nodes (default: `0`).

Creates a default `HBox`.
Creates an `HBox` with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

javafx.scene.layout.VBox

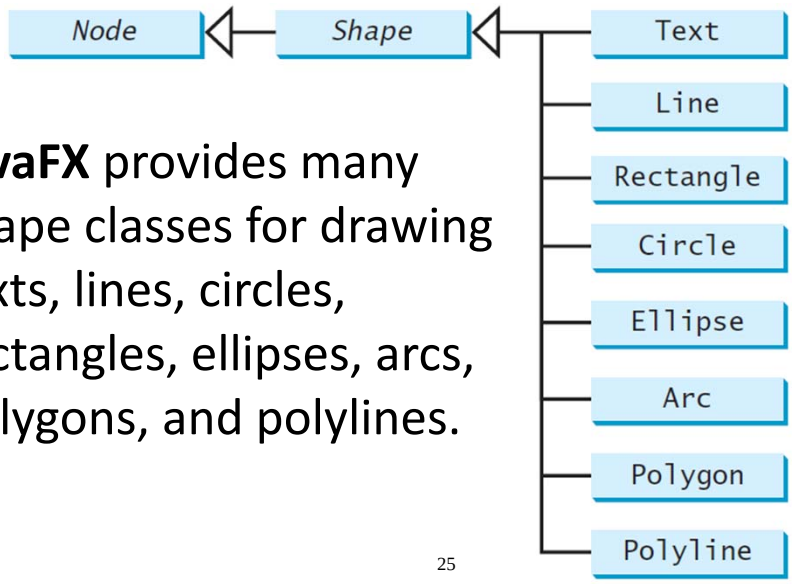
```
-alignment: ObjectProperty<Pos>
-fillWidth: BooleanProperty
-spacing: DoubleProperty
+VBox()
+VBox(spacing: double)
+setMargin(node: Node, value: Insets): void
```

The overall alignment of the children in the box (default: `Pos.TOP_LEFT`).
Is resizable children fill the full width of the box (default: `true`).
The vertical gap between two nodes (default: `0`).

Creates a default `VBox`.
Creates a `VBox` with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

24

Shapes



- **JavaFX** provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.

25

Text

javafx.scene.text.Text	
-text: StringProperty	Defines the text to be displayed.
-x: DoubleProperty	Defines the x-coordinate of text (default 0).
-y: DoubleProperty	Defines the y-coordinate of text (default 0).
-underline: BooleanProperty	Defines if each line has an underline below it (default `false`).
-strikethrough: BooleanProperty	Defines if each line has a line through it (default `false`).
-font: ObjectProperty<Font>	Defines the font for the text.
+Text()	Creates an empty Text.
+Text(text: String)	Creates a Text with the specified text.
+Text(x: double, y: double, text: String)	Creates a Text with the specified x-, y-coordinates and text.

26

Line

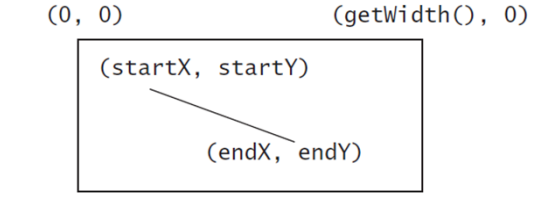
javafx.scene.shape.Line

```
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
```

```
+Line()
+Line(startX: double, startY:
   double, endX: double, endY:
   double)
```

The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.

Creates an empty `Line`.
Creates a `Line` with the specified starting and ending points.



27

Rectangle

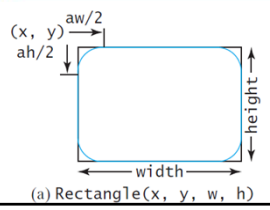
javafx.scene.shape.Rectangle

```
-x: DoubleProperty
-y:DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty

-arcHeight: DoubleProperty
```

```
+Rectangle()
+Rectanlge(x: double, y:
   double, width: double,
   height: double)
```

The x-coordinate of the upper-left corner of the rectangle (default 0).
The y-coordinate of the upper-left corner of the rectangle (default 0).
The width of the rectangle (default: 0).
The height of the rectangle (default: 0).
The `arcWidth` of the rectangle (default: 0). `arcWidth` is the horizontal diameter of the arcs at the corner (see Figure 14.31a).
The `arcHeight` of the rectangle (default: 0). `arcHeight` is the vertical diameter of the arcs at the corner (see Figure 14.31a).

Creates an empty `Rectangle`.
Creates a `Rectangle` with the specified upper-left corner point, width, and height.



(a) `Rectangle(x, y, w, h)`

28

Circle, Ellipse

javafx.scene.shape.Circle

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radius: DoubleProperty
+Circle()
+Circle(x: double, y: double)
+Circle(x: double, y: double,
   radius: double)
```

The x-coordinate of the center of the circle (default 0).
The y-coordinate of the center of the circle (default 0).
The radius of the circle (default: 0).

Creates an empty `Circle`.
Creates a `Circle` with the specified center.
Creates a `Circle` with the specified center and radius.

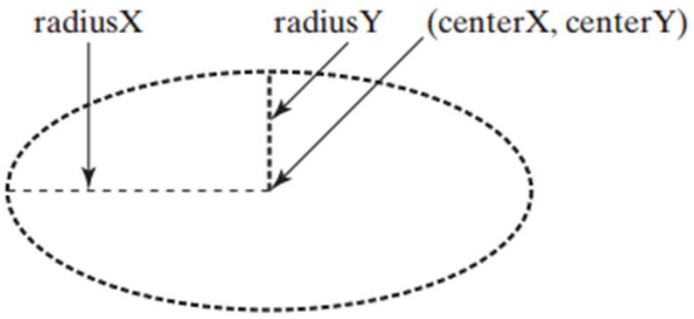
javafx.scene.shape.Ellipse

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
+Ellipse()
+Ellipse(x: double, y: double)
+Ellipse(x: double, y: double,
   radiusX: double, radiusY:
   double)
```

The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).

Creates an empty `Ellipse`.
Creates an `Ellipse` with the specified center.
Creates an `Ellipse` with the specified center and radiuses.

Ellipse

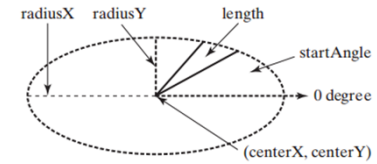


(a) `Ellipse(centerX, centerY, radiusX, radiusY)`

30

Arc

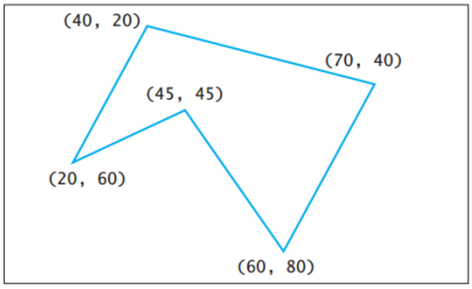
javafx.scene.shape.Arc	
-centerX: DoubleProperty	The x-coordinate of the center of the ellipse (default 0).
-centerY: DoubleProperty	The y-coordinate of the center of the ellipse (default 0).
-radiusX: DoubleProperty	The horizontal radius of the ellipse (default: 0).
-radiusY: DoubleProperty	The vertical radius of the ellipse (default: 0).
-startAngle: DoubleProperty	The start angle of the arc in degrees.
-length: DoubleProperty	The angular extent of the arc in degrees.
-type: ObjectProperty<ArcType>	The closure type of the arc (ArcType.OPEN, ArcType.CHORD, ArcType.ROUND).
+Arc()	Creates an empty Arc.
+Arc(x: double, y: double, radiusX: double, radiusY: double, startAngle: double, length: double)	Creates an Arc with the specified arguments.



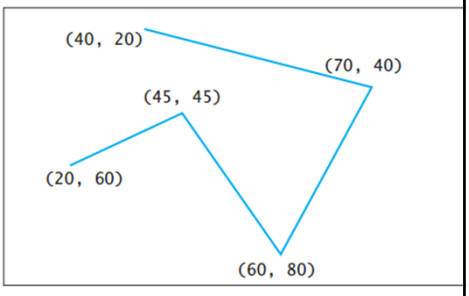
(a) Arc(centerX, centerY, radiusX, radiusY, startAngle, length)

31

Polygon and Polyline



(a) Polygon



(b) Polyline

javafx.scene.shape.Polygon	
+Polygon()	Creates an empty polygon.
+Polygon(double... points)	Creates a polygon with the given points.
+getPoints(): ObservableList<Double>	Returns a list of double values as x- and y-coordinates of the points.

32

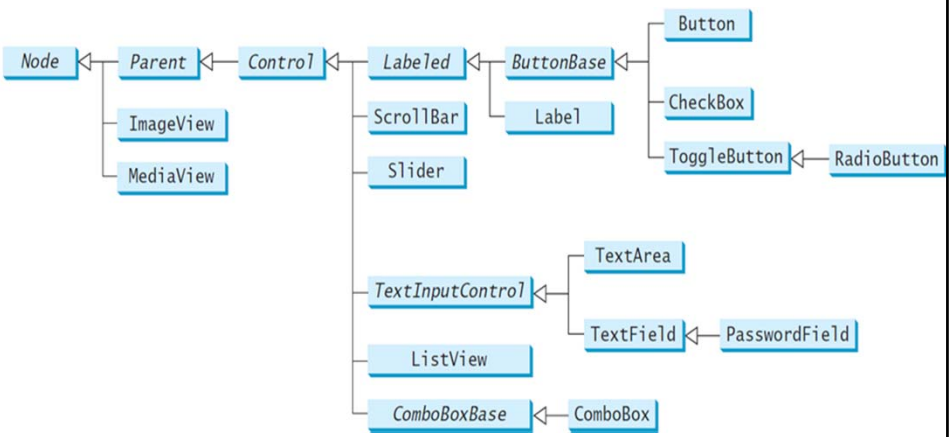
BIRZEIT UNIVERSITY

JavaFX UI Controls

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All

By: Mamoun Nawahdah (Ph.D.)
2015/2016

Frequently Used UI Controls



- Throughout this book, the prefixes **lbl**, **bt**, **chk**, **rb**, **tf**, **pf**, **ta**, **cbo**, **lv**, **scb**, **sld**, and **mp** are used to name reference variables for **Label**, **Button**, **CheckBox**, **RadioButton**, **TextField**, **PasswordField**, **TextArea**, **ComboBox**, **ListView**, **ScrollBar**, **Slider**, and **MediaPlayer**.

2

Labeled

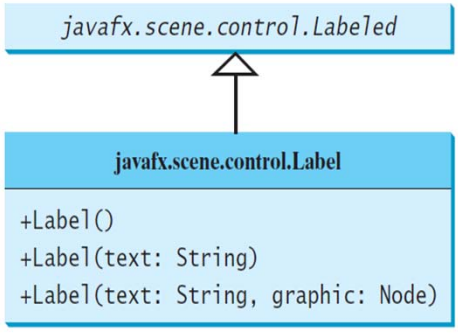
- A ***label*** is a display area for a short text, a node, or both.
- It is often used to label other controls (usually text fields).
- Labels and buttons share many common properties. These common properties are defined in the **Labeled** class.

javafx.scene.control.Labeled	
-alignment: ObjectProperty<Pos>	Specifies the alignment of the text and node in the labeled.
-contentDisplay: ObjectProperty<ContentDisplay>	Specifies the position of the node relative to the text using the constants TOP, BOTTOM, LEFT, and RIGHT defined in ContentDisplay.
-graphic: ObjectProperty<Node>	A graphic for the labeled.
-graphicTextGap: DoubleProperty	The gap between the graphic and the text.
-textFill: ObjectProperty<Paint>	The paint used to fill the text.
-text: StringProperty	A text for the labeled.
-underline: BooleanProperty	Whether text should be underlined.
-wrapText: BooleanProperty	Whether text should be wrapped if the text exceeds the width.

3

Label

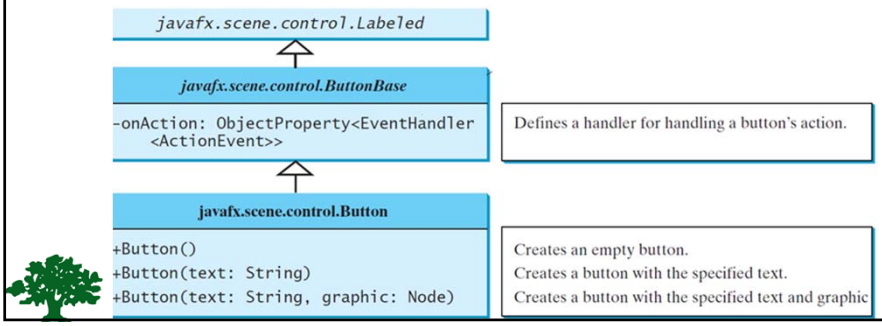
- The **Label** class defines labels.



4

ButtonBase and Button

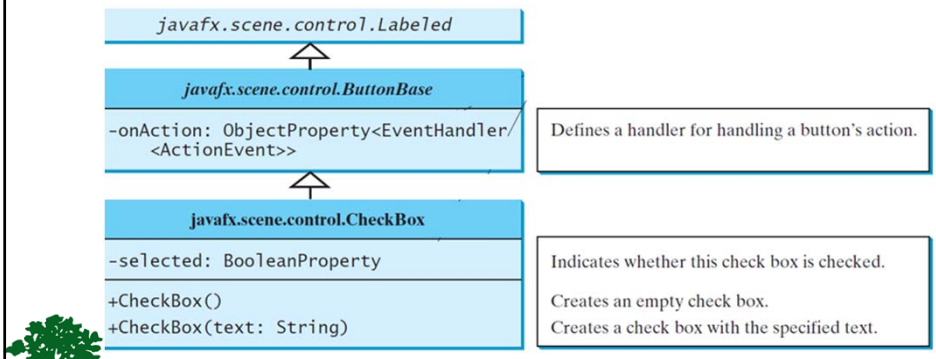
- A ***button*** is a control that triggers an action event when clicked.
- JavaFX provides regular buttons, toggle buttons, check box buttons, and radio buttons.
- The common features of these buttons are defined in **ButtonBase** and **Labeled** classes.



5

CheckBox

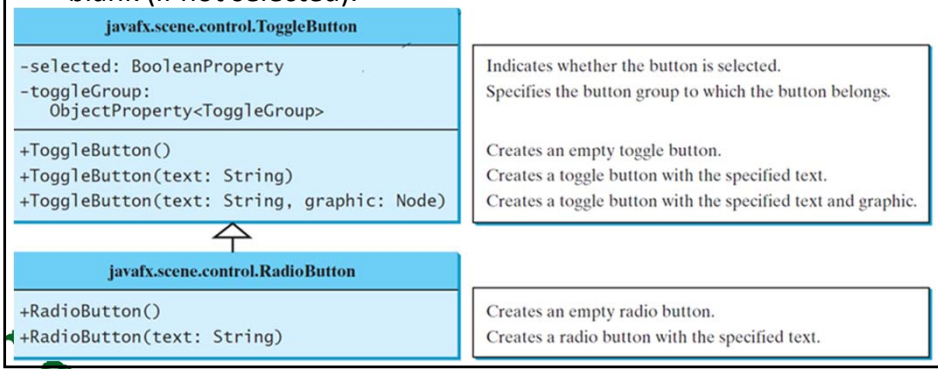
- A **CheckBox** is used for the user to make a selection.
- Like **Button**, **CheckBox** inherits all the properties such as **onAction**, **text**, **graphic**, **alignment**, **graphicTextGap**, **textFill**, **contentDisplay** from **ButtonBase** and **Labeled**.



6

RadioButton

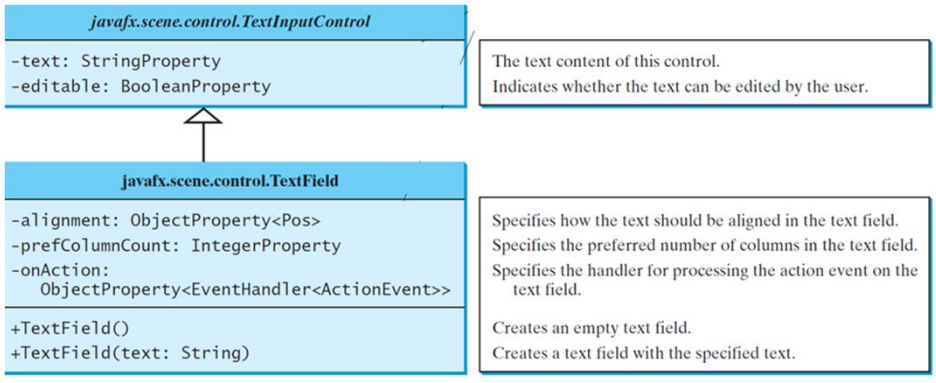
- Radio buttons, also known as *option buttons*, enable you to choose a single item from a group of choices.
- In appearance radio buttons resemble check boxes, but check boxes display a square that is either checked or blank, whereas radio buttons display a circle that is either filled (if selected) or blank (if not selected).



7

TextField

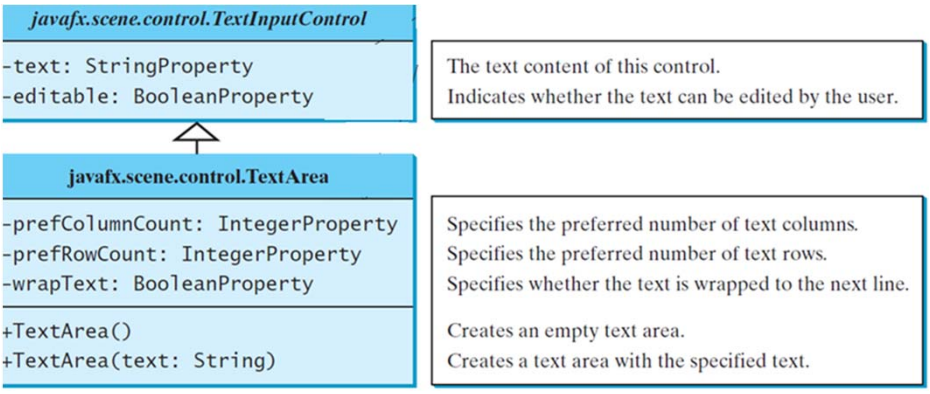
- A text field can be used to enter or display a string. **TextField** is a subclass of **TextInputControl**.



8

TextArea

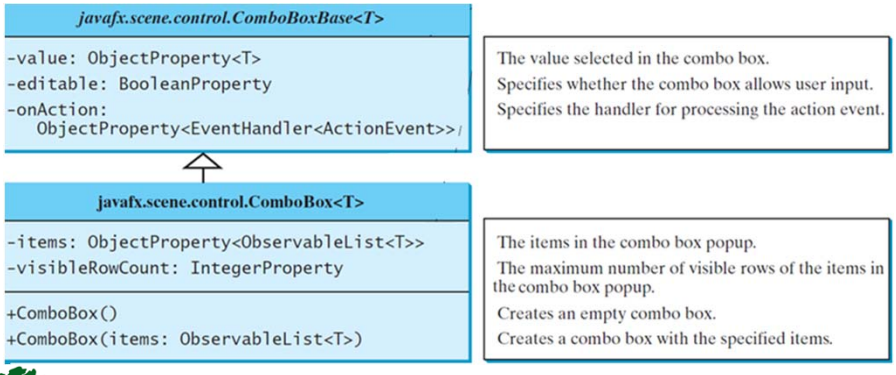
❖ A **TextArea** enables the user to enter multiple lines of text.



9

ComboBox

❖ A combo box, also known as a choice list or drop-down list, contains a list of items from which the user can choose.



10

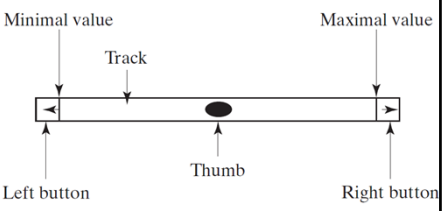
ListView

❖ A *list view* is a component that performs basically the same function as a combo box, but it enables the user to choose a single value or multiple values.

javafx.scene.control.ListView<T>	
-items: ObjectProperty<ObservableList<T>>	The items in the list view.
-orientation: BooleanProperty	Indicates whether the items are displayed horizontally or vertically in the list view.
-selectionModel: ObjectProperty<MultipleSelectionModel<T>>	Specifies how items are selected. The `SelectionModel` is also used to obtain the selected items.
+ListView()	Creates an empty list view.
+ListView(items: ObservableList<T>)	Creates a list view with the specified items.

ScrollBar

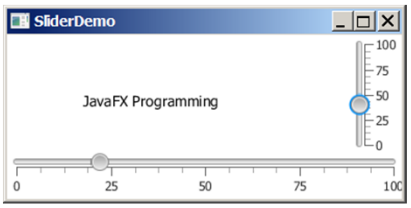
❖ A *scroll bar* is a control that enables the user to select from a range of values. The scrollbar appears in two styles: *horizontal* and *vertical*.



javafx.scene.control.ScrollBar	
-blockIncrement: DoubleProperty	The amount to adjust the scroll bar if the track of the bar is clicked (default: 10).
-max: DoubleProperty	The maximum value represented by this scroll bar (default: 100).
-min: DoubleProperty	The minimum value represented by this scroll bar (default: 0).
-unitIncrement: DoubleProperty	The amount to adjust the scroll bar when the `increment()` and `decrement()` methods are called (default: 1).
-value: DoubleProperty	Current value of the scroll bar (default: 0).
-visibleAmount: DoubleProperty	The width of the scroll bar (default: 15).
-orientation: ObjectProperty<Orientation>	Specifies the orientation of the scroll bar (default: `HORIZONTAL`).
+ScrollBar()	Creates a default horizontal scroll bar.
+increment()	Increments the value of the scroll bar by `unitIncrement`.
+decrement()	Decrements the value of the scroll bar by `unitIncrement`.

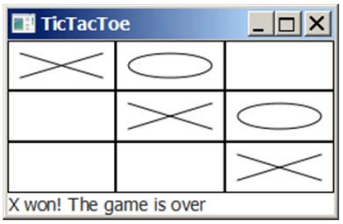
Slider

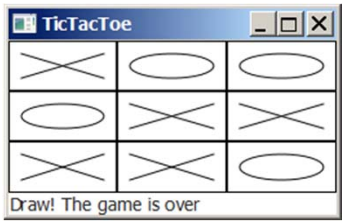
- **Slider** is similar to **ScrollBar**, but Slider has more properties and can appear in many forms.

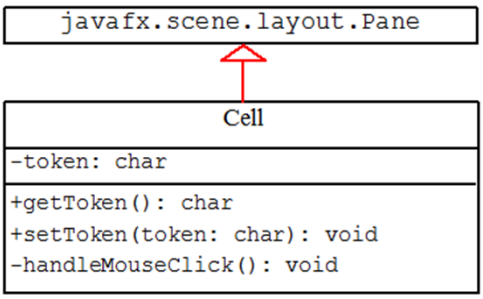


javafx.scene.control.Slider	
-blockIncrement: DoubleProperty	The amount to adjust the slider if the track of the bar is clicked (default: 10).
-max: DoubleProperty	The maximum value represented by this slider (default: 100).
-min: DoubleProperty	The minimum value represented by this slider (default: 0).
-value: DoubleProperty	Current value of the slider (default: 0).
-orientation: ObjectProperty<Orientation>	Specifies the orientation of the slider (default: HORIZONTAL).
-majorTickUnit: DoubleProperty	The unit distance between major tick marks.
-minorTickCount: IntegerProperty	The number of minor ticks to place between two major ticks.
-showTickLabels: BooleanProperty	Specifies whether the labels for tick marks are shown.
-showTickMarks: BooleanProperty	Specifies whether the tick marks are shown.
+Slider()	Creates a default horizontal slider.
+Slider(min: double, max: double, value: double)	Creates a slider with the specified min, max, and value.

Case Study: TicTacToe

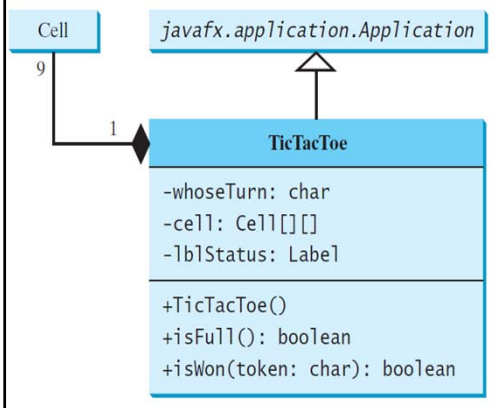






Token used in the cell (default: ' ').
Returns the token in the cell.
Sets a new token in the cell.
Handles a mouse click event.

Case Study: TicTacToe cont.

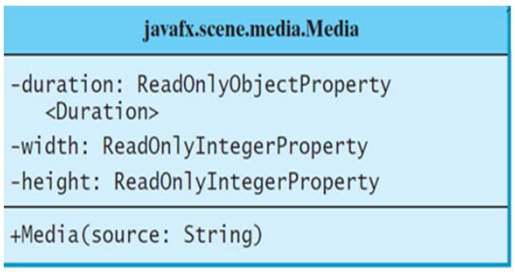


TicTacToe

15

Media

- You can use the **Media** class to obtain the source of the media, the **MediaPlayer** class to play and control the media, and the **MediaView** class to display the video.



16

MediaPlayer

❖The **MediaPlayer** class playes and controls the media with properties such as **autoPlay**, **currentCount**, **cycleCount**, **mute**, **volume**, and **totalDuration**.

javafx.scene.media.MediaPlayer	
-autoPlay: BooleanProperty	Specifies whether the playing should start automatically.
-currentCount: ReadOnlyIntegerProperty	The number of completed playback cycles.
-cycleCount: IntegerProperty	Specifies the number of time the media will be played.
-mute: BooleanProperty	Specifies whether the audio is muted.
-volume: DoubleProperty	The volume for the audio.
-totalDuration: ReadOnlyObjectProperty<Duration>	The amount of time to play the media from start to finish.
+MediaPlayer(media: Media)	Creates a player for a specified media.
+play(): void	Plays the media.
+pause(): void	Pauses the media.
+seek(): void	Seeks the player to a new playback time.

17

MediaView

❖ The **MediaView** class is a subclass of **Node** that provides a view of the **Media** being played by a **MediaPlayer**. The **MediaView** class provides the properties for viewing the media.

javafx.scene.media.MediaView	
-x: DoubleProperty	Specifies the current x-coordinate of the media view.
-y: DoubleProperty	Specifies the current y-coordinate of the media view.
-mediaPlayer: ObjectProperty<MediaPlayer>	Specifies a media player for the media view.
-fitWidth: DoubleProperty	Specifies the width of the view for the media to fit.
-fitHeight: DoubleProperty	Specifies the height of the view for the media to fit.
+MediaView()	Creates an empty media view.
+MediaView(mediaPlayer: MediaPlayer)	Creates a media view with the specified media player.

18

BIRZEIT UNIVERSITY

Event-Driven Programming

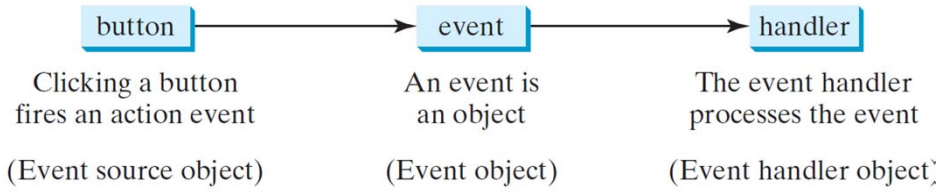
By: Mamoun Nawahdah (Ph.D.)
2015/2016

Procedural vs. Event-Driven Programming

- ***Procedural programming*** is executed in procedural order.
- In ***event-driven programming***, code is executed upon activation of events.

Handling GUI Events

- **Source object** (e.g., button)
- **Listener object** contains a method for processing the event.



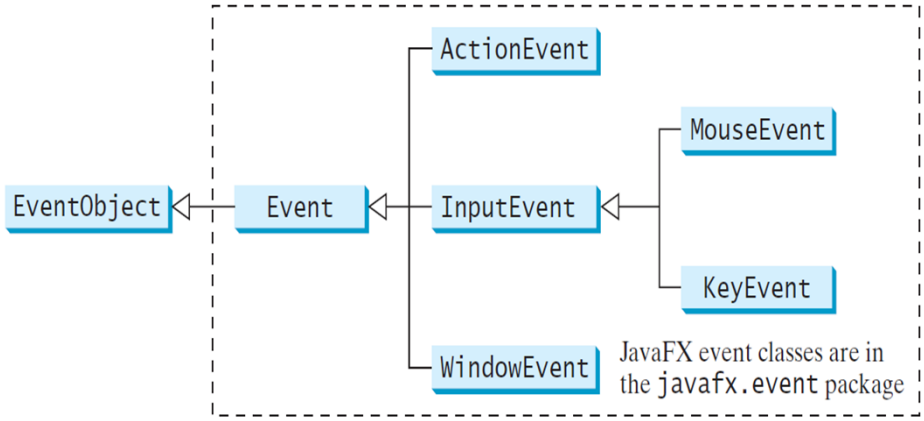
3

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 clicks, or keystrokes.

4

Event Classes



5

Event Information

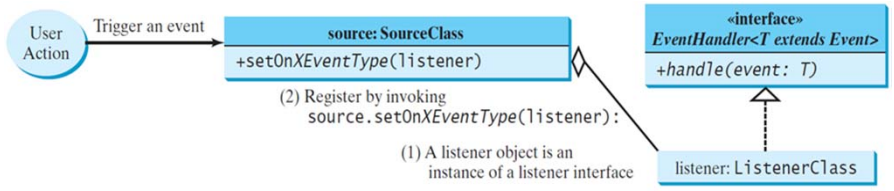
- An event object contains whatever properties are pertinent to the event.
- You can identify the source object of the event using the **getSource()** instance method in the **EventObject** class.
- The subclasses of **EventObject** deal with special types of events, such as button actions, window events, component events, mouse movements, and keystrokes.

6

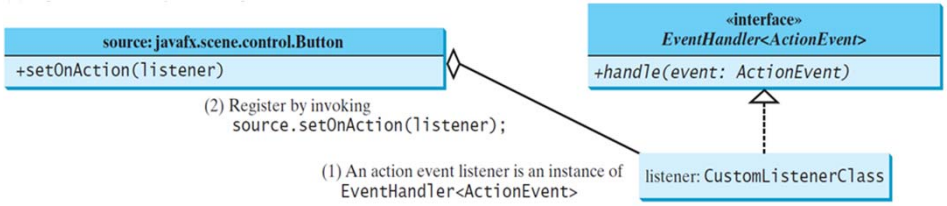
Selected User Actions and Handlers

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

The Delegation Model



(a) A generic source object with a generic event T



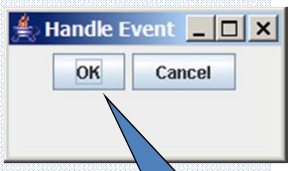
(b) A `Button` source object with an `ActionEvent`

The Delegation Model: Example

```
Button btOK = new Button("OK");
OKHandlerClass handler = new OKHandlerClass();
btOK.setOnAction(handler);
```


```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

1. Start from the main method to create a window and display it

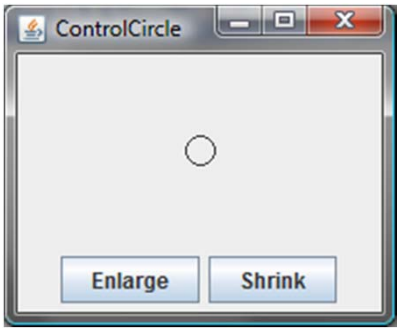


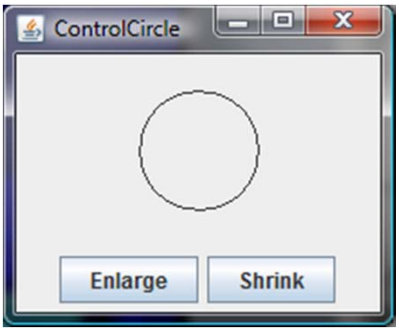
2. Click OK

3. Click OK. The JVM invokes the listener's handle method

Example: ControlCircle

❖ Now let us consider to write a program that uses two buttons to control the size of a circle.





11

Inner Class Listeners

❖ A listener class is designed specifically to create a listener object for a GUI component (e.g., a button).

❖ It will **not be shared** by other applications.

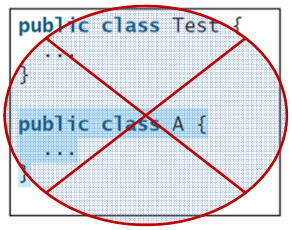
❖ So, it is appropriate to define the listener class inside the frame class as an **inner class**.

12

Inner Classes

- **Inner class**: A class is a member of another class.
- Advantages: In some applications, you can use an inner class to make programs **simple**:
  - An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.

Inner Classes cont.



(a)

```
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

Inner Classes cont.

- Inner classes can make programs simple and concise.
- An inner class supports the work of its containing outer class and is compiled into a class named ***OuterClassName*$*InnerClassName*.class**.
  - For example, the inner class **InnerClass** in **OuterClass** is compiled into ***OuterClass*$*InnerClass*.class** .

15

Inner Classes cont.

- An inner class can be declared **public**, **protected**, or **private** subject to the same visibility rules applied to a member of the class.
- An inner class can be declared **static**.
- A **static** inner class can be accessed using the outer class name.
- A **static** inner class cannot access non-static members of the outer class

16

Anonymous Inner Classes

- An anonymous inner class **must** always extend a superclass or implement an interface, **but** it cannot have an explicit **extends** or **implements** clause.
- An anonymous inner class **must** implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is **Object()**.
- An anonymous inner class is compiled into a class named **OuterClassName$*n*.class**.
  - For example, if the outer class **Test** has two anonymous inner classes, these two classes are compiled into **Test$1.class** and **Test$2.class**.

17

Anonymous Inner Classes cont.

- Inner class listeners can be shortened using anonymous inner classes.
- An *anonymous inner class* is an inner class without a name.
- It combines declaring an inner class and creating an instance of the class in one step.
- An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface
  // Other methods if necessary
}
```

18

Anonymous Inner Classes cont.

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new class EnlargeHandler
      implements EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

(b) Anonymous inner class

Simplifying Event Handing Using Lambda Expressions

- ***Lambda expression*** is a new feature in **Java 8**.
- Lambda expressions can be viewed as an anonymous method with a concise syntax.
- For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

```
btEnlarge.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      // Code for processing event e
    }
  }
});
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

(b) Lambda expression event handler

Basic Syntax for a Lambda Expression

❖ The basic syntax for a lambda expression is either:

(type1 param1, type2 param2, ...) -> expression

or

(type1 param1, type2 param2, ...) -> { statements; }

❖ The data type for a parameter may be explicitly declared or implicitly inferred by the compiler.

❖ The parentheses can be omitted if there is only one parameter without an explicit data type.

21

Single Abstract Method Interface (SAM)

❖ The statements in the lambda expression is all for that method.

❖ If it contains multiple methods, the compiler will not be able to compile the lambda expression.

❖ So, for the compiler to understand lambda expressions, the interface **must** contain exactly one abstract method.

❖ Such an interface is known as a ***functional interface***, or a ***Single Abstract Method*** **(SAM)** interface.

22

MouseEvent

javafx.scene.input.MouseEvent	
+getButton(): MouseButton	Indicates which mouse button has been clicked.
+getClickCount(): int	Returns the number of mouse clicks associated with this event.
+getX(): double	Returns the *x*-coordinate of the mouse point in the event source node.
+getY(): double	Returns the *y*-coordinate of the mouse point in the event source node.
+getSceneX(): double	Returns the *x*-coordinate of the mouse point in the scene.
+getSceneY(): double	Returns the *y*-coordinate of the mouse point in the scene.
+getScreenX(): double	Returns the *x*-coordinate of the mouse point in the screen.
+getScreenY(): double	Returns the *y*-coordinate of the mouse point in the screen.
+isAltDown(): boolean	Returns true if the `Alt` key is pressed on this event.
+isControlDown(): boolean	Returns true if the `Control` key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse `Meta` button is pressed on this event.
+isShiftDown(): boolean	Returns true if the `Shift` key is pressed on this event.

23

The KeyEvent Class

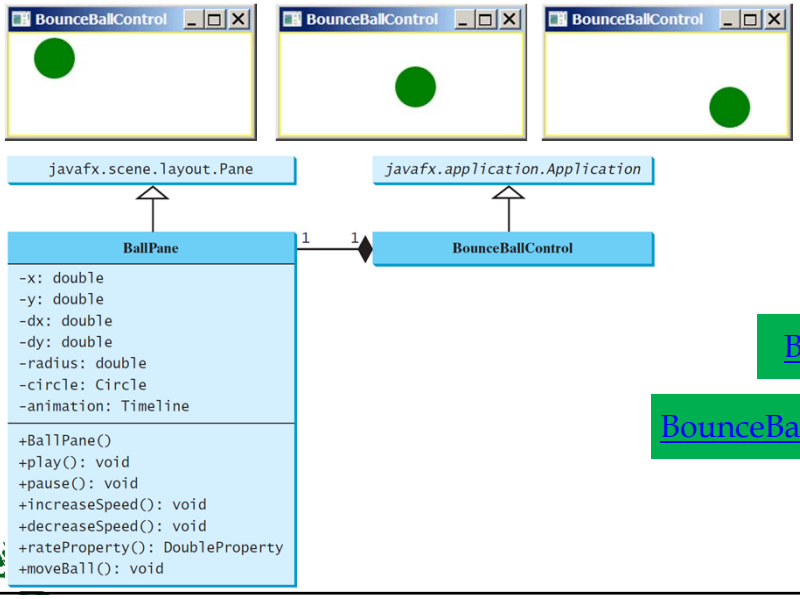
javafx.scene.input.KeyEvent	
+getCharacter(): String	Returns the character associated with the key in this event.
+getCode(): KeyCode	Returns the key code associated with the key in this event.
+getText(): String	Returns a string describing the key code.
+isAltDown(): boolean	Returns true if the `Alt` key is pressed on this event.
+isControlDown(): boolean	Returns true if the `Control` key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse `Meta` button is pressed on this event.
+isShiftDown(): boolean	Returns true if the `Shift` key is pressed on this event.

24

The KeyCode Constants

Constant	*Description*	*Constant*	*Description*
HOME	The Home key	CONTROL	The Control key
END	The End key	SHIFT	The Shift key
PAGE_UP	The Page Up key	BACK_SPACE	The Backspace key
PAGE_DOWN	The Page Down key	CAPS	The Caps Lock key
UP	The up-arrow key	NUM_LOCK	The Num Lock key
DOWN	The down-arrow key	ENTER	The Enter key
LEFT	The left-arrow key	UNDEFINED	The keyCode unknown
RIGHT	The right-arrow key	F1 to F12	The function keys from F1 to F12
ESCAPE	The Esc key	0 to 9	The number keys from 0 to 9
TAB	The Tab key	A to Z	The letter keys from A to Z

Case Study: Bouncing Ball



BallPane

BounceBallControl