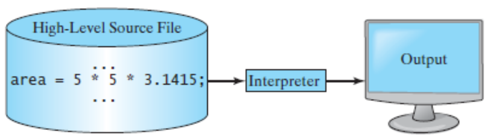
Networks and Internet Programming

Java Revision – Part I

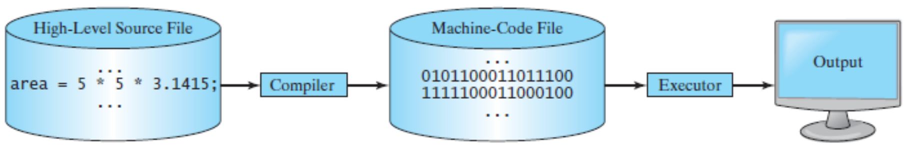
Interpreters vs. Compilers

- Interpreters read one statement from the source code, translate it to machine code and execute it right away.



```
...
area = 5 * 5 * 3.1415;
...
```

- Compilers translate the entire source code into a machine code, and the machine file is then executed.



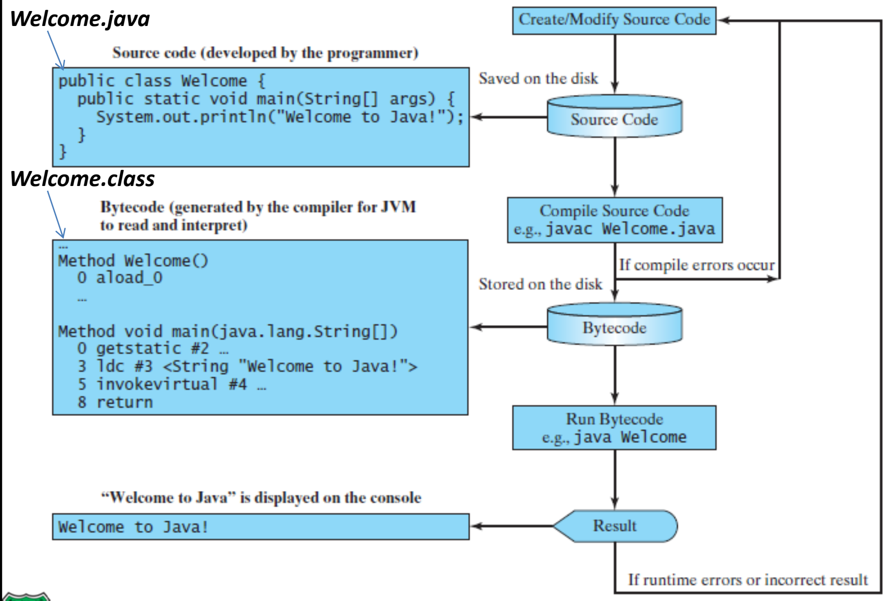
```
...
area = 5 * 5 * 3.1415;
...
```

```
...
0101100011011100
1111100011000100
...
```

Java API, JDK, and IDE

- The *Application Program Interface* (*API*) contains predefined classes and interfaces for developing Java programs.
- The *Java Development Kit* (*JDK*) consists of a set of separate programs, each invoked from command line, for developing and testing Java programs.
- The *Integrated Development Kit* (*IDE*) provides graphical user interface to edit, compile, build, and debug programs.

Creating, Compiling, and Executing a Java Program

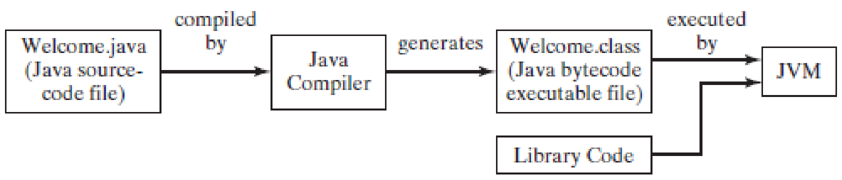

Welcome.java
Source code (developed by the programmer)
Welcome.class
Bytecode (generated by the compiler for JVM to read and interpret)
"Welcome to Java" is displayed on the console
Create/Modify Source Code
Saved on the disk
Source Code
Compile Source Code e.g., javac Welcome.java
If compile errors occur
Stored on the disk
Bytecode
Run Bytecode e.g., java Welcome
Result
If runtime errors or incorrect result

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

```
…
Method Welcome()
  0 aload_0
  …

Method void main(java.lang.String[])
  0 getstatic #2 …
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 …
  8 return
```

```
Welcome to Java!
```

Creating, Compiling, and Executing a Java Program (Cont.)


Welcome.java (Java source-code file)
compiled by
Java Compiler
generates
Welcome.class (Java bytecode executable file)
executed by
JVM
Library Code

- Java source code is compiled into Java *bytecode*.
- Your Java code may use the code in the Java library.
- The *bytecode* is similar to machine instructions, but is *architecture neutral* and can run on any platform that has a *Java Virtual Machine* (*JVM*).
- The JVM is an interpreter, which translates individual instructions in the bytecode into the target machine language code and executes it immediately.

Programming Errors

- Syntax errors.
  - Detected by the compiler.
  - Result from errors in code construction.
- Runtime errors.
  - Cause a program to terminate abnormally.
  - Occur while the program is running if the environment detects an operation that is impossible to carry out.
  - Examples include input errors and division by zero.
- Logic errors.
  - Occur when a program does not perform the way it is intended to.

Primitive Data Types

datatype	size	range	Default value	Wrapper class
byte	1 byte	-128 to 127	0	Byte
short	2 bytes	-32768 to 32767	0	Short
int	4 bytes	-2^{31} to $2^{31}-1$	0	Integer
long	8 bytes	-2^{63} to $2^{63}-1$	0	Long
float	4 bytes	-3.4e38 to 3.4e38	0.0	Float
double	8 bytes	-1.7e308 to 1.7e308	0.0	Double
boolean	NA	NA(But allowed values are true, false)	false	Boolean
char	2 bytes	0 to 65535	0(balnk spaces)	Character

Identifiers

- Identifiers are the names of things that appear in the program.
  – Names of variables, constants, methods, classes, packages…
- All identifiers must obey the following rules:
  – An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar sign ($).
  – Cannot start with a digit.
  – Cannot be a reserved word.
  – Can be of any length.
- Examples of legal identifiers: $2, area, Area, S_3.
- Examples of illegal identifiers: 2A, d+4, S#6.

Variables

- Variables are used to represent values that may be changed in the program.
- The syntax for declaring a variable:

  datatype variableName
- Examples of variable declarations:
  - *int count;*
  - *double rate;*
  - *char letter;*
  - *boolean found;*

Assignment Statement

- An assignment statement designates a value for a variable.
- The *equal sign (=)* is used as the assignment operator.
- Examples:
  - *x = 1;*
  - *x = x+1;*
  - *area = radius * radius * 3.14159;*

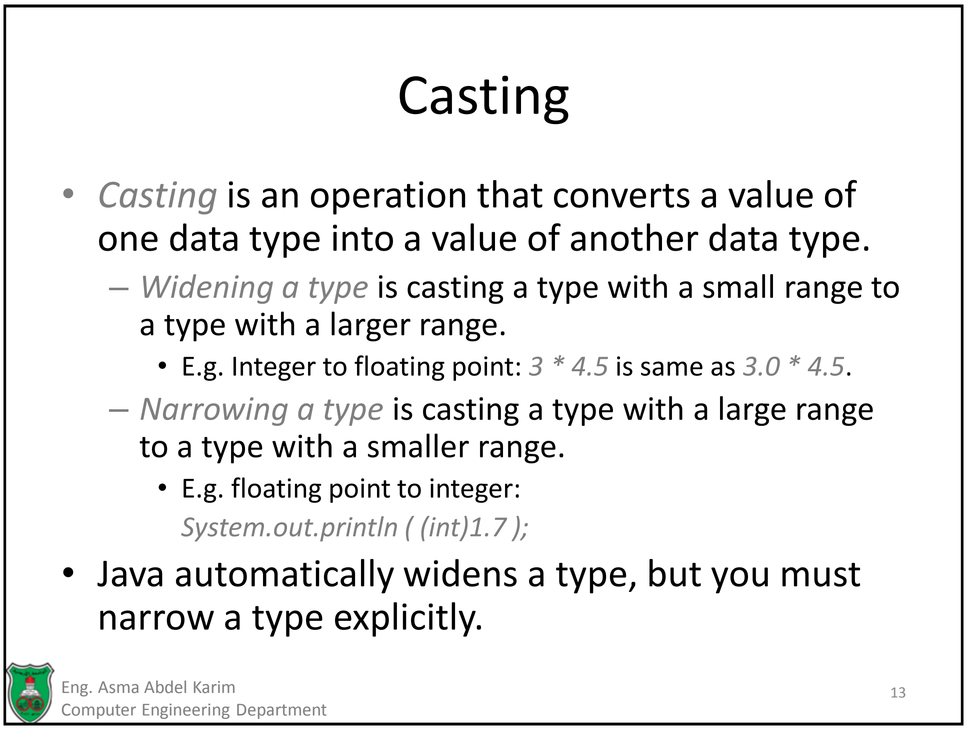
Character Literals

- A character literal is a single character enclosed in single quotation marks (‘ ‘).
- Escape characters are used to represent special characters.

Escape Character	*Name*
`\b`	Backspace
`\t`	Tab
`\n`	Linefeed
`\f`	Formfeed
`\r`	Carriage Return
`\\`	Backslash
`\"`	Double Quote

Constants

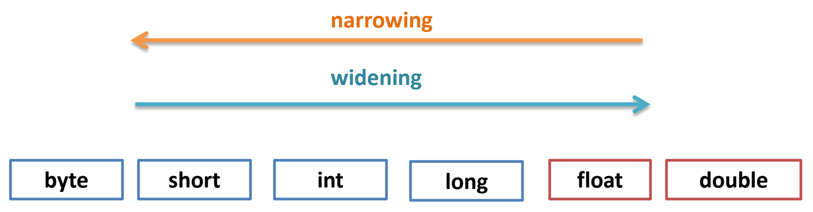
- A *constant* is an identifier that represents a permanent value.
- The syntax for declaring a constant:
  - *final datatype CONSTANT_NAME = value;*
- Example:
  - *final double PI = 3.14159;*



Casting

- *Casting* is an operation that converts a value of one data type into a value of another data type.
  - *Widening a type* is casting a type with a small range to a type with a larger range.
    - E.g. Integer to floating point: *3 * 4.5* is same as *3.0 * 4.5*.
  - *Narrowing a type* is casting a type with a large range to a type with a smaller range.
    - E.g. floating point to integer:
      System.out.println ((int)1.7);
- Java automatically widens a type, but you must narrow a type explicitly.

Casting (Cont.)



The String Type

- A string is a sequence of characters.
- To represent a string of characters, use the data type called *String*:
  - E.g. *String message = "Welcome to Java";*
- String is a predefined class in the Java library.
- The String type is not a primitive type.
- A string literal must be enclosed on quotation marks (" ").

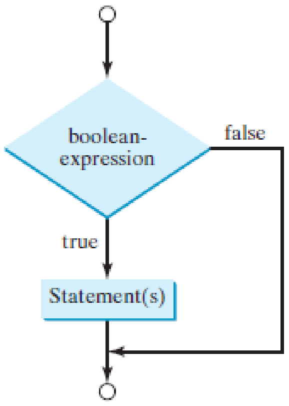
The String Type (Cont.)

- The *plus sign* (+) is the *concatenation* operator if at least one of the operands is a string.
  - If one of the operands is a non string (e.g. a number), the non string value is converted into a string and concatenated with the string.
  - Examples:
    - *String message = "Welcome " + "to " + "Java!";*
      message becomes: Welcome to Java!
    - *String s = "Chapter" + 2;*
      s becomes: Chapter2
    - *String appendix = "Appendix" + 'B';*
      appendix becomes: AppendixB
- If neither of the operands is a string, the *plus sign* (+) is the *addition* operator.

Selections: One-way If Statements

- A *one-way if* statement executes an action *if an only if* the condition is *true*.
  - If the condition is *false*, nothing is done.
- The syntax for a one-way if statement is:

if (boolean-expression){
statement(s);
}



Selections: Two-way If-else Statements

- A *two-way if-else* statement executes an action if the condition is *true* and another action if the condition is *false*.
- The syntax for a two-way if-else statement is:

if (boolean-expression){
statement(s)-for-the-true-case;
}
else{
statement(s)-for-the-false-case;
}

Selections: Nested If

- An if statement can be inside another if statement to form a *nested* if statement.
- Example:

```
if (i > k){
    if (j > k)
        System.out.println("i and j are greater than k");
}
else
    System.out.println("i is less than or equal to k");
```

Executed only if i>k and j>k

Executed if i<=k

Selections: switch Statements

- The syntax for the switch statement is:

```
switch (switch-expression){
    case value1: statement(s)1;
                 break;
    case value2: statement(s)2;
                 break;
    .....
    case valueN: statement(s)N;
                 break;
    default:     statement(s)-for-default;
}
```

Must yield a value of char, byte, short, int, or string

Constant expressions of the same type as the value of switch-expression

When the value in a case statement matches the value of the switch-expression, statements starting from this case are executed until either a break statement or the end of the switch statement is reached

Statements of the default case are executed when none of the specified cases matches the switch-expression.

Selections: Conditional Expressions

- A *conditional expression* evaluates an expression based on a condition.
- The syntax is:
  - *boolean-expression ? expression1 : expression2;*
  - The result of the conditional expression is *expression1* if *boolean-expression* is *true*, otherwise the result is *expression2*.
- Example:

  max = (num1 > num2) ? num1 : num2;

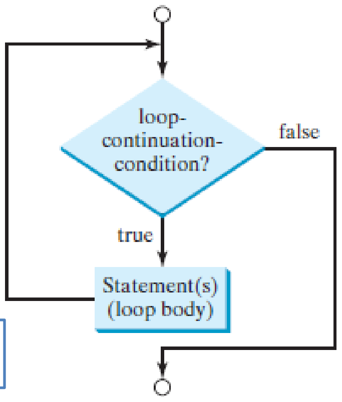
While Loops

- A *while* loop executes statements repeatedly while the condition is *true*.
- The syntax for the *while* loop is:

```
while (loop-continuation-condition){
     statement(s);
}
```

Loop body

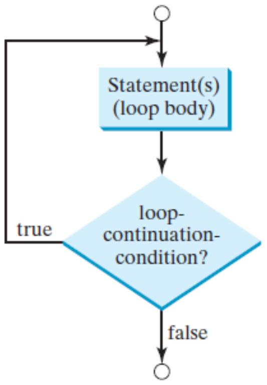
Evaluated each time to determine whether to execute the loop body



Do-While Loops

- Same as the *while* loop except that it executes the loop body first then checks the loop continuation condition.
- The syntax for the *do-while* loop:

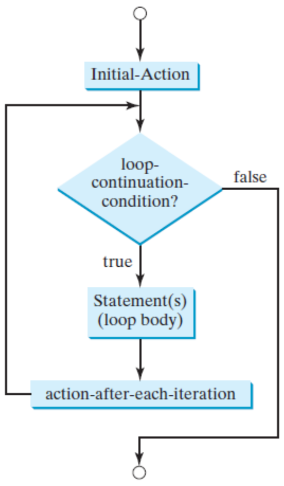
```
do {
   statement(s);
} while (loop-continuation-condition);
```



For Loops

- A for loop has a concise syntax for writing loops.
- The syntax for the for loop is:

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



Keywords *break* and *continue*

- The *break* and *continue* keywords provide additional controls in a loop.
- The *break* keyword is used in a loop to immediately terminate the loop.
- Example of using the *break* keyword:

```
for (int n=0, sum=0; n<20; n++){
      sum += n;
      if (sum >= 100) break;
}
```

Eng. Asma Abdel Karim
Computer Engineering Department

25

Keywords *break* and *continue* (Cont.)

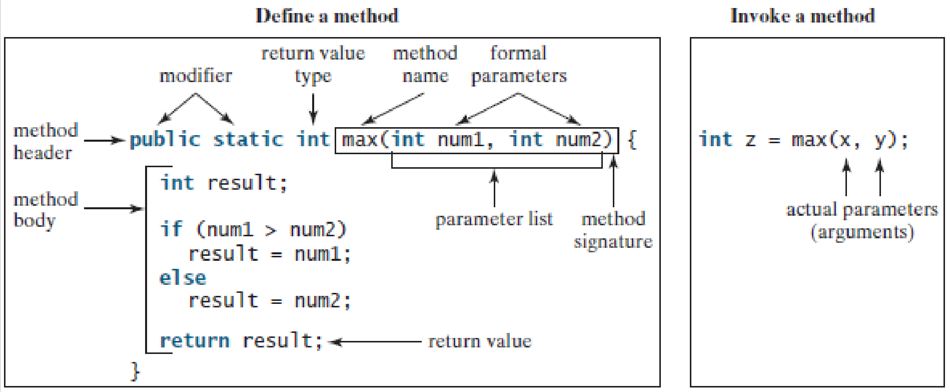
- The *continue* keyword is used in a loop to end the current iteration and program control goes to the end of the loop body.
- Example of using the *continue* keyword:

```
for (int n=0, sum=0; n<20; n++){
       if (n == 10 || n == 11) continue;
      sum += n;
}
```

Eng. Asma Abdel Karim
Computer Engineering Department

26

Method Definition: An Example

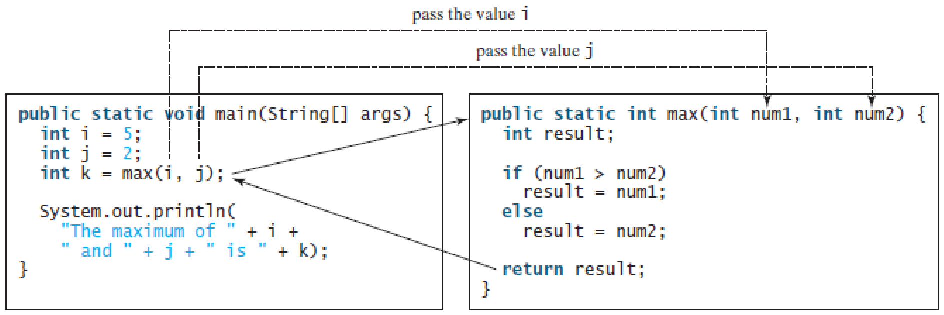


```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

- In a method definition, you define what the method is to do.

Method Invocation: An Example



```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum of " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

- Calling a method executes the code in the method.
- The *main* method is just like any other method except that it is invoked by the *JVM* to start the program.

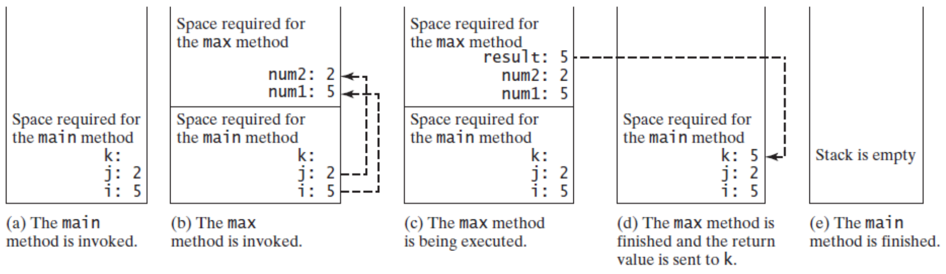
What happens when a method is invoked?

- Each time a method is invoked, the system creates an *activation record*.
  - *Activation record* stores parameters and variables for the method .
  - *Activation record* is placed in an area of memory known as the *call stack*, or simply the *stack*.
- When a method invokes another method, the caller's activation record is kept intact, and a new activation record is created.
- When a method finishes its work and returns to its caller, its activation record is removed from the stack.
- A call stack stores methods in last-in, first-out fashion.

Eng. Asma Abdel Karim
Computer Engineering Department

29

What happens when a method is invoked? (Example)


Space required for the main method
k:
j: 2
i: 5
Space required for the max method
num2: 2
num1: 5
result: 5
k: 5
Stack is empty
(a) The main method is invoked.
(b) The max method is invoked.
(c) The max method is being executed.
(d) The max method is finished and the return value is sent to k.
(e) The main method is finished.

Eng. Asma Abdel Karim
Computer Engineering Department

30

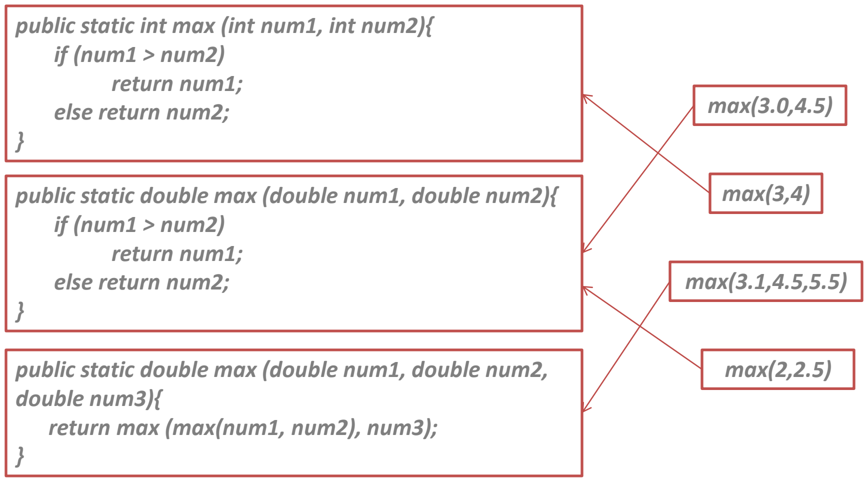
Passing Parameters by Values (Cont.)

- When you invoke a method with an argument, the value of the argument is passed to the parameter.
  - This is referred to as *pass-by-value*.
- If a value of a variable is passed as an argument to a parameter, the variable is not affected, regardless of the changes made to the parameter inside the method.

Overloading Methods

- Two methods that have the *same name*, but *different parameter lists* within one class.
- The Java compiler determines which method to use based on the *method signature*.
  - It finds the most specific method for a method invocation.

Overloading Methods: An Example



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

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

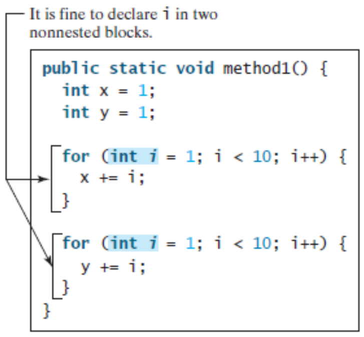
```
public static double max (double num1, double num2,
double num3){
      return max (max(num1, num2), num3);
}
```

The Scope of Variables

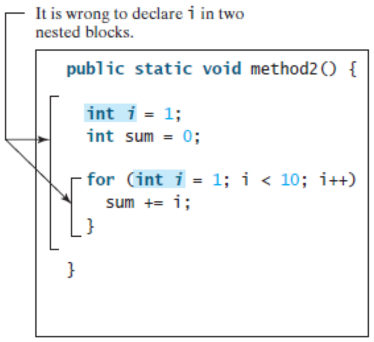
- The *scope* of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a method is referred to as a *local variable*.
- A parameter is actually a local variable.
  - The scope of a method parameter covers the entire method.

The Scope of Variables (Cont.)

- You can declare a local variable with the same name in different blocks in a method.
- But you cannot declare a local variable twice in the same block or in nested 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;
  }
}
```



```
public static void method2() {

  int i = 1;
  int sum = 0;

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

}
```

Eng. Asma Abdel Karim
Computer Engineering Department
35

Declaring Arrays

- To use an array in a program, you must *declare* a variable to reference the array and specify the array's elements type.
  - All elements in the array have the same data type.
- The syntax for declaring an array variable is:

  elementType [] arrayRefVar;
- Example:

  double [] myList;

Eng. Asma Abdel Karim
Computer Engineering Department
36

Declaring Arrays (Cont.)

- Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array.
  - It creates only a storage location for the reference to an array.

Creating Arrays

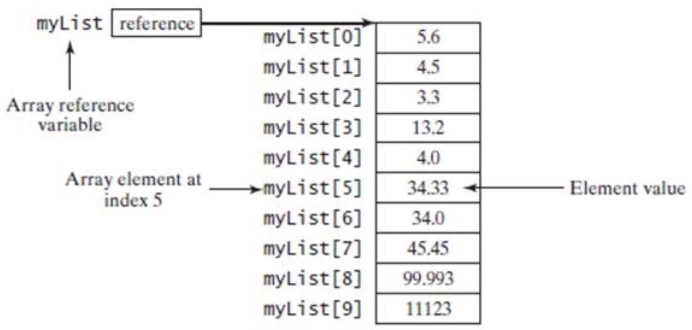
- An array is created using the *new* operator with the following syntax:
  arrayRefVar = new elementType [arraySize];
  - This statement does two things:
    - It creates an array using new elementType [arraySize]
    - It assigns the reference of the newly created array to the variable arrayRefVar.
- Example:

```
double [] myList;                //array declaration
mylist = new double [10];               //array creation
```

Array Size and Default Values

- The size of an array cannot be changed after the array is created.
- Array size can be obtained using: *arrayRefVar.length*
- When an array is created, its elements are assigned their default values:
  - *0* for *numeric* primitive data types.
  - *\u0000* for *char* types.
  - *False* for *boolean* types.

Array Indexed variables



- The array elements are accessed through the index.
- Array indices are 0 based.
- Each element in the array is represented using the following syntax: *arrayRefVar [index]*

Array Initializers

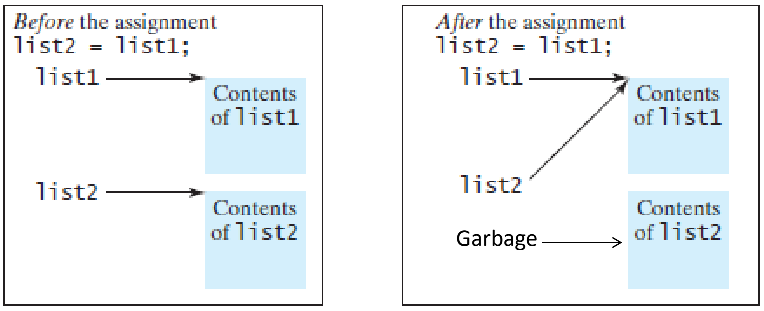
- *Array initializer* is a shorthand notation which combines the *declaration*, *creation*, and *initialization* of an array in one statement.
- The syntax for array initializer:

  elementType [] arrayRefVar = {value0, value1, …., valuek};
- Example:

  double [] myList = {1.9, 2.5, 3.4, 4.5};

Copying Arrays

- The assignment operator does not copy the contents of an array into another, it instead merely copies the reference values.



Copying Arrays (Cont.)

- To copy the contents of one array into another, you have to copy the *array's individual elements* into the other array.
- Use a loop to copy every element from the source array to the corresponding element in the target array.
- Example:

```
int [] sourceArray = {2, 3, 1, 5, 10};
int [] targetArray = new int [sourceArray.length];
for (int i=0; i < sourceArray.length; i++)
        targetArray [i] = sourceArray [i];
```

Passing Arrays to Methods

- When passing an array to a method, the *reference* of the array is passed to a method.
- This differs from passing arguments of a primitive type:
  - For an argument of a primitive type, the argument's value is passed.
    - *The passed variable will not be affected by any change to the value inside the method.*
  - For any argument of an array type, the value of the argument is a reference to an array.
    - *The passed array will be affected by any change inside the method.*

Passing Arrays to Methods: Example

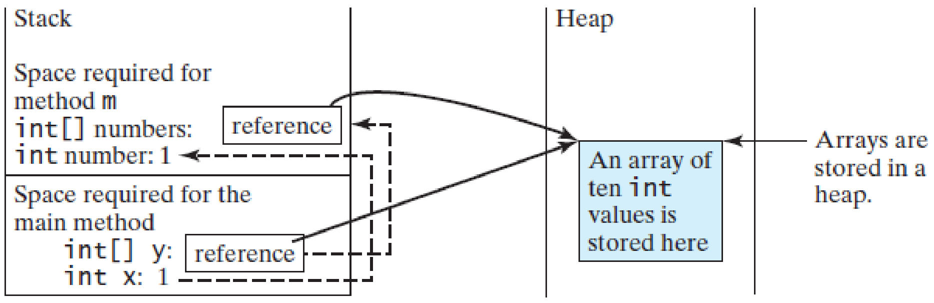
```
public class Test {
    public static void main (String [] args){
          int x=1;
          int [] y = new int [10];
          m (x, y);
          System.out.println("x is "+ x);
          System.out.println("y[0] is "+ y[0]);
    }
    public static void m(int number, int [] numbers){
          number =1003;
          numbers[0]=5555;
    }
}
```

Output:
x is 1
y[0] is 5555

Eng. Asma Abdel Karim
Computer Engineering Department

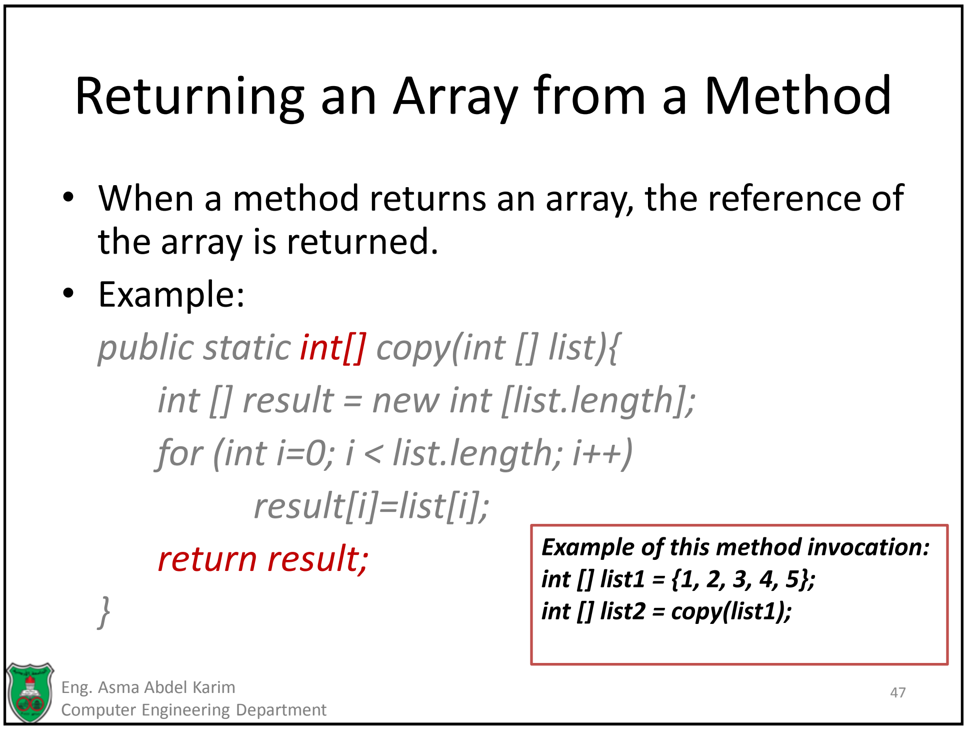
45

Passing Arrays to Methods: Example (Cont.)


Stack
Space required for method m
int[] numbers:
reference
int number: 1
Space required for the main method
int[] y:
reference
int x: 1
Heap
An array of ten int values is stored here
Arrays are stored in a heap.

Eng. Asma Abdel Karim
Computer Engineering Department

46



Returning an Array from a Method

- When a method returns an array, the reference of the array is returned.
- Example:

```
public static int[] copy(int [] list){
      int [] result = new int [list.length];
      for (int i=0; i < list.length; i++)
                result[i]=list[i];
      return result;
}
```

Example of this method invocation:
int [] list1 = {1, 2, 3, 4, 5};
int [] list2 = copy(list1);

Eng. Asma Abdel Karim
Computer Engineering Department

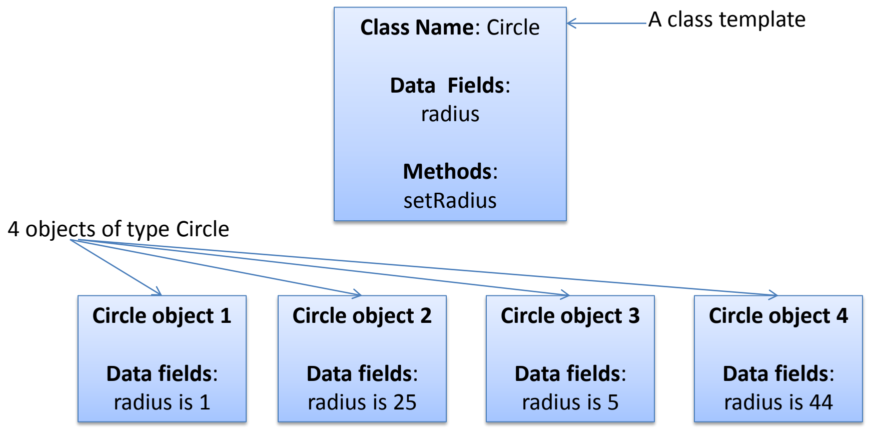
47

Networks and Internet Programming

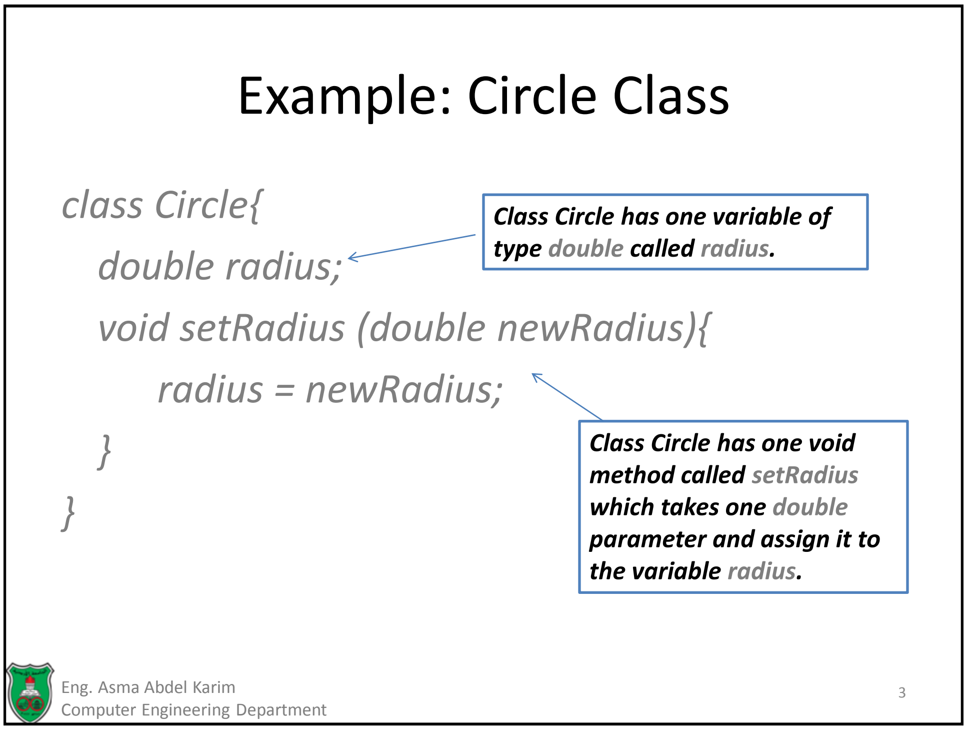
Java Revision – Part II

Eng. Asma Abdel Karim
Computer Engineering Department

What is a Class? Example


Class Name: Circle
Data Fields: radius
Methods: setRadius
A class template
4 objects of type Circle
Circle object 1 **Data fields**: radius is 1
Circle object 2 **Data fields**: radius is 25
Circle object 3 **Data fields**: radius is 5
Circle object 4 **Data fields**: radius is 44

Eng. Asma Abdel Karim
Computer Engineering Department



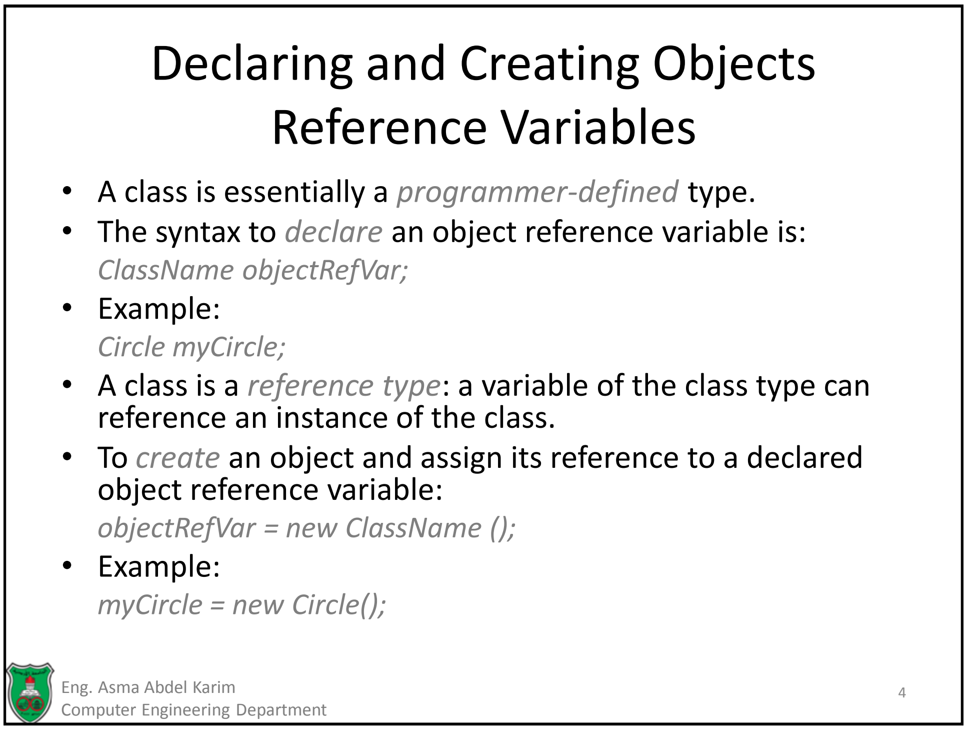
Example: Circle Class

```
class Circle{
   double radius;
   void setRadius (double newRadius){
        radius = newRadius;
   }
}
```

Class Circle has one variable of type double called radius.

Class Circle has one void method called setRadius which takes one double parameter and assign it to the variable radius.

Eng. Asma Abdel Karim
Computer Engineering Department
3



Declaring and Creating Objects Reference Variables

- A class is essentially a *programmer-defined* type.
- The syntax to *declare* an object reference variable is:
  ClassName objectRefVar;
- Example:
  Circle myCircle;
- A class is a *reference type*: a variable of the class type can reference an instance of the class.
- To *create* an object and assign its reference to a declared object reference variable:
  objectRefVar = new ClassName ();
- Example:
  myCircle = new Circle();

Eng. Asma Abdel Karim
Computer Engineering Department
4

Declaring and Creating Objects Reference Variables (Cont.)

- A single statement can be used to combine 1) the declaration of an object reference variable, 2) the creation of an object, and 3) the assigning of an object reference to the variable as follows:

  ClassName objectRefVar = new ClassName();

- Example:

  Circle myCircle = new Circle ();

Accessing an Object's Members

- In OOP, object's members are its *data fields* and *methods*.
- An object's data can be accessed and its methods invoked using the *dot operator (.)*.
- To reference a data field in an object:
  - *objectRefVar.dataField*
- Example:
  - *myCircle.radius*
- To invoke a method on an object:
  - *objectRefVar.method(arguments)*
- Example:
  - *myCircle.setRadius(5);*

Example: TestCircle.java

Only one class in a file can be a public class.

```
public class TestCircle{
     public static void main (String [] args){
          Circle circle1 = new Circle ();
          circle1.setRadius(5);
          System.out.println("The radius of this circle is
          "+circle1.radius);
     }
}
class Circle{
     double radius;
     void setRadius (double newRadius){
          radius = newRadius;
     }
}
```

The public class must have the same name as the file name.

Constructing Objects Using Constructors

- A *constructor* is invoked to create an object using the *new* operator.
- *Constructors* are a special kind of method.
- They have three peculiarities:
  - A constructor 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.
    - They play the role of initializing objects.

Constructing Objects Using Constructors (Cont.)

- A class may be defined without constructors.
- In this case, a *default constructor* is provided automatically:
  - A *default constructor* is a *public no-argument* constructor with an empty body which is implicitly defined in the class.
  - A *default constructor* is provided only if there are no other constructors explicitly defined in the class.

Example TestCircle.java Revisited

```
public class TestCircle{
    public static void main (String [] args){
        Circle circle1 = new Circle (5);
        System.out.println("The radius of this circle is   "+circle1.radius);
    }
}
class Circle{
    double radius;
    Circle (double initialRadius){
        radius = initialRadius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Constructors Overloading

- Like regular methods, constructors can be overloaded.
- Example:

```
class Circle{
    double radius;
    Circle(){          <-- Circle myFirstCircle = new Circle ();
        radius = 1;
    }
    Circle (double initialRadius){   <-- Circle mySecondCircle = new Circle(5);
        radius = initial Radius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Eng. Asma Abdel Karim
Computer Engineering Department
11

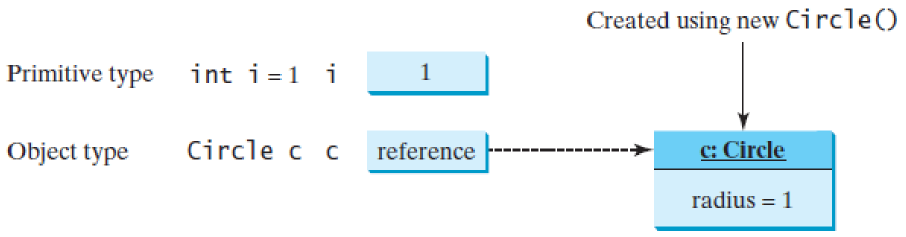
Reference Data Fields and the *null* Value

- Java assigns default values to data fields when an object is created.
  - *0* for *numeric* type.
  - *false* for a *boolean* type.
  - *\u0000* for a *char* type.
  - *Null* for a *reference* type.
    - *Null* is a special literal used for reference types.
- However, Java assigns no default value to a local variable inside a method.

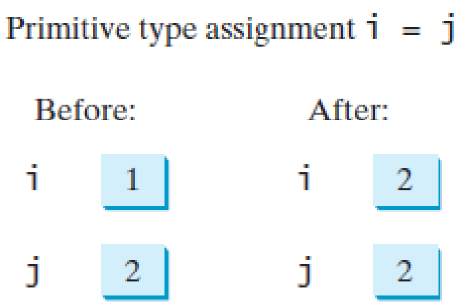
Eng. Asma Abdel Karim
Computer Engineering Department
12

Difference between Variables of Primitive Types and Reference Types

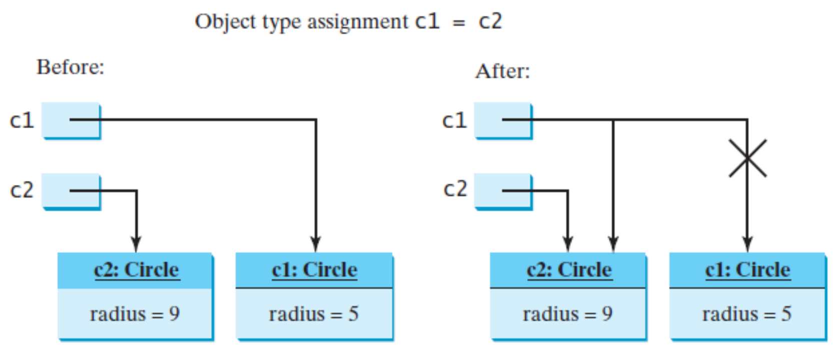
- Every variable represents a memory location that holds a value.
- A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.



Difference between Variables of Primitive Types and Reference Types (Cont.)



Difference between Variables of Primitive Types and Reference Types (Cont.)



Eng. Asma Abdel Karim
Computer Engineering Department

15

Static Variables, Constants, and Methods

- All variables declared in the data fields of the previous examples are called *instance variables*.
- An *instance variable* is tied to a specific instance of the class.
  - It is not shared among objects of the same class.
  - It has independent memory storage for each instance.
- In the following example, the *radius* of the first object "*circle1*" is independent of the *radius* of the second object "*circle2*":

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

Eng. Asma Abdel Karim
Computer Engineering Department

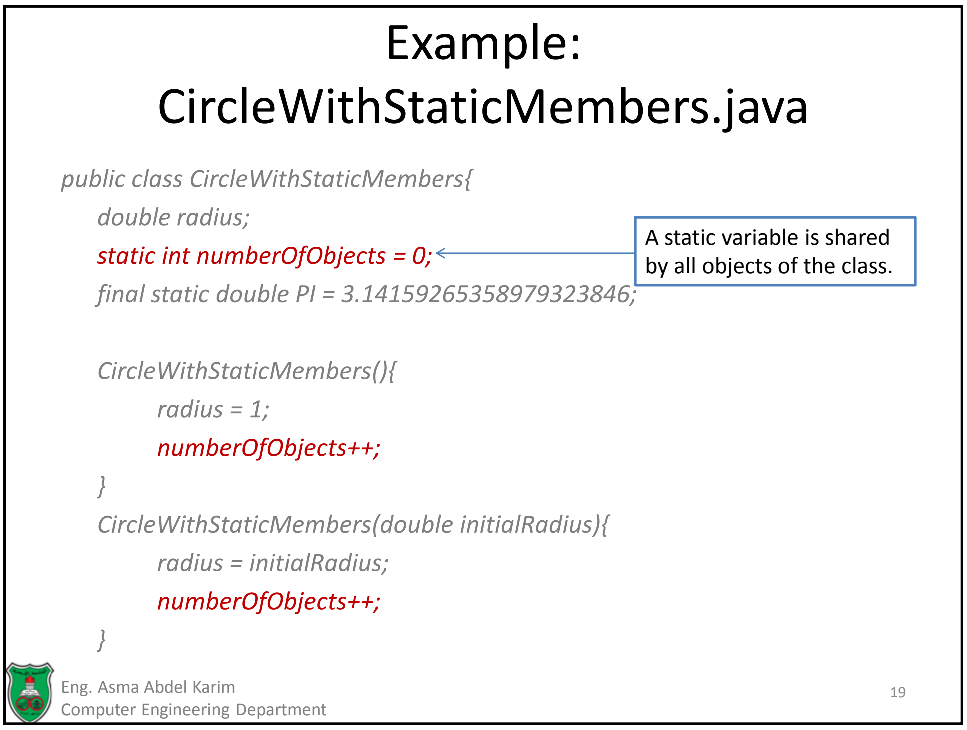
16

Static Variables, Constants, and Methods (Cont.)

- *Static variables*, also known as *class variables*, store values for the variables in a common memory location.
  - A *static variable* is used when it is wanted that all instances of the class to share data.
  - If one instance of the class changes the value of a static variables, all instances of the same class are affected.
- Static methods can be called without creating an instance of the class.

Static Variables, Constants, and Methods (Cont.)

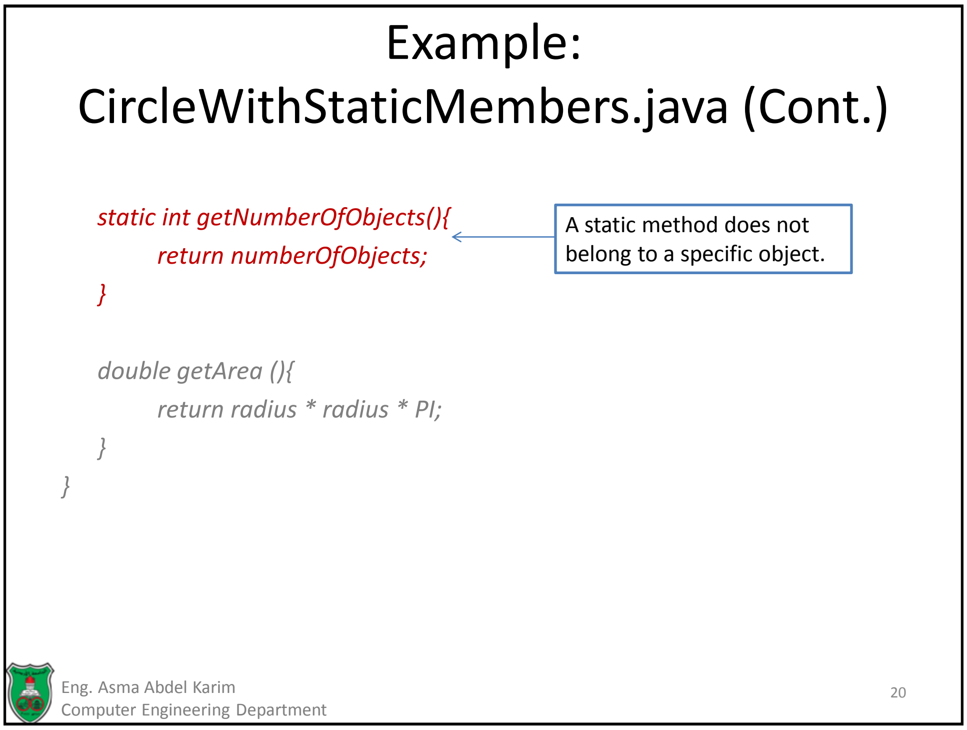
- To declare a static variable or define a static method, put the modifier *static* in the variable or method declaration.
- Since constants in a class are shared by all objects of the class, they should be declared static.
  - *final static double PI = 3.14159265358979323846;*
- Static variables and methods can be accessed from a reference variable or from their class name.



Example: CircleWithStaticMembers.java

```
public class CircleWithStaticMembers{
    double radius;
    static int numberOfObjects = 0;
    final static double PI = 3.14159265358979323846;

    CircleWithStaticMembers(){
        radius = 1;
        numberOfObjects++;
    }
    CircleWithStaticMembers(double initialRadius){
        radius = initialRadius;
        numberOfObjects++;
    }
```

A static variable is shared by all objects of the class.



Example: CircleWithStaticMembers.java (Cont.)

```
    static int getNumberOfObjects(){
        return numberOfObjects;
    }

    double getArea (){
        return radius * radius * PI;
    }
}
```

A static method does not belong to a specific object.

Example: TestCircleWithStaticMembers.java

Static variable accessed from its class name

Static variable accessed from a reference variable.

```
public class TestCircleWithStaticMembers{
    public static void main (String [] args){
        System.out.println("Before creating objects");
        System.out.println("The number of circle objects is " +
        CircleWithStaticMembers.numberOfObjects);

        CircleWithStaticMembers c1 = new CircleWithStaticMembers();

        System.out.println("After creating c1");
        System.out.println("c1 radius (" + c1.radius + ") and number of circle
        objects (" + c1.numberOfObjects + ")" );
```

Eng. Asma Abdel Karim
Computer Engineering Department

Example: TestCircleWithStaticMembers.java (Cont.)

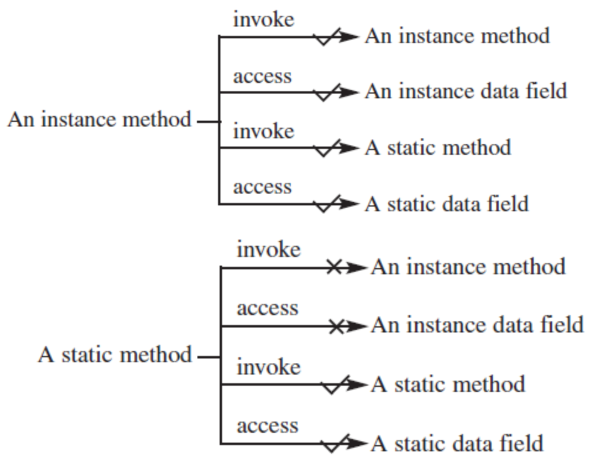
```
        CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);

        c1.radius = 9;

        System.out.println("After creating c2 and modifying c1");
        System.out.println("c1 radius (" + c1.radius + ") and number of circle
        objects (" + c1.numberOfObjects + ")" );
        System.out.println("c2 radius (" + c2.radius + ") and number of circle
        objects (" + c2.numberOfObjects + ")" );
    }
}
```

Eng. Asma Abdel Karim
Computer Engineering Department

Example: TestCircleWithStaticMembers.java (Output)

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

Relationship between Static and Instance Members


An instance method
invoke ✓ An instance method
access ✓ An instance data field
invoke ✓ A static method
access ✓ A static data field
A static method
invoke ✗ An instance method
access ✗ An instance data field
invoke ✓ A static method
access ✓ A static data field

Visibility Modifiers

- *Visibility modifiers* can be used to specify the *visibility* of a class and its members.
- A *visibility modifier* specifies how data fields and methods in a class can be accessed from outside the class.
  - There is no restriction on accessing data fields and methods from inside the class.

Visibility Modifiers: The Default

- If no visibility modifier is used, then by *default* the classes, methods, and data fields are accessible by any class in the same package.
  - This is known as *package-private* or *package-access*.
- *Packages* are used to organize classes. To do so, you need to add the following statement as the first statement in the program.
  - *package packageName;*
- If a class is defined without the package statement, it is said to be placed in the default package.

Visibility Modifiers: Public and Private

- The *public* modifier can be used for classes, methods and data fields to denote that they can be accessed from any other classes.
- The *private* modifier makes methods and data fields accessible only from within its own class.

Visibility Modifiers: Methods and Data Fields Example

```
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();
  }
}
```

Visibility Modifiers: Classes Example

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

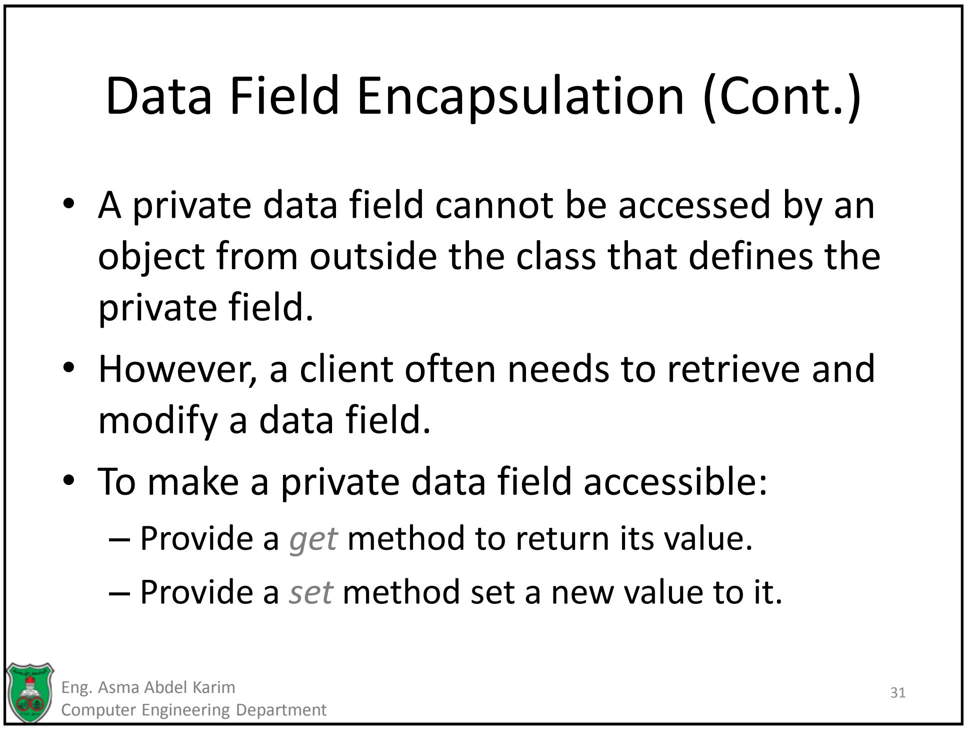
Visibility Modifiers: Another Example

```
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.

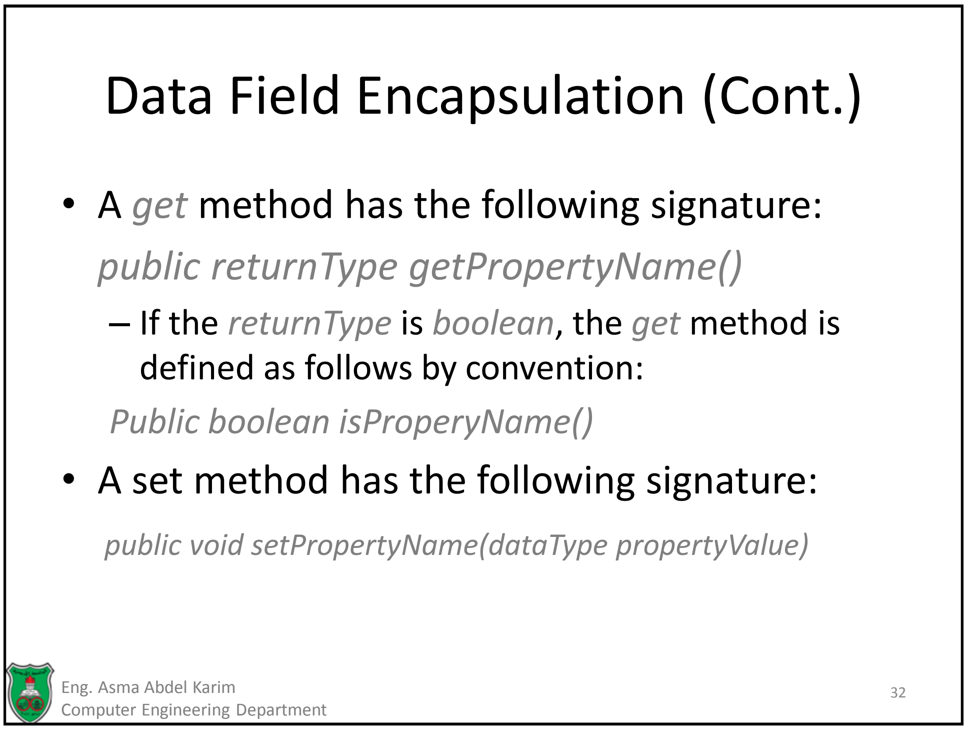


Data Field Encapsulation (Cont.)

- A private data field cannot be accessed by an object from outside the class that defines the private field.
- However, a client often needs to retrieve and modify a data field.
- To make a private data field accessible:
  - Provide a *get* method to return its value.
  - Provide a *set* method set a new value to it.

Eng. Asma Abdel Karim
Computer Engineering Department

31



Data Field Encapsulation (Cont.)

- A *get* method has the following signature:

  public returnType getPropertyName()
  - If the *returnType* is *boolean*, the *get* method is defined as follows by convention:

  Public boolean isProperyName()
- A set method has the following signature:

  public void setPropertyName(dataType propertyValue)

Eng. Asma Abdel Karim
Computer Engineering Department

32

Example: CircleWithPrivateDataFields.java

```
public class CircleWithPrivateDataFields{
    private double radius = 1;
    private static int numberOfObjects = 0;
    final static double PI = 3.14159265358979323846;

    public CircleWithPrivateDataFields(){
        numberOfObjects++;
    }
    public CircleWithPrivateDataFields(double initialRadius){
        radius = initialRadius;
        numberOfObjects++;
    }
```

Example: CircleWithPrivateDataFields.java (Cont.)

```
    public double getRadius(){
        return radius;
    }
    public void setRadius (double newRadius){
        radius = (newRadius>=0) ? newRadius : 0;
    }
    public static int getNumberOfObjects(){
        return numberOfObjects;
    }
    public double getArea(){
        return radius * radius * PI;
    }
}
```

These two methods are the only way to access the *radius.*

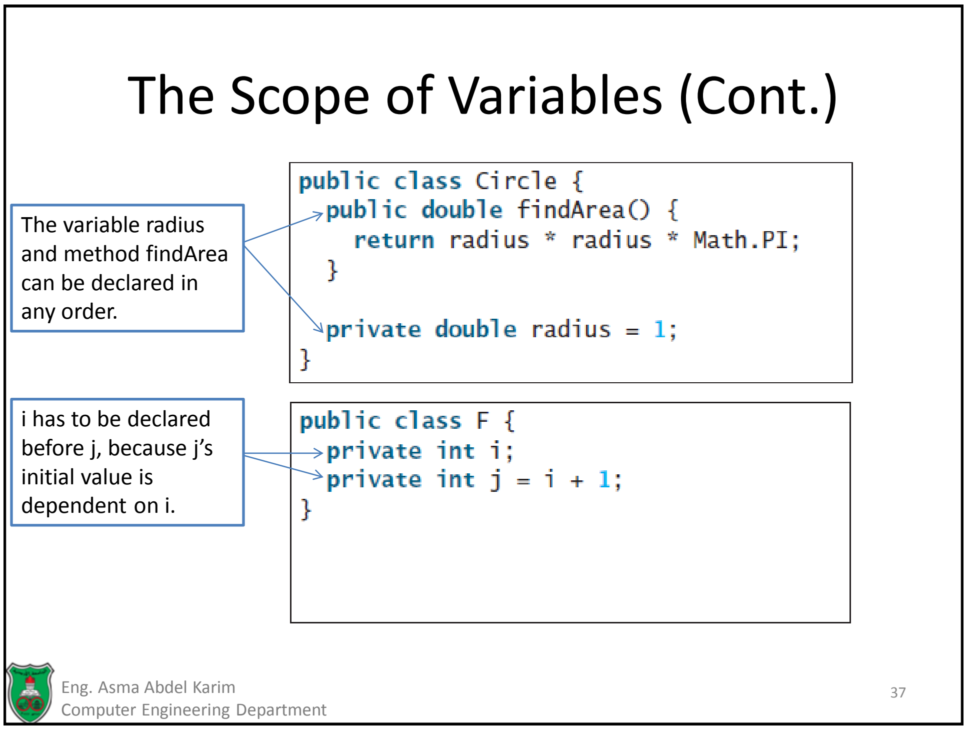
This method is the only way to read the *numberOfObjects*. *numberOfObjects* is only modified when a new object is created, there is no other way to modify it.

Example: TestCircleWithPrivateDataFields.java (Cont.)

```
public class TestCircleWithPrivateDataFields{
    public static void main (String [] args){
            CircleWithPrivateDataFields c1 = new
            CircleWithPrivateDataFields(5);
            System.out.println("The area of the circle of radius " +
            c1.getRadius() + "is " + c1.getArea());

            c1.setRadius(c1.getRadius()*1.1);
             System.out.println("The area of the circle of radius " +
            c1.getRadius() + "is " + c1.getArea());

            System.out.println("Number of circles created is " +
            CircleWithPrivateDataFields.getNumberOfObjects());
    }
}
```

The Scope of Variables

- The scope of a *class's variables* or *data fields* is the *entire class*, regardless of where the variables are declared.
- A class's variables and methods can appear in any order in the class.
  - The exception is when a data field is initialized based on a reference to another data field.



The Scope of Variables (Cont.)

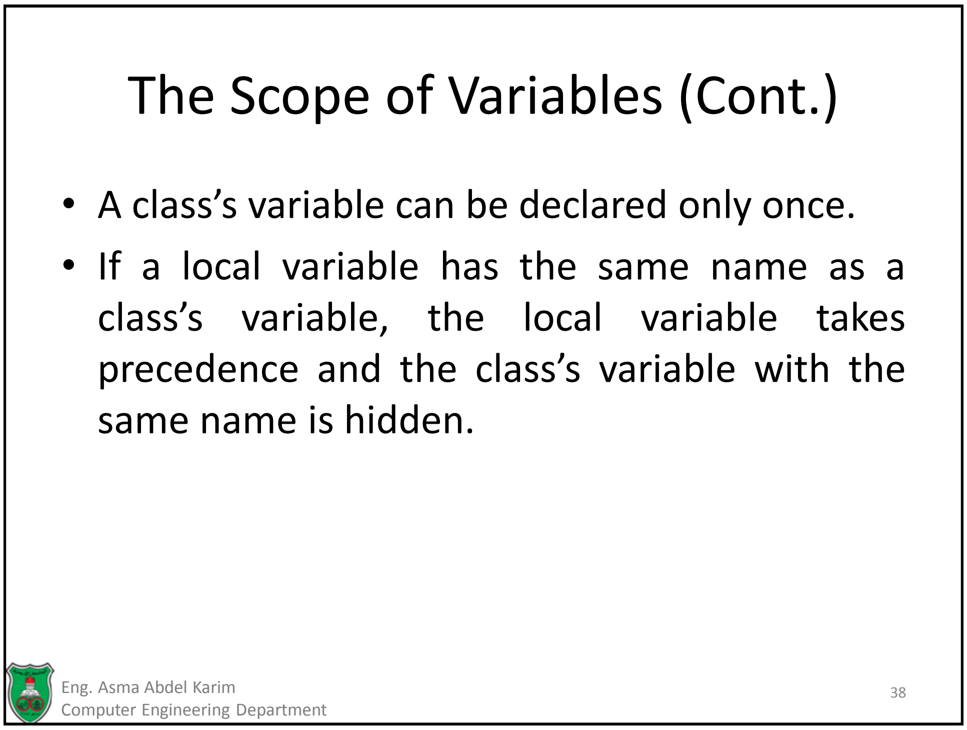
The variable radius and method findArea can be declared in any order.

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

  private double radius = 1;
}
```

i has to be declared before j, because j's initial value is dependent on i.

```
public class F {
  private int i;
  private int j = i + 1;
}
```

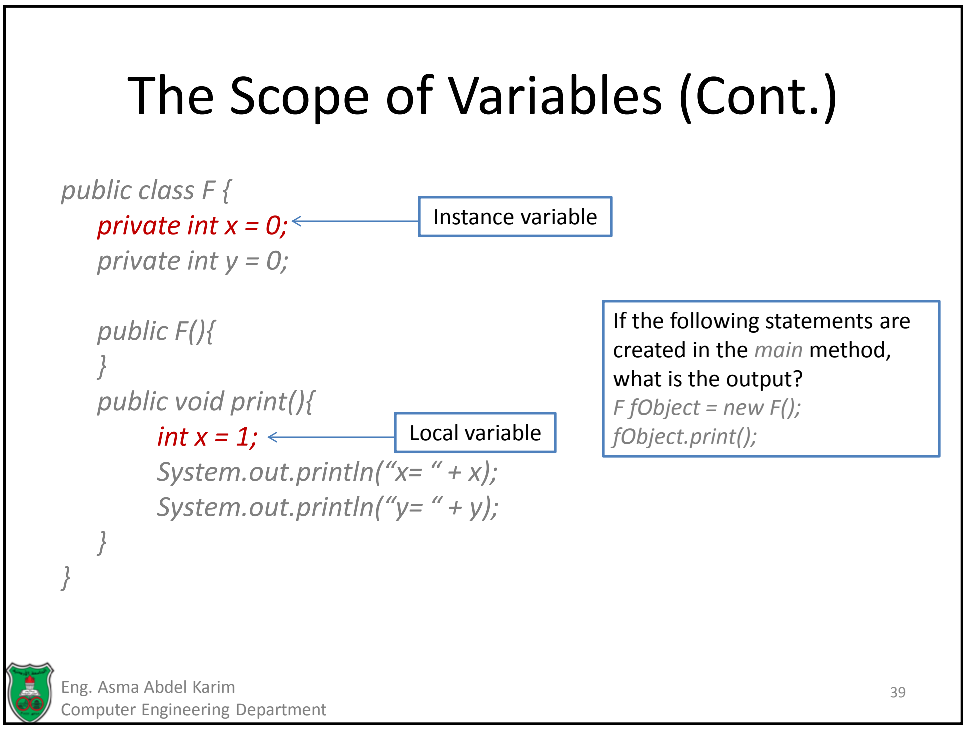
Eng. Asma Abdel Karim
Computer Engineering Department 37



The Scope of Variables (Cont.)

- A class's variable can be declared only once.
- If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is hidden.

Eng. Asma Abdel Karim
Computer Engineering Department 38



The Scope of Variables (Cont.)

```
public class F {
    private int x = 0;
    private int y = 0;

    public F(){
    }
    public void print(){
        int x = 1;
        System.out.println("x= " + x);
        System.out.println("y= " + y);
    }
}
```

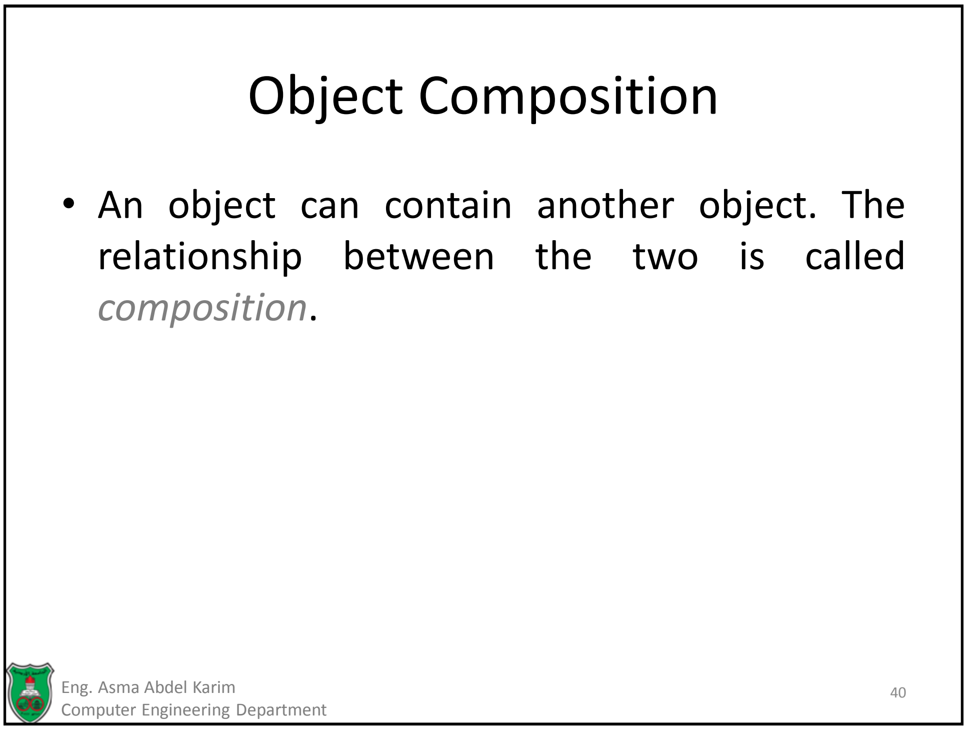
Instance variable

Local variable

If the following statements are created in the *main* method, what is the output?
F fObject = new F();
fObject.print();

Eng. Asma Abdel Karim
Computer Engineering Department

39



Object Composition

- An object can contain another object. The relationship between the two is called *composition*.

Eng. Asma Abdel Karim
Computer Engineering Department

40

Object Composition (Cont.)

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

Aggregated class

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

  ...
}
```

Aggregating class

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

Aggregated class

Passing Objects to Methods: Example

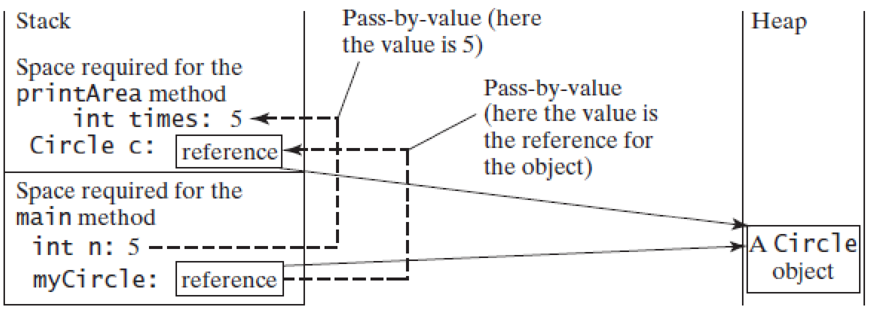
```
public class TestPassObject{
      public static void main(String args[]){
            CircleWithPrivateDataFields myCircle =  new CircleWithPrivateDataFields(1);
            int n = 5;
            printAreas (myCircle , n);
            System.out.println(“Radius is “ + myCircle.getRadius());
            System.out.println(“n is ” + n);
      }
      public static void printAreas (CircleWithPrivateDataFields 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);
                time--;
            }
      }
}
```

Passing an object to a method is to pass the reference of the object.

Passing Objects to Methods: Example (Cont.)

```
Radius      Area
1.0         3.141592653589793
2.0         12.566370614359172
3.0         29.274333882308138
4.0         50.26548245743669
5.0         79.53981633974483
Radius is 6.0
n is 5
```

Passing Objects to Methods: Example (Cont.)


Stack
Space required for the printArea method
int times: 5
Circle c: reference
Space required for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

Array of Objects

- An *array* can hold *objects* as well as primitive type values.
- Example:
  - In order to *declare* an array of ten Circle Objects:

    Circle [] circleArray = new Circle [10];
  - In order to *initialize* objects of this array;

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

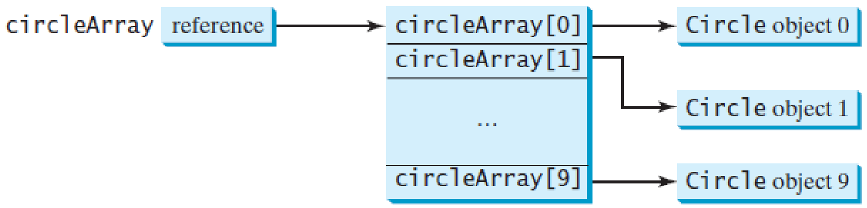
    circleArray[i] = new Circle();

    }

Array of Objects (Cont.)

- An array of objects is actually an array of reference variables.
- Example:

  CircleArray[1].getArea() involves two levels of referencing.



Array of Objects: Example

```
public class CircleArrayArea{
    public static void main (String [] args){
          CircleWithPrivateDataFields [] circleArray;
          circleArray = createCircleArray();
          printCircleArray (circleArray);
    }
    public static CircleWithPrivateDataFields [] createCircleArray(){
          CircleWithPrivateDataFields [] circleArray = new
          CircleWithPrivateDataFields [5];
          for (int i=0; i < circleArray.length; i++){
             circleArray[i] = new CircleWithPrivateDataFields (i+1);
          }
          return circleArray;
     }
```

Array of Objects: Example (Cont.)

```
     Public static void printCircleArea (CircleWithPrivateDataFields []
        circleArray){
        System.out.println("Radius \t\tArea");
        for (int i=0; i < circleArray.length; i++){
               System.out.println (circleArray[i].getRadius()+"
               \t\t"+circleArray[i].getArea());
        }
     }
}
```

Immutable Objects and Classes

- Normally, you create an object and allow its contents to be changed later.
- However, occasionally it is desirable to create an object <u>whose contents cannot be changed once the object has been created</u>.
  - Such an object is called *immutable object* and its class is called *immutable class*.

Immutable Objects and Classes (Cont.)

- For a class to be immutable, it must meet the following requirements:
  - All data fields must be private.
  - There can't be any mutator methods for data fields.
  - No accessor methods can return a reference to a data field that is mutable.

Immutable Objects and Classes: Example

```
public class Student{
    private int id;
    private String name;
    private double [] grades;

    public Student (int ssn, String newName){
        id = ssn;
        name = newName;
        grades = new double [3];
    }
    public int getId(){
        return id;
    }
    public String getName(){
        return name;
    }
    public double [] getGrades(){
        return grades;
    }
}
```

This method actually returns a reference to the array *grades*, which means it can be changed once returned.

Immutable Objects and Classes: Example (Cont.)

```
public class test {
    public static void main(String [] args){
        Student student = new Student (112233,  "John");
        double [] G = student.getGrades();
        G[0] = 90.0;
        G[1] = 95.5;
        G[2] = 92.9;
    }
}
```

The *this* Reference

- The keyword *this* refers to the object itself.

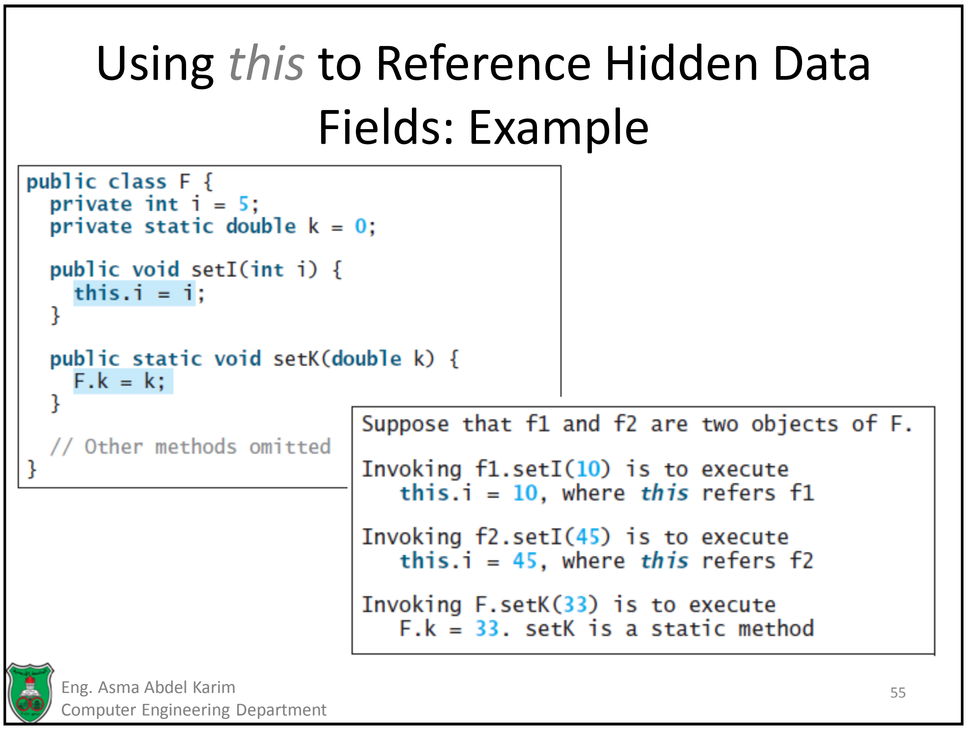
```
public class Circle{
    private double radius;
    ……
    public double getRadius(){
        return this.radius;
    …..
}
```

Equivalent

```
public class Circle{
    private double radius;
    ……
    public double getRadius(){
        return radius;
    …..
}
```

Using *this* to Reference Hidden Data Fields

- The *this* keyword can be used to reference a class's hidden data fields.
- A hidden *static variable* can be accessed simply by using the *ClassName.staticVariable*.
- A hidden *instance variable* can be accessed by using the keyword *this*.



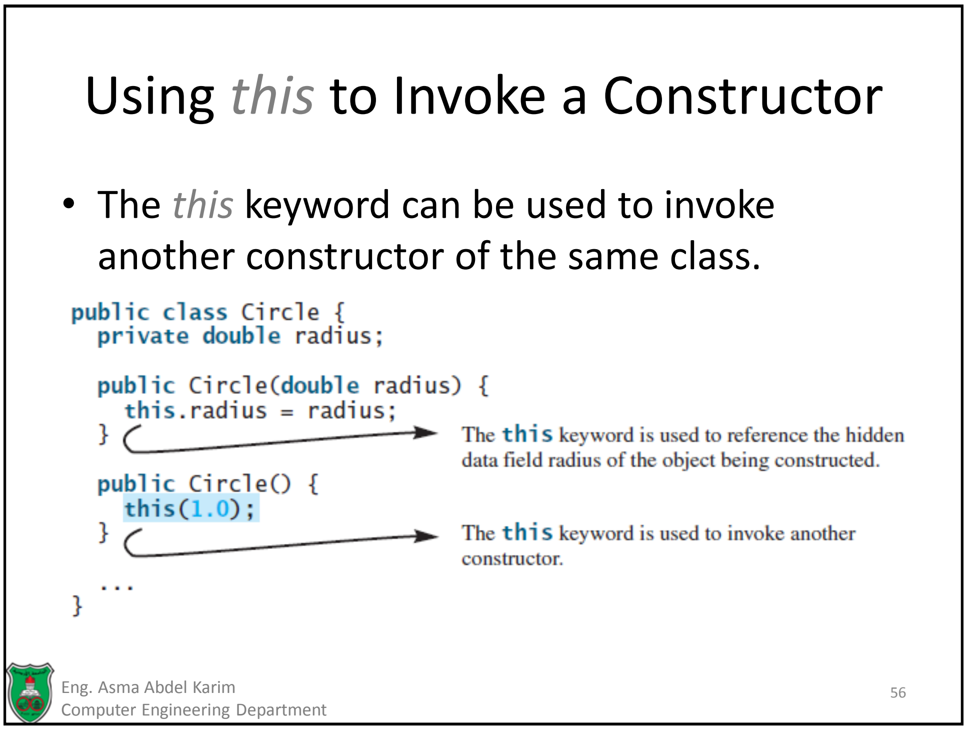
Using *this* to Reference Hidden Data Fields: Example

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

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

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

  // Other methods omitted
}
```

```
Suppose that f1 and f2 are two objects of 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

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

Eng. Asma Abdel Karim
Computer Engineering Department

55



Using *this* to Invoke a Constructor

- The *this* keyword can be used to invoke another constructor of the same class.

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

  ...
}
```

The **this** keyword is used to reference the hidden data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

Eng. Asma Abdel Karim
Computer Engineering Department

56

Networks and Internet Programming

Java Revision – Part III

Eng. Asma Abdel Karim
Computer Engineering Department

1

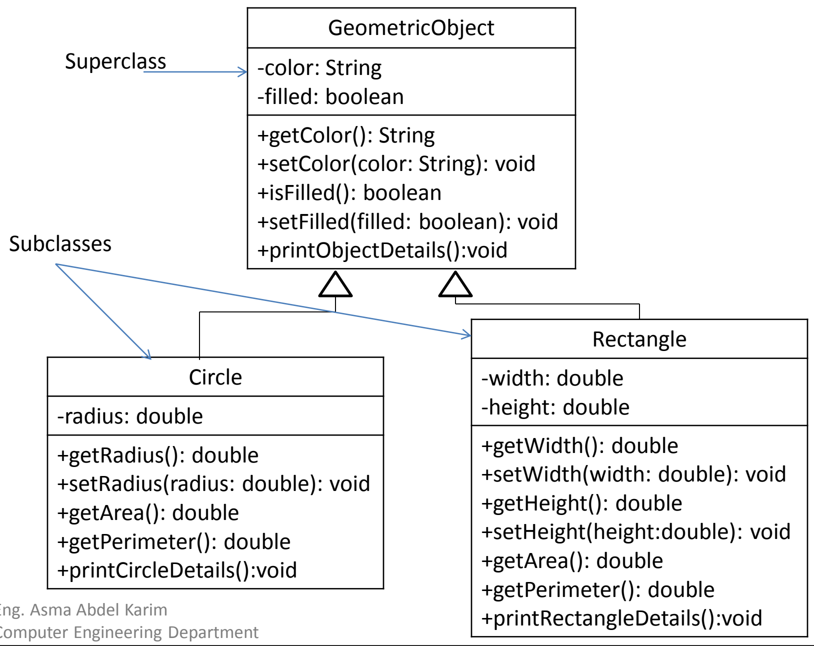
Inheritance
Superclasses and Subclasses

- A class (A) that is extended from another class (B) is called a *subclass*. (B) is called a *superclass*.
- A subclass:
  - Inherits accessible data fields and methods from its superclass.
  - May also add new data fields and methods.

Eng. Asma Abdel Karim
Computer Engineering Department

2

Superclasses and Subclasses: Example



GeometricObject.java

```
public class GeometricObject{
   private String color = "White";
   private boolean filled;
   public String getColor(){
         return color;
   }
   public void setColor(String color){
         this.color = color;
   }
```

GeometricObject.java (Cont.)

```
    public boolean isFilled(){
        return filled;
    }
    public void setFilled(boolean filled){
        this.filled = filled;
    }
    public void printObjectDetails(){
        System.out.println("Color is "+color + "Is filled? "+filled);
     }
}
```

Circle.java

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

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

Circle.java (Cont.)

```
    public double getPerimeter(){
        return 2 * radius * Math.PI;
    }
    public void printCircleDetails(){
        System.out.println("Circle color is "+ getColor() + " , circle
        filled? "+ isFilled()+ " , circle radius is "+ radius);
    }
}
```

Rectangle.java

```
public class Rectangle extends GeometricObject{
    private double width = 1;
    private double height = 1;

    public double getWidth(){
        return width;
    }
    public void setWidth(double width){
        this.width = width;
    }
    public double getHeight(){
        return height;
    }
```

Rectangle.java (Cont.)

```
    public void setHeight(double height){
            this.height = height;
    }
    public double getArea(){
            return width * height;
    }
    public double getPerimeter(){
            return 2 * (width + height);
    }
    public void printRectangleDetails(){
            System.out.println("Rectangle color is "+ getColor() + " , rectangle   filled?
            "+ isFilled()+ " , rectangle width is "+ width + " , rectangle height is " +
            height);
    }
}
```

Eng. Asma Abdel Karim
Computer Engineering Department
9

TestCircleRectangle.java

```
public class TestCircleRectangle {
   public static void main(String[] args) {
     Circle c = new Circle ();
     c.printObjectDetails();
     c.setColor("Black");
     c.setFilled(true);
     c.setRadius(5);
     c.printCircleDetails();
     Rectangle r = new Rectangle ();
     r.setColor("Red");
     r.setFilled(true);
     r.setWidth(3);
     r.setHeight(5);
     r.printRectangleDetails();
   }
}
```

These methods are inherited from the GeometricObject class.

These methods are inherited from the GeometricObject class.

Eng. Asma Abdel Karim
Computer Engineering Department
10

The *Super* Keyword

- The keyword *super* refers to the superclass and can be used to:
  - Call a superclass constructor.
  - Call a superclass method.

Eng. Asma Abdel Karim
Computer Engineering Department

11

Using the *Super* Keyword to Call a Superclass Constructor

- The syntax to call a superclass's constructor is:
  - *super();* *// to invoke the no-arg constructor*
  - *super(parameters);* *//to invoke a constructor with parameters*
- The statement *super()* or *super(parameters)* must appear in the first line of the subclass's constructor.
- The following constructor can be added to the *Circle* class of the previous example:

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

Invokes the no-arg constructor, which is the default constructor of the GeometricObject class.

Eng. Asma Abdel Karim
Computer Engineering Department

12

Constructor Chaining

- A constructor may invoke an overloaded constructor (using *this*) or its superclass constructor (using *super*).
- If neither is invoked explicitly, the compiler automatically puts *super()* as the first statement in the constructor.

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

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

Equivalent

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

Caution!!

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.
- Example: this code cannot be compiled:

```
public class Apple extends Fruit{

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

The default no-arg constructor of Apple will try to invoke a no-arg constructor of Fruit, which does not exist!

Using the *Super* Keyword to Call a Superclass Method

- The keyword *super* can be used to reference a method other than the constructor in the superclass. The syntax is:
  - ***super.method(parameters);***
- The printCircleDetails method in the *Circle* class could be rewritten as follows:

```
public void printCircleDetails(){
        System.out.println("Circle color is "+ super.getColor() + " ,      circle
        filled? "+ super.isFilled()+ " , circle radius is "+ radius);
}
```

- It is not necessary to put the super keyword before the methods *getcolor* and *isFilled* in the previous example.
  - These methods are inherited by the Circle class.
  - Cases were the *super* keyword is needed to invoke the superclass methods will be showed when methods overriding is introduced.

Overriding Methods

- A subclass inherits methods from a superclass.
- Sometimes, it is necessary for the subclass to modify the implementation of a method defined in the superclass.
- In the previous example, the method *printObjectDetails* of the *GeometricObject* class can be overridden in the *Circle* class as follows:

```
public void printObjectDetails(){
      super.printObjectDetails();
      System.out.println("Circle radius = "+radius);
}
```

Overriding Methods (Cont.)

- 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.
- 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.
  - The hidden static methods can be invoked using the syntax SuperClassName.staticMethodName.

Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.
  - The method should be defined in the subclass using the same signature and the same return type.

Overriding vs. Overloading: Example

```java
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);
  }
}
```

```java
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);
  }
}
```

Overriding vs. Overloading: Notes

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

Override Annotation

- To avoid mistakes, you can use a special Java syntax, called *override annotation*:
  - Place @Override before the method in the subclass.
- This annotation denotes that the annotated method is required to override a method in the superclass.
  - If a method with this annotation does not override its superclass's method, the compiler will report an error.
- For example, in order to denote that the *printObjectDetails* is overridden in the *Circle* class:

```
@Override
public void printObjectDetails(){
    super.printObjectDetails();
    System.out.println("Circle radius = "+radius);
}
```

The Object Class and Its toString() Method

- If no inheritance is defined when a class is defined, the superclass of the class is *Object* by default.
  - Every class in Java is descended from the *java.lang.Object* class.
- For example the following two class definitions are the same:

```
public class ClassName {
  ...
}
```

Equivalent

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

The Object Class and Its toString() Method (Cont.)

- One of the most important methods provided by the *Object* class is the method *toString*.
- The signature of the *toString* method is:
  - ***public String toString()***
- Invoking *toString()* on an object returns a string that describes the object.
  - By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.
  - For example, the output of the following code is something like: Circle@780324ff

    Circle c = new Circle();

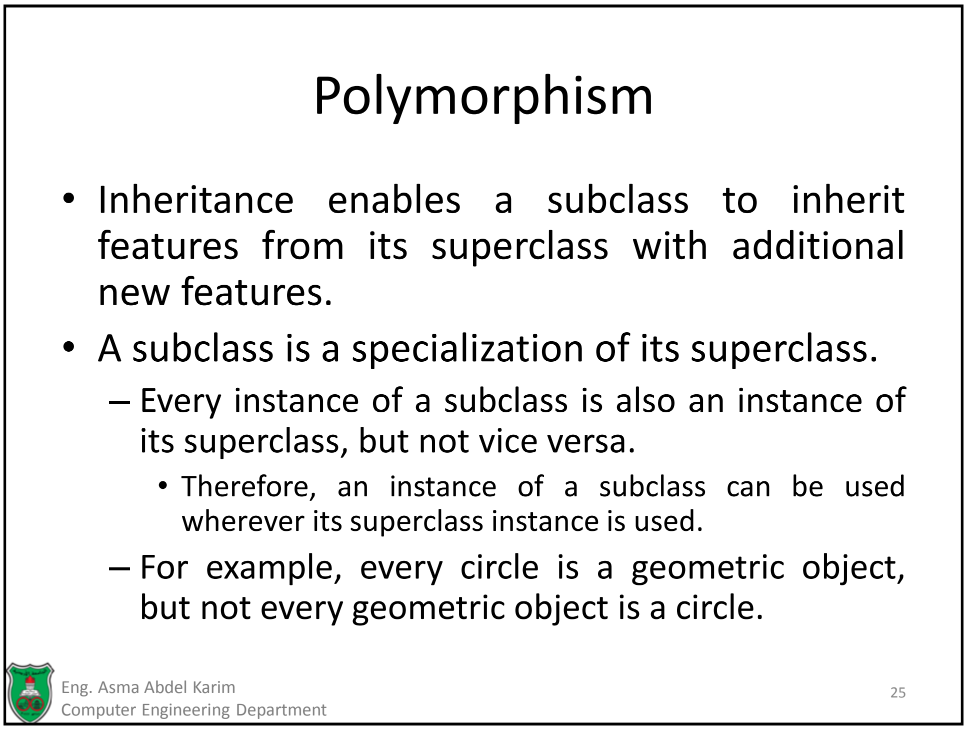
    System.out.println(c.toString());

The Object Class and Its toString() Method (Cont.)

- Usually, we override the *toString* method so that it returns a descriptive string representation of the object.
- For example, the *toString* method in the *Object* class can be overridden for the *Circle* class as follows:

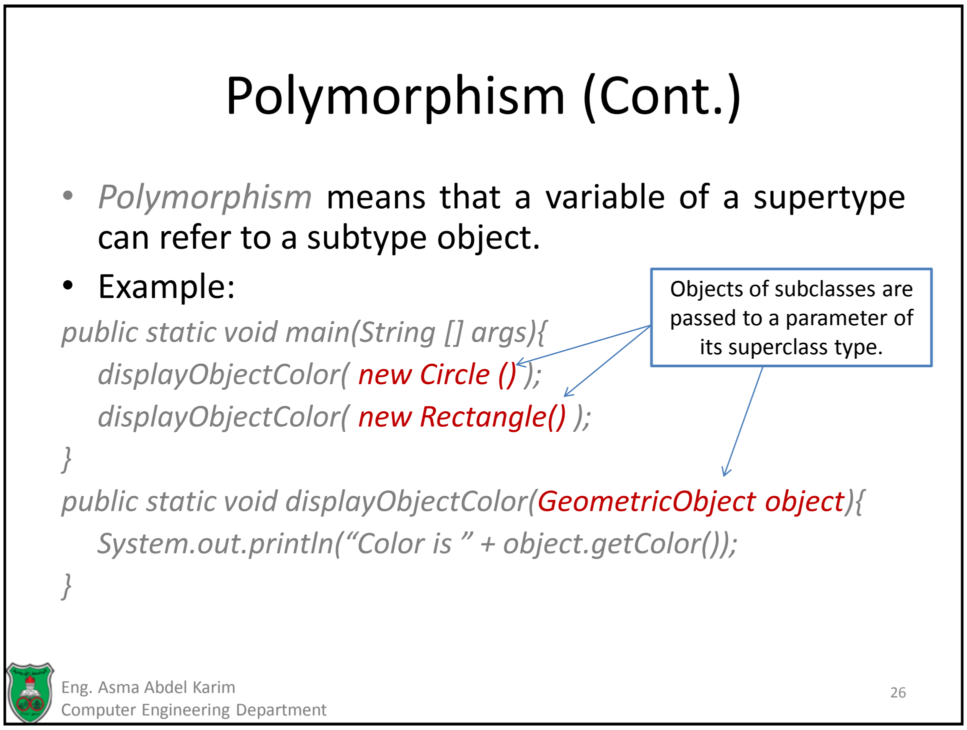
```
public String toString(){
    return "Color is " + getColor() + ". Is filled? " + isFilled() +
    ". Radius is " + radius + ".";
}
```

- You can also pass an object to invoke System.out.println(object) and System.out.print(object).
  - This is equivalent to invoking System.out.println(object.toString()) and System.out.print(object.toString()).



Polymorphism

- Inheritance enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass.
  - Every instance of a subclass is also an instance of its superclass, but not vice versa.
    - Therefore, an instance of a subclass can be used wherever its superclass instance is used.
  - For example, every circle is a geometric object, but not every geometric object is a circle.



Polymorphism (Cont.)

- *Polymorphism* means that a variable of a supertype can refer to a subtype object.
- Example:

```
public static void main(String [] args){
    displayObjectColor( new Circle () );
    displayObjectColor( new Rectangle() );
}
public static void displayObjectColor(GeometricObject object){
    System.out.println("Color is " + object.getColor());
}
```

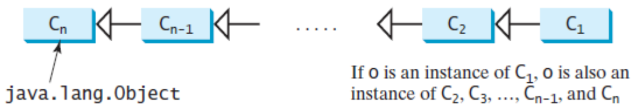
Objects of subclasses are passed to a parameter of its superclass type.

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain.
- The JVM decides which method is invoked at runtime.
  - This is known as *dynamic binding*.

Dynamic Binding (Cont.)

- Dynamic binding works as follows:
  - Suppose o is an instance of C_1, C_2, C_3,...., C_n. As follows:


C_n
C_{n-1}
.....
C_2
C_1
java.lang.Object
If o is an instance of C_1, o is also an instance of C_2, C_3, ..., C_{n-1}, and C_n

  - If o invokes a method p:
    - The JVM searches for the implementation of the method p in C_1, C_2, C_3,...., C_n, in this order, until it is found.
    - Once an implementation is found, the search stops and the first-found implementation is invoked.

Dynamic Binding (Cont.)

- Consider the following code:

Object o = new GeometricObject();
System.out.println(o.toString());

- There are two important terms in order to identify which *toString* method will be invoked by o:
  - ***Declared type***: the type that declares a variable.
    - **In the previous code example, o's declared type is *Object*.**
  - ***Actual type***: The actual class for the object referenced by the variable.
    - **In the previous code example, o's actual type is *GeometricObject*.**
- Which method is invoked is determined by the object actual type.
  - In other words, the JVM starts the search with the class that defines the actual type.

Dynamic Binding: Example

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

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

```
Student
Student
Person
java.lang.object@9fa8988
```

Dynamic Binding (Cont.)

- Matching a method signature and binding a method implementation are two separate issues:
  - The declared type of the reference variable decides which method to match at compile time.
    - The compiler finds a matching method according to the parameter type, number of parameters, and order of parameters.
  - The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

Casting Objects

- The compiler implicitly casts an instance of a subclass to a variable of a superclass.
  - Because an instance of a subclass is always an instance of its superclass.
  - Known as *upcasting*.
  - Example:

  Object o = new Student();

Casting Objects (Cont.)

- When casting an instance of a superclass to a variable of its subclass, explicit casting must be used:
  - To confirm your intention to the compiler; otherwise a compile error will occur.
  - Known as *downcasting*.
  - Example:

```
Object o = new Student();
Student b = o;              // causes compile error
Student b = (Student)o;  //does not cause compile error
```

Casting Objects (Cont.)

- In order for casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
  - If the superclass object is not an instance of the subclass, a runtime *ClassCastException* occurs.
- It is a good practice, to ensure that the object is an instance of another object before attempting a casting.
  - This can be accomplished using the *instanceof* operator.

The *instanceof* Operator

```
Object myObject = new Circle;
If (myObject instanceof Circle){
   System.out.println("The circle area is " +
   ((Circle)myObject).getArea());
}
```

The object member access operator (.) precedes the casting operator. Parentheses are used to ensure that casting is done before the (.) operator.

Why Casting is Necessary?

- In the previous example:
  - The variable *myObject* is declared *Object*.
  - The declared type decides which method to match <u>at compile time</u>.
    - Using *myObject.getArea()* would cause a compile error, because the *Object* class does not have the *getArea* method.
    - Therefore it is necessary to cast *myObject* into the *Circle* type to tell the compiler that *myObject* is also an instance of *Circle*.
  - Why not define, *myObject* as a *Circle* type in the first place?
    - To enable generic programming: it is a good practice to define a variable with a supertype, which can accept a value of any subtype.

Casting & Polymorphism: Example

```
public class Casting {
   public static void main(String[] args) {
      Object object1 = new Circle();
      Object object2 = new Rectangle();

      displayObject(object1);
      displayObject(object2);
   }
   public static void displayObject(Object object){
      if (object instanceof Circle){
         System.out.println("The circle radius is "+((Circle)object).getRadius());
      }
      if (object instanceof Rectangle){
         System.out.println("The rectangle width is "+((Rectangle)object).getWidth());
         System.out.println("The rectangle height is "+((Rectangle)object).getHeight());
      }
   }
}
```

The *Object's equals* Method

- The *equals* method is defined in the *Object* class.
- The *equals* method signature is:
  public boolean equals (Object o)
- The syntax for invoking the *equals* method is:
  object1.equals(object2)

The *Object's equals* Method (Cont.)

- The default implementation of the equals method in the Object class is:

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

- This implementation checks whether two reference variables point to the same object using the == operator.
- You should override this method in your custom class to test whether two distinct objects have the same content.

The *Object's equals* Method: Example

- The *equals* method can be overridden in the *Circle* class to compare whether two circles are equal based on their radius as follows:

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

- Note that when overriding the *equals* method, you should use the signature *equals(**Object** obj)* not *equals(someClassName o)* (e.g. *equals (Circle c)*).

Protected Class Members

- Remember that:
  - Private members can be accessed only from inside the class.
  - Public members can be accessed from any other classes.
- Often it is desirable:
  - To allow subclasses to access data fields or methods defined in the superclass.
  - But not to allow non-subclasses to access these data fields and methods.
- To accomplish this you can use the protected keyword (access modifier).

Protected Class Members (Cont.)

Visibility increases

$\longrightarrow$

private, default (no modifier), protected, public

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

Protected Class Members (Cont.)

- A subclass may override a method defined in its superclass and increase its accessibility.
  - For example, change its accessibility from protected (as defined in the superclass) to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.
  - For example, a method defined as public in the superclass should be defined as public in the subclass.

Preventing Extending and Overriding

- You can prevent a class from being extended using the *final* modifier.
- For example, the following class cannot be extended:

```
public final class A{
}
```

Preventing Extending and Overriding (Cont.)

- You can prevent a method from being overridden, by its subclasses, using the *final* modifier.
- For example the following method cannot be overridden:

```
public class Test{
      public final void m{
      //Do something
      }
}
```

Eng. Asma Abdel Karim
Computer Engineering Department
45

What are Abstract Methods?

- In the example of the previous section, *GeometricObject* was defined as the superclass for Circle and Rectangle.
- Bot Circle and Rectangle contain *getArea()* and *getPerimeter()* methods.
- It is better to define the *getArea()* and *getPerimeter()* methods in the *GeometricObject* class.
- However, these methods cannot be implemented in the *GeometricObject* class, because their implementation depends on the specific type of a geometric object.

Eng. Asma Abdel Karim
Computer Engineering Department
46

What are Abstract Methods? (Cont.)

- Such methods can be defined in the superclass as *abstract methods*.
  - An abstract method is defined without an implementation in the superclass.
    - Its implementation is provided by the subclasses.
  - Abstract methods are denoted using the *abstract* modifier in the method header.

Eng. Asma Abdel Karim
Computer Engineering Department

47

What are Abstract Classes?

- A class that contains at least one abstract method must be defined as an *abstract class*.
  - Abstract classes are denoted using the *abstract* modifier in the class header.
- Abstract classes are like regular classes, but you cannot create instances of abstract classes using the *new* operator.
  - Constructors of an abstract class are defined as protected, because they are used only by subclasses.

Eng. Asma Abdel Karim
Computer Engineering Department

48

Abstract Classes & Methods: Example

```
public abstract class GeometricObject{
    private String color = "White";
    private boolean filled;
    protected GeometricObject(String color, boolean filled){
        this.color=color;
        this.filled=filled;
    }
    public String getColor(){
        return color;
    }
    public void setColor(String color){
        this.color = color;
    }
```

Abstract Classes & Methods: Example (Cont.)

```
    public boolean isFilled(){
        return filled;
    }
    public void setFilled(boolean filled){
        this.filled = filled;
    }
    public void printObjectDetails(){
       System.out.println("Color is "+color + "Is filled? "+filled);
    }
    public abstract double getArea();
    public abstract double getPerimeter;
}
```

Why Abstract Methods? Example

```
public class testGeometricObject{
    public static void main(String args[]){
            GeometricObject g1 = new Circle(5);
            GeometricObject g2 = new Rectangle(5, 3);
            System.out.println("The two geometric objects           have
            the       same area?" + equalArea(g1, g2));
            displayGeometricObject(g1);
            displayGeometricObject(g2);
    }
    public static boolean equalArea(GeometricObject object1,
           GeometricObject object2){
              return object1.getArea() == object2.getArea();
    }
```

Eng. Asma Abdel Karim
Computer Engineering Department

51

Why Abstract Methods? Example

```
    public static void displayGeometricObject(GeometricObject
         object){
             System.out.println();
             System.out.println("The area is "+object.getArea());
            System.out.println("The perimeter is "+
            object.getPerimeter());
    }
}
```

Eng. Asma Abdel Karim
Computer Engineering Department

52

Important Notes Regarding Abstract Classes and Methods

- An abstract method cannot be contained in a non-abstract class.
  - If a subclass of an abstract superclass does not implement all abstract methods, the subclass must be defined as abstract.
  - Abstract methods are non-static.
- An abstract class cannot be instantiated using the new operator, but:
  - You still can define its constructors which are invoked in the constructors of its subclasses.
  - An abstract class can be used as a data type.
    - Therefore, the following statement, which creates an array whose elements are of the GeometricObject type is correct:

    GeometricObject [] Objects = new GeometricObject[10];

What is an Interface?

- An *interface* is a class-like construct that contains only constants and abstract methods.
- The intent of interfaces is to define common behavior for related or unrelated classes.
- An interface is treated like a special class in Java.
  - Each interface is compiled into a separate bytecode file.
  - You can use an interface more or less the same way you use an abstract class.
    - An interface can be used as a data type for a reference variable.
    - You cannot create an instance from an interface with the new operator.

How to Define an Interface?

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

```
modifier interface InterfaceName {
      //Constant declarations
      //Abstract method signatures
}
```

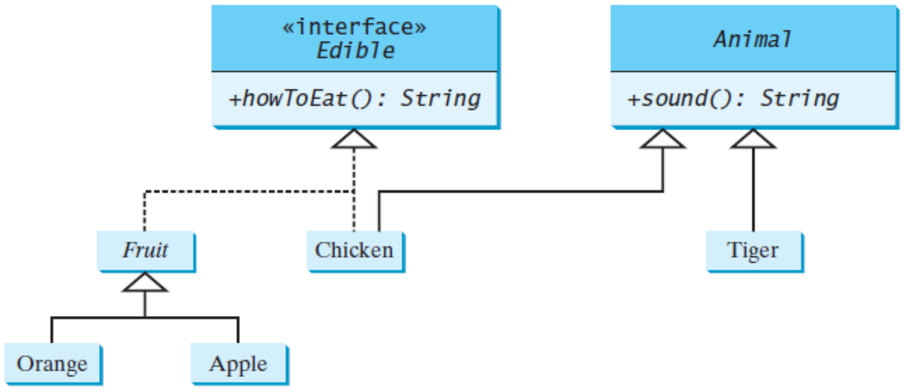
- Example:

```
public interface Edible{
      public abstract String howToEat();
}
```

Implementing an Interface

- You can use an interface to specify the behavior of an object, by letting the class for the object *implement* this interface using the *implements* keyword.
  - When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type.
- The relationship between an interface and a class that implements it is known as *interface inheritance*.
  - Since *interface inheritance* and *class inheritance* are essentially the same, both are referred to as *inheritance*.

Implementing an Interface: An Example (UML Diagram)



Implementing an Interface: An Example (Edible.java)

```
public interface Edible {
   public abstract String howToEat();
}
```

Implementing an Interface: An Example (Animal.java)

```
public abstract class Animal {
   public abstract String sound();
}
class Chicken extends Animal implements Edible{
   @Override
   public String howToEat(){
      return "Chicken: fry it";
   }
   @Override
   public String sound(){
      return "Chicken: Cluck";
   }
}
class Tiger extends Animal{
   @Override
   public String sound(){
      return "Tiger: RROAAAAAR";
   }
}
```

The *Animal* class is declared abstract since it has the abstract method *sound*.

Eng. Asma Abdel Karim
Computer Engineering Department

59

Implementing an Interface: An Example (Fruit.java)

```
public abstract class Fruit implements Edible{

}
class Apple extends Fruit{
   @Override
   public String howToEat(){
      return "Apple: make apple pie";
   }
}
class Orange extends Fruit{
   @Override
   public String howToEat(){
      return "Orange: make orange juice";
   }
}
```

The *Fruit* class does not implement the *howToEat* method. Hence it must be declared abstract.

Eng. Asma Abdel Karim
Computer Engineering Department

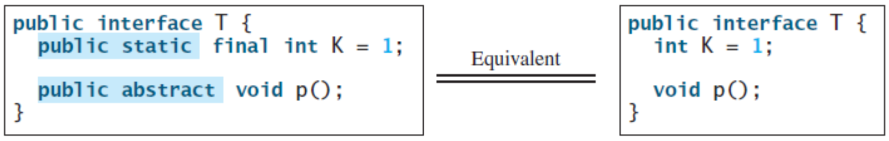
60

Implementing an Interface: An Example (TestEdible.java)

```
public class TestEdible {
  public static void main(String[] args) {
    Object [] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i=0; i< objects.length; i++){
      if (objects[i] instanceof Edible)
        System.out.println(((Edible)objects[i]).howToEat());

      if (objects[i] instanceof Animal)
        System.out.println(((Animal)objects[i]).sound());
    }
  }
}
```

A Note Regarding Interfaces

- All data fields of an interface are *public static final*.
- All methods of an interface are *public abstract*.
- Therefore, Java allows these modifiers to be omitted as follows:



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

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

Interfaces vs. Abstract Classes

	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.

More on Interfaces

- Java allows only *single inheritance* for class extension, but allow *multiple inheritance* for interface extension.
- For example:

public class NewClass extends BaseClass implements Interface1, …., InterfaceN {

}

More on Interfaces (Cont.)

- An interface can inherit other interfaces using the *extends* keyword.
  - Such an interface is called a sub-interface.
  - An interface can extend other interfaces but not classes.
- For example, NewInterface in the following code is a sub-interface of interface1, …., and interfaceN:

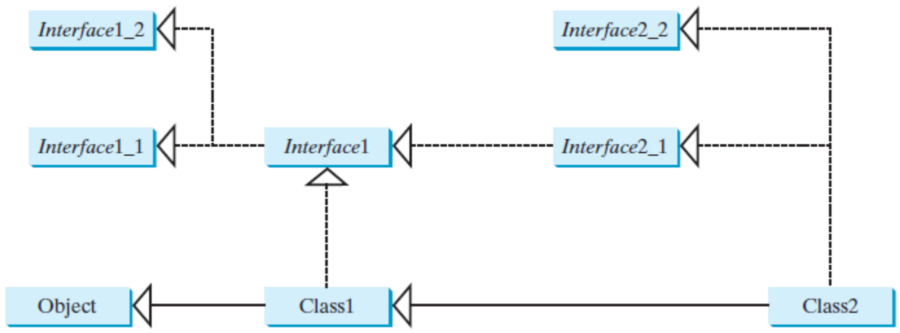
```
public interface NewInterface extends Interface1, …, InterfaceN{

}
```

  - A class implementing NewInterface must implement the abstract methods defined in NewInterface, Interface1, …., and InterfaceN.

More on Interfaces (Cont.)

- All classes share a single root, the *object* class, but there is no single root for interfaces.
- A variable of an interface type can reference any instance of the class that implements the interface.
  - If a class implements an interface, the interface is like a superclass for the class.

More on Interfaces (Cont.)



- Suppose c is an instance of class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Object-Oriented Problem Solving

Exception Handling

Based on Chapter 12 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Introduction (12.1)
- Exception Handling Overview (12.2)
- Exception Types (12.3)
- More on Exception Handling (12.4)
  - Declaring Exceptions (12.4.1)
  - Throwing Exceptions (12.4.2)
  - Catching Exceptions (12.4.3)
  - Getting Information from Exceptions (12.4.4)
  - Example: Declaring, Throwing, and Catching Exceptions (12.4.5)
- The *finally* Clause (12.5)
- When to use Exceptions? (12.6)
- Rethrowing Exceptions (12.7)
- Chained Exceptions (12.8)
- Defining Custom Exception Classes (12.9)

Introduction

- Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out.
  - If you access an array using an index that is out of bounds, you will get a runtime error with an *ArrayIndexOutOfBoundsException*.
  - If you enter a *double* value when your program expects an *integer*, you will get a runtime error with an *InputMismatchException*.
- In Java, runtime errors are thrown as *exceptions*.
- An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally.
- If the *exception* is not handled, the program will terminate abnormally.
  - *Exception handling* enables a program to deal with exceptional situations and continue its normal execution.

Exception Handling Overview Quotient.java

```java
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));
  }
}
```

```
Enter two integers: 5 2 ↵Enter
5 / 2 is 2
```

```
Enter two integers: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:11)
```

Exception Handling Overview (Cont.) QuotientWithIf.java

```java
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 ");
  }
}
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Exception Handling Overview (Cont.) QuotientWithMethod.java

```java
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

  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();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

Exception Handling Overview (Cont.) QuotientWithMethod.java (Output)

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Program is terminated if number2 equals 0.

Problem: what if the caller should decide whether to terminate the program!

Exception Handling Overview QuotientWithException.java

```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  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();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

If an ArithmeticException occurs

Exception Handling Overview QuotientWithException.java (Output)

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
Execution continues ...
```

```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

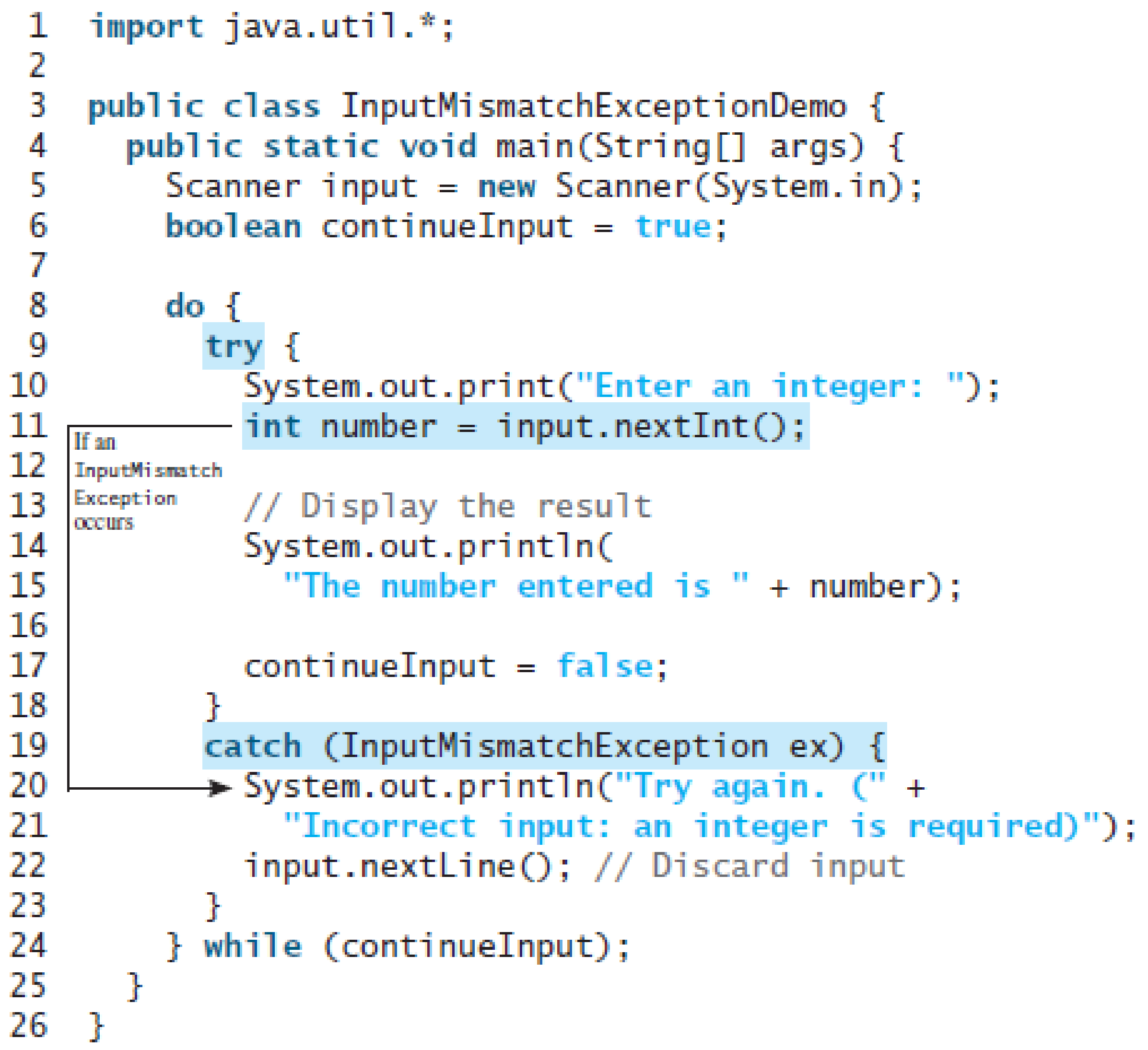
Exception Handling Overview

```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

Exception Handling Overview
Benefit of Exception Handling

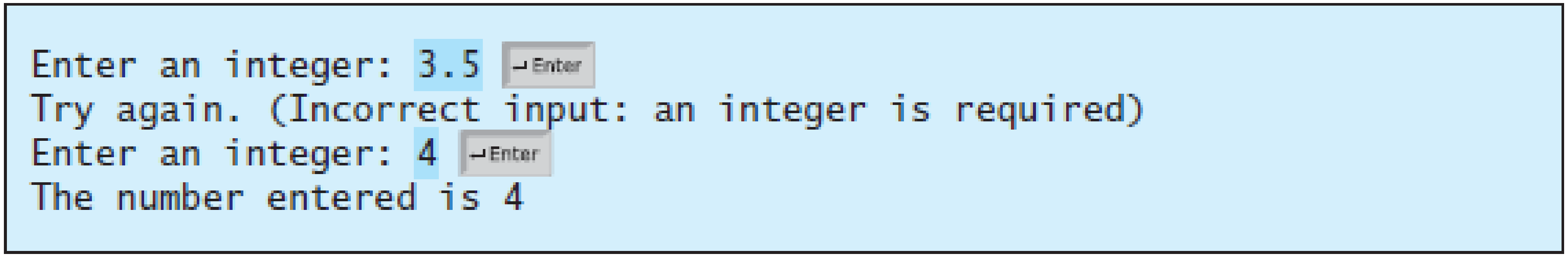
- The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).
  - Often the called method does not know what to do in case of error.
  - This is typically the case for the library methods.
    - The library method can detect the error, but only the caller knows what needs to be done when an error occurs.

Exception Handling Overview InputMismatchExceptionDemo.java



```java
import java.util.*;

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

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // Discard input
      }
    } while (continueInput);
  }
}
```

Exception Handling Overview
InputMismatchExceptionDemo.java (Output)

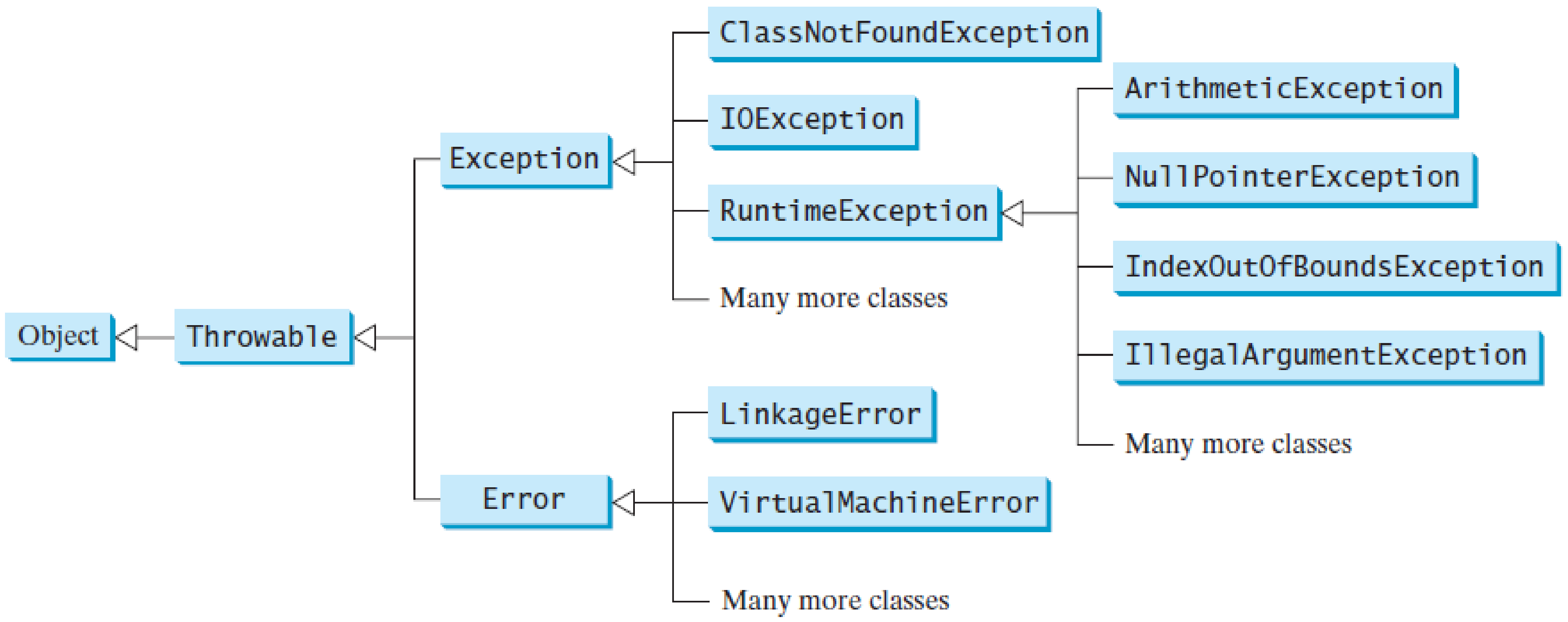


```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

Exception Types

- Exceptions are objects, and objects are defined using classes.
- The root class for all exceptions is *java.lang.Throwable*.
- There are many predefined exception classes in the Java API.
- You can also define your own exception classes.

Exception Types (Cont.)



Exception Types (Cont.)

- The *Throwable* class is the root of all exception classes.
  - All Java exception classes inherit directly or indirectly from *Throwable*.
  - You can create your own exception classes by extending *Exception* or a subclass of *Exception*.
- The exception classes can be classified into three major types:
  - System Errors.
  - Runtime Exceptions.
  - Other exceptions.

Exception Types: System Errors

- *System errors* are thrown by the JVM and are represented in the *Error* class.
- The *Error* class describes internal system errors, though such errors rarely occur.
  - If one occurs, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exception Types: Runtime Exceptions

- Runtime exceptions are represented in the RunTimeException class.
  - Describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method is passed an argument that is illegal or inappropriate.

Exception Types: Other Exceptions

- Other exceptions are represented in the *Exception* class.
  - Describes errors caused by your program and by external circumstances.

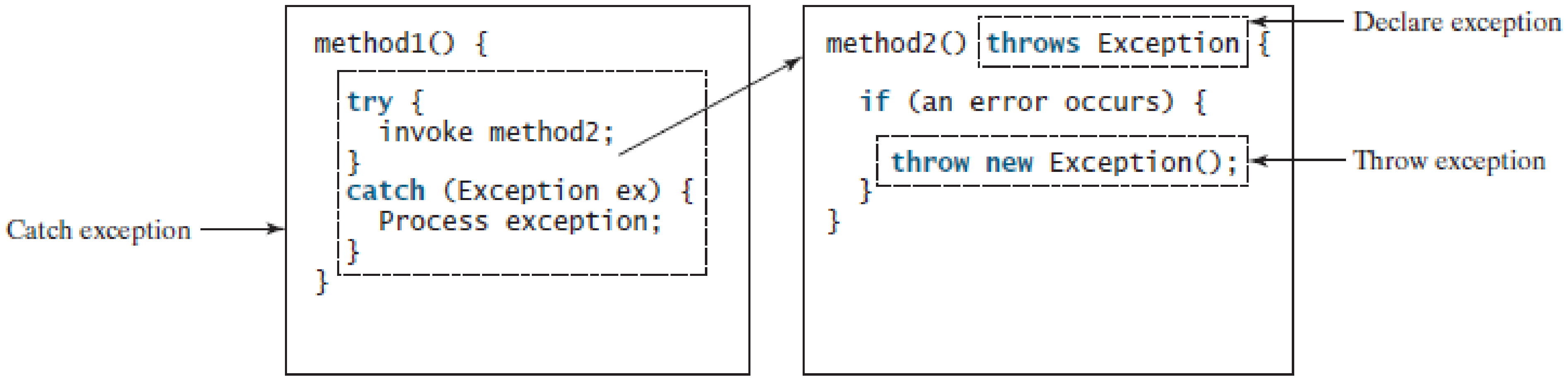
Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End of File), and `FileNotFoundException`.

Exception Types: Checked and Unchecked Exceptions

- *RunTimeException*, *Error* and their subclasses are known as *unchecked exceptions*.
  - In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable.
  - To avoid cumbersome overuse of try-catch blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.
- All other exceptions are known as *checked exceptions*.
  - The compiler forces the programmer to check and deal with them in a try-catch block or declare it in the method header.

More On Exception Handling

- Java exception handling model is based on three operations:
  - Declaring an exception.
  - Throwing an exception.
  - Catching an exception.



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

More On Exception Handling Declaring Exceptions

- Every method must state the types of *checked exceptions* it might throw.
  - This is known as *declaring exceptions*.
  - Java does not require that you declare unchecked exceptions explicitly in the method.
- To declare an exception in a method, use the *throws* keyword in the method header.
- Example:

  public void myMethod() throws IOException

More On Exception Handling Declaring Exceptions (Cont.)

- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas after throws:

  public void myMethod() throws Exception1, Exception2, …, ExceptionN

- If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

More On Exception Handling
Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it.
  - This is known as *throwing an exception*.
- Example:

  Suppose the program detects that a negative argument is passed when it should be nonnegative, the program can create an instance of *IllegalArgumentException* and throw it as follows:

  IllegalArgumentException ex = new IllegalArgumentException ("Wrong Argument");

  throw ex;

  OR

  throw new IllegalArgumentException ("Wrong Argument");

More On Exception Handling
Throwing Exceptions (Cont.)

- In general, each exception class in the Java API has at least two constructors:
  - A no-arg constructor, and
  - A constructor with a String argument that describes the exception.
    - The argument is called the *exception message*, which can be obtained using *getMessage()*;
- Note that:
  - The keyword to declare an exception is *throws*.
  - The keyword to throw an exception is *throw*.

More On Exception Handling
Catching Exceptions

- When an exception is thrown, it can be caught and handled in a *try-catch* block, as follows:

```
try{
        statements; //statements that may throw exception
}
catch (Exception1 exVar1){
        handler for exception1;
}
catch (Exception2 exVar2){
        handler for exception2;
}
…
catch (ExceptionN exVarN){
        handler for exceptionN;
}
```

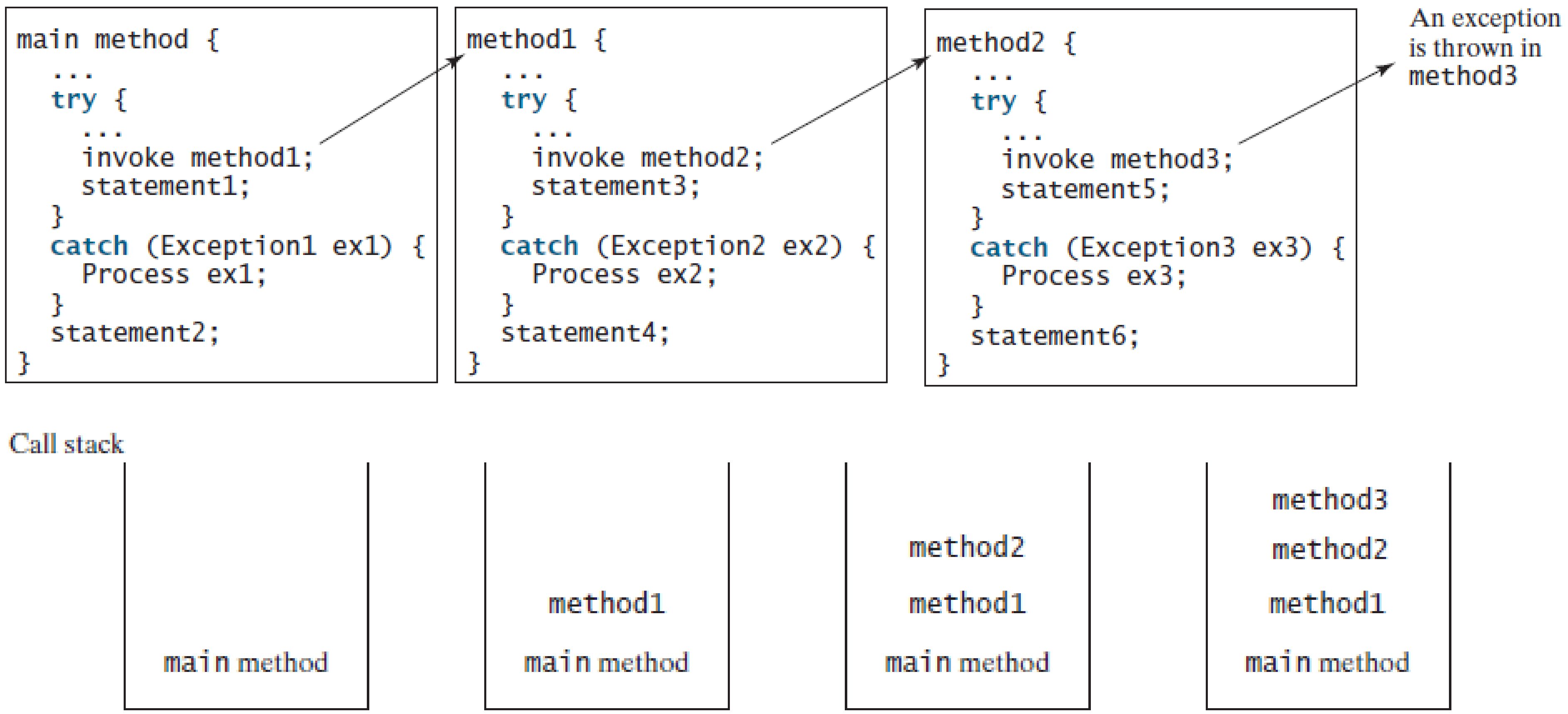
More On Exception Handling Catching Exceptions (Cont.)

- If no exceptions arise during the execution of the *try* block, the *catch* blocks are skipped.
- If one of the statements inside the *try* block throws an exception:
  - Java skips the remaining statements in the *try* block, and
  - Starts the process of finding the code to handle the exception, which is called *catching an exception*.
    - The code that handles the exception is called the *exception handler*.

More On Exception Handling Catching Exceptions (Cont.)

- An *exception handler* is found by *propagating the exception* backward through a chain of method calls, starting from the current method.
- Each catch block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block
  - If so, the exception object is assigned to the variable declared, and the code in the catch block is executed.
  - If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.
  - If no handler is found in the chain of methods being invoked, the program terminates and prints an error message to the console.

Catching Exceptions: An Example



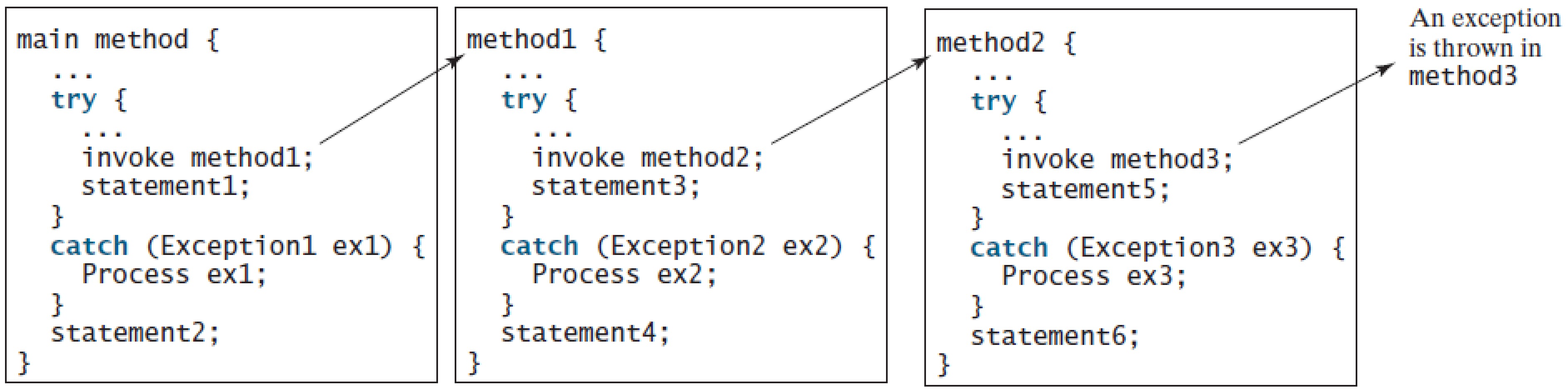
```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

Catching Exceptions: An Example (Case 1)



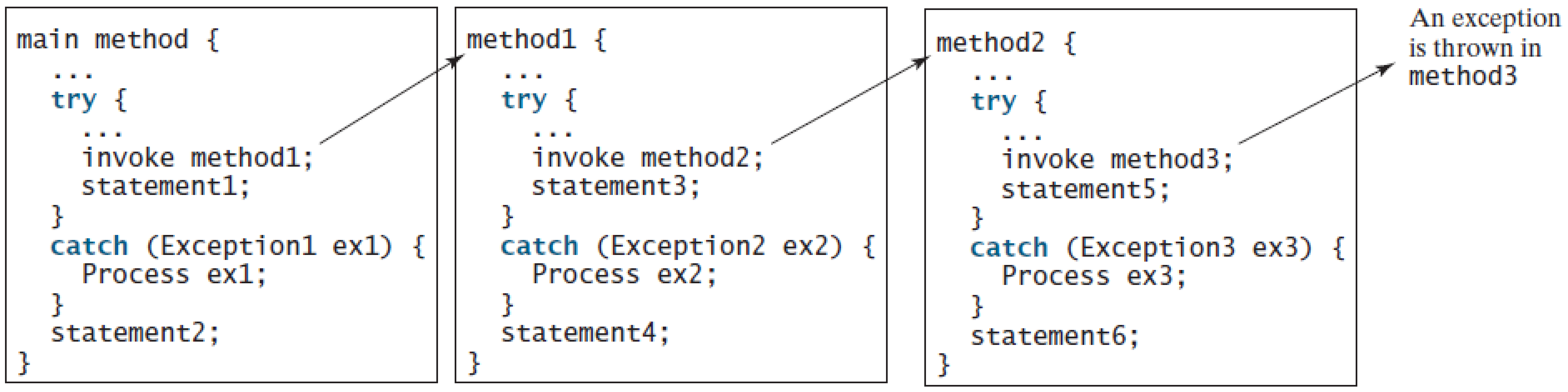
```
main method {
   ...
   try {
     ...
     invoke method1;
     statement1;
   }
   catch (Exception1 ex1) {
     Process ex1;
   }
   statement2;
}
```

```
method1 {
   ...
   try {
     ...
     invoke method2;
     statement3;
   }
   catch (Exception2 ex2) {
     Process ex2;
   }
   statement4;
}
```

```
method2 {
   ...
   try {
     ...
     invoke method3;
     statement5;
   }
   catch (Exception3 ex3) {
     Process ex3;
   }
   statement6;
}
```

- If the exception type is Exception3:
  - It is caught by the catch block for handling exception ex3 in method2.
  - Statement 5 is skipped, and statement6 is executed.

Catching Exceptions: An Example (Case 2)



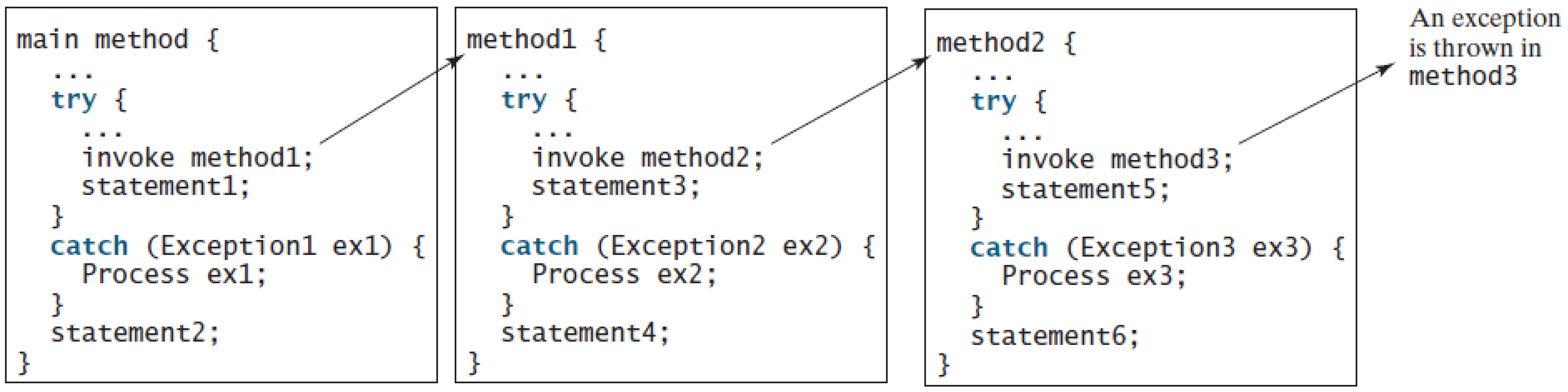
```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

- If the exception type is Exception2:
  - Method2 is aborted, the control is returned to method1.
  - The exception is caught by the catch block for handling exception ex2 in method1.
  - Statement3 is skipped, and statement4 is executed.

Catching Exceptions: An Example (Case 3)



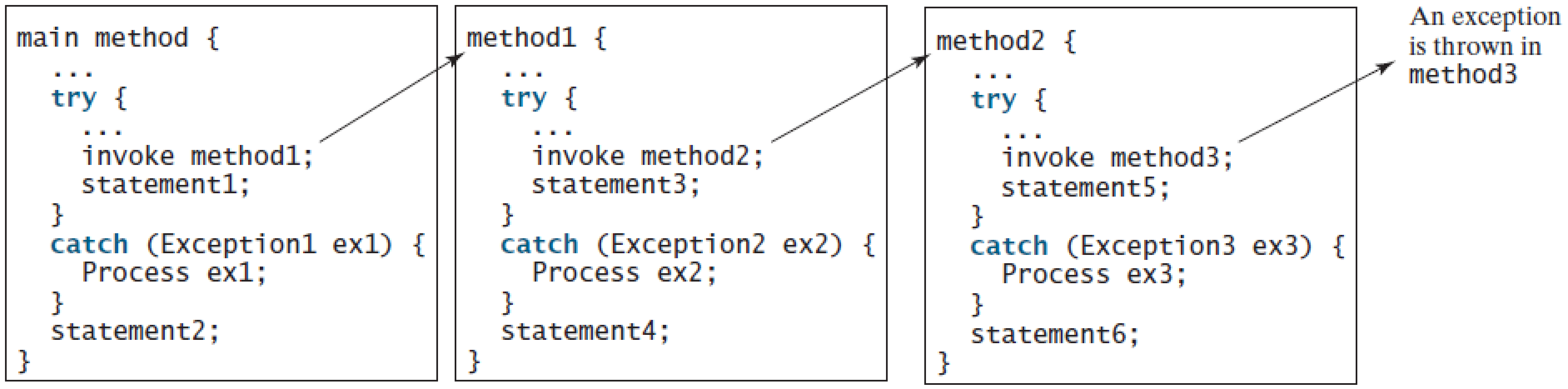
```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

- If the exception type is Exception1:
  - Method2 and method1 are aborted, the control is returned to the main method.
  - The exception is caught by the catch block for handling exception ex1 in the main method.
  - Statement1 is skipped, and statement2 is executed.

Catching Exceptions: An Example (Case 4)


```
main method {                     method1 {                         method2 {
   ...                               ...                               ...
   try {                             try {                             try {
     ...                               ...                               ...
     invoke method1;                   invoke method2;                   invoke method3;
     statement1;                       statement3;                       statement5;
   }                                 }                                 }
   catch (Exception1 ex1) {          catch (Exception2 ex2) {          catch (Exception3 ex3) {
     Process ex1;                      Process ex2;                      Process ex3;
   }                                 }                                 }
   statement2;                       statement4;                       statement6;
}                                 }                                 }
```

- If the exception type is not caught in method2, method1, or the main method:
  - Program terminates, and statement1 and statement2 are not executed.

More on Catching Exceptions

- Various exception classes can be derived from a common superclass.
  - If a *catch* block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of the superclass.
- The order in which exceptions are specified in *catch* blocks is important.
  - A compile error will result if a *catch* block for a superclass type appears before a *catch* block for a subclass type.

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

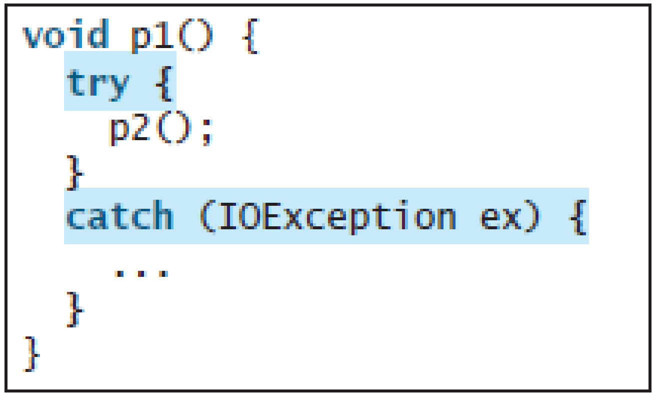
(a) Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

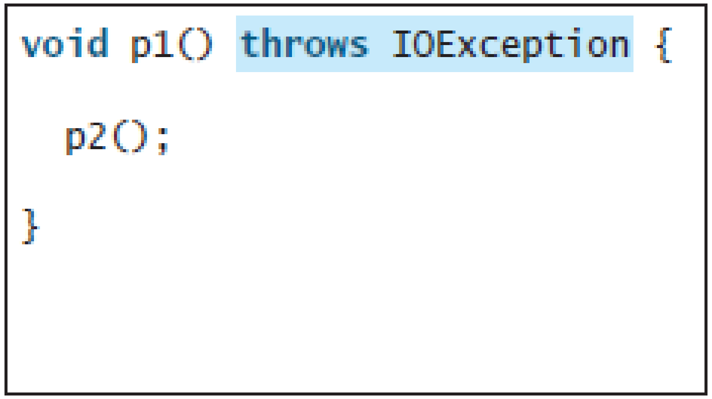
More on Catching Exceptions (Cont.)

- Java forces you to deal with checked exceptions.
- If a method declares a checked exception (i.e., an exception other than *Error* or *RuntimeException*), 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 shown in (a) or (b) below.



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

(a) Catch exception



```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

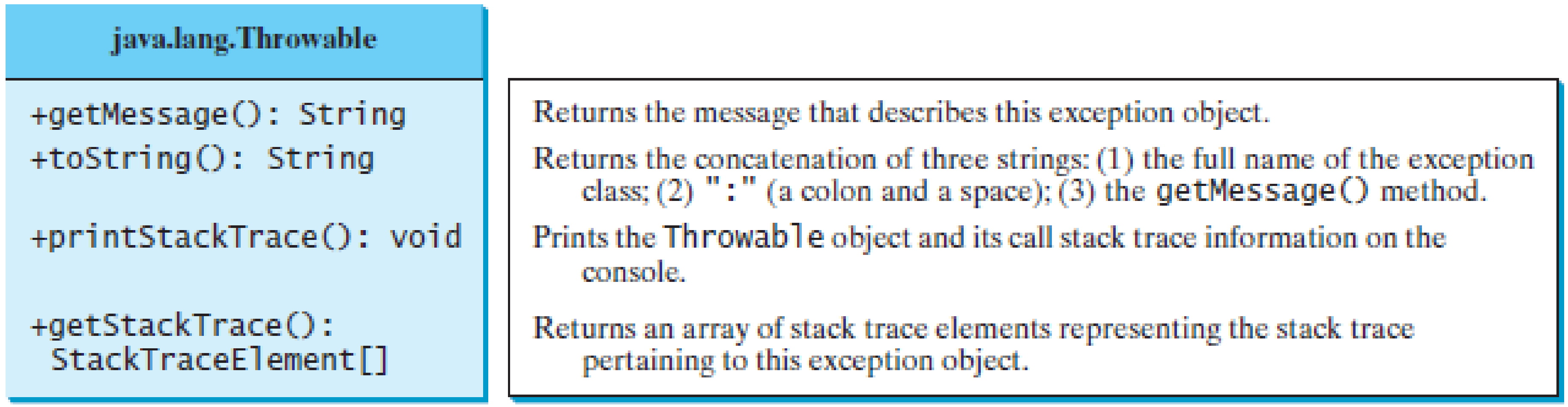
More on Catching Exceptions (Cont.)

- You can use the new JDK 7 multi-catch feature to simplify coding for the exceptions with the same handling code.
- The syntax is:

```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
  // Same code for handling these exceptions
}
```

Getting Information from Exceptions

- An exception object contains valuable information about the exception.

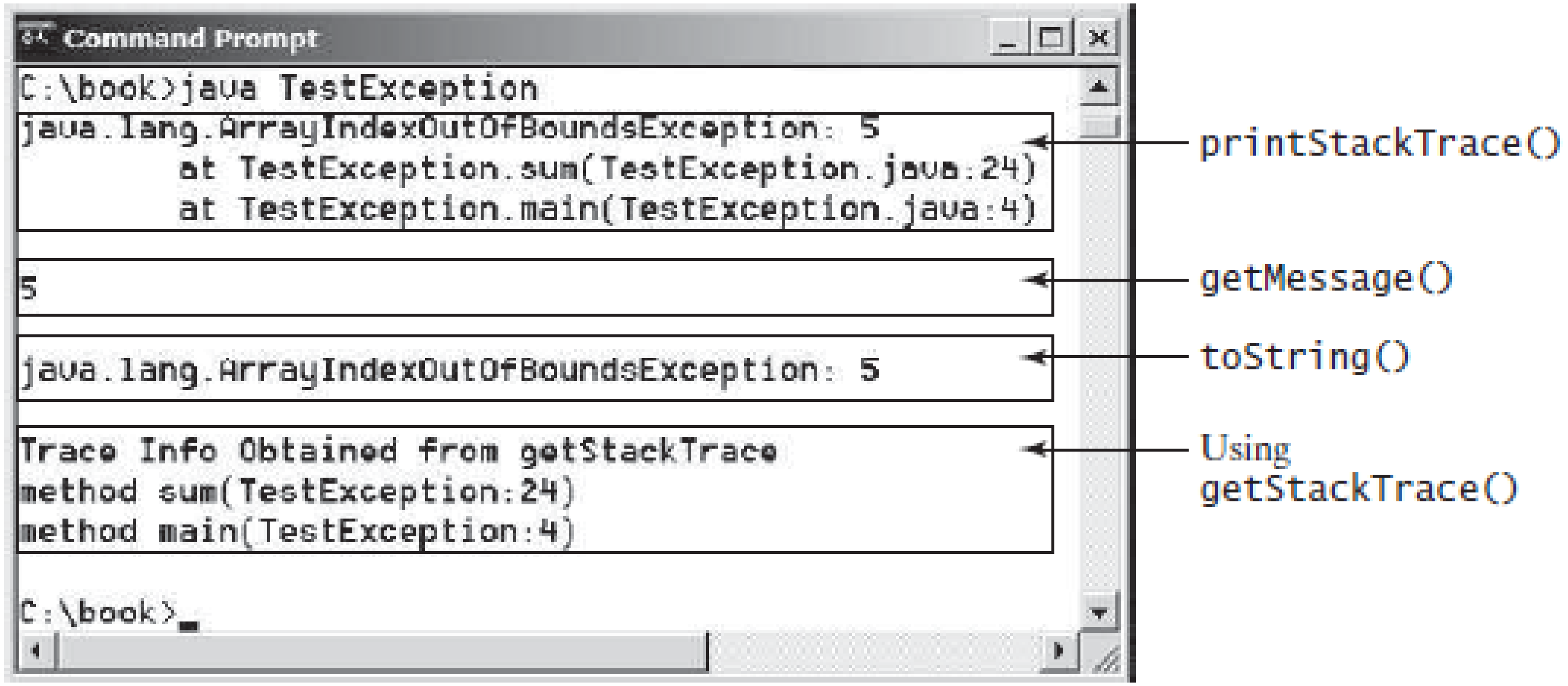


java.lang.Throwable	
`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `":"` (a colon and a space); (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

TestException.java

```java
public class TestException  {
  public static void main(String[] args) {
    try {
      System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.out.println("\n" + ex.getMessage());
      System.out.println("\n" + ex.toString());

      System.out.println("\nTrace Info Obtained from getStackTrace");
      StackTraceElement[] traceElements = ex.getStackTrace();
      for (int i = 0; i < traceElements.length; i++) {
        System.out.print("method " + traceElements[i].getMethodName());
        System.out.print("(" + traceElements[i].getClassName() + ":");
        System.out.println(traceElements[i].getLineNumber() + ")");
      }
    }
  }

  private static int sum(int[] list) {
    int result = 0;
    for (int i = 0; i <= list.length; i++)
      result += list[i];
    return result;
  }
}
```

TestException.java (Output)



```
C:\book>java TestException
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:24)
        at TestException.main(TestException.java:4)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:24)
method main(TestException:4)

C:\book>_
```

Example: Declaring, Throwing, and Catching Exceptions (CircleWithException.java)

```
public class CircleWithException{
   private double radius;
   private static int numberOfObjects=0;

   public CircleWithException(double newRadius){
      setRadius(newRadius);
      numberOfObjects++;
   }

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

   public static int getNumberOfObjects(){
      return numberOfObjects;
   }
}
```

Example: Declaring, Throwing, and Catching Exceptions (TestCircleWithException.java)

```
public class TestCircleWithException{
   public static void main (String [] args){
      try{
         CircleWithException C1 = new CircleWithException(5);
         CircleWithException C2 = new CircleWithException(-5);
         CircleWithException C3 = new CircleWithException(0);
      }
      catch (IllegalArgumentException ex){
         System.out.println(ex);
      }
      System.out.println("Number of circle objects created: "+
              CircleWithException.getNumberOfObjects());
   }
}
```

<u>Output:</u>

```
java.lang.IllegalArgumentException: Radius cannot be negative!
Number of circle objects created: 1
```

The *finally* Clause

- The *finally* clause is executed under all circumstances, regardless of whether an exception occurs in the *try* block or is caught.
- The syntax for the *finally* clause is as follows:

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

The *finally* Clause (Cont.)

- If no exception arises in the *try* block:
  - The *finally* clause is executed, and
  - The next statement after the *try* statement is executed.
- If a statement causes an exception in the *try* block that is caught in the *catch* block:
  - The rest of the statements in the *try* block are skipped,
  - The *catch* block is executed,
  - The *finally* clause is executed, and
  - The next statement after the *try* statement is executed.
- If a statement causes an exception that is not caught in any *catch* block:
  - The other statements in the *try* block are skipped,
  - The *finally* clause is executed, and
  - The exception is passed to the caller of this method.
- Note: the *finally* block executes even if there is a *return* statement prior to reaching the *finally* block.

When to Use Exceptions?

- The *try* block contains the code that is executed in normal circumstances.
- The *catch* block contains the code that is executed in exceptional circumstances.
- 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.
  - Requires instantiating a new exception object,
  - Rolling back the call stack, and
  - Propagating the exception through the chain of methods invoked to search for the handler.

When to Use Exceptions? (Cont.)

- An exception occurs in a method:
  - If you want the exception to be processed by the method’s 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 or use exception objects.
    - Simple errors that may occur in individual methods are best handled without throwing exceptions.
    - This can be done by using *if* statements to check for errors.

Rethrowing Exceptions

- Java allows an exception handler to *rethrow* the exception if 1) the handler cannot fully process the exception or 2) simply wants to let its caller be notified of the exception.
- The syntax for re-throwing an exception is as follows:

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

Chained Exceptions

- Throwing an exception along with another exception forms a chained exception.
- Sometimes, you may need to throw a new exception (with additional information) along with the original exception.
- This is called *chained exceptions*.

Chained Exceptions ChainedExceptionDemo.java

```java
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
       throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```

Chained Exceptions ChainedExceptionDemo.java (Output)

```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```

Defining Custom Exception Classes

- Java provides quite a few exception classes.
- Use them whenever possible instead of defining your own exception classes.
- However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from *Exception* or from a subclass of *Exception*, such as *IOException*.

Defining Custom Exception Classes InvalidRadiusException.java

```
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

  /** Return the radius */
  public double getRadius() {
    return radius;
  }
}
```

Defining Custom Exception Classes TestCircleWithCustomException.java

```java
public class TestCircleWithCustomException {
  public static void main(String[] args) {
    try {
      new CircleWithCustomException(5);
      new CircleWithCustomException(-5);
      new CircleWithCustomException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithCustomException.getNumberOfObjects());
  }
}

```

Defining Custom Exception Classes TestCircleWithCustomException.java (Cont.)

```java
class CircleWithCustomException {
  /** The radius of the circle */
  private double radius;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithCustomException() throws InvalidRadiusException {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithCustomException(double newRadius)
      throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }

```

Defining Custom Exception Classes TestCircleWithCustomException.java (Cont.)

```java
  /** Return radius */
  public double getRadius() {
    return radius;
  }

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

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

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